

VisualWorks

User's Guide

Copyright © 1995 by ParcPlace-Digitalk, Inc. All rights reserved.

Part Number: DS10005004

Revision 2.1, October 1995 (Software Release 2.5)

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

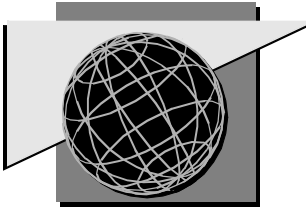
Trademark acknowledgments:

ObjectKit, ObjectWorks, ParcBench, ParcPlace, and VisualWorks are trademarks of ParcPlace Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. DataForms, MethodWorks, ObjectLens, ObjectSupport, ParcPlace Smalltalk, Visual Data Modeler, VisualWorks Advanced Tools, VisualWorks Business Graphics, VisualWorks Database Connect, VisualWorks DLL and C Connect, and VisualWorks ReportWriter are trademarks of ParcPlace Systems, Inc., its subsidiaries, or successors. ENVY is a registered trademark of Object Technology International, Inc. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

VisualWorks is furnished under a license and may not be used, copied, disclosed and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 1995 by ParcPlace-Digitalk, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from ParcPlace-Digitalk.



Contents

	About This Book	xvii
	Audience	xvii
	Organization	xviii
	Conventions	xix
	Typographic Conventions	xix
	Special Symbols	xx
	Screen Conventions	xx
	Mouse Buttons	xx
	Mouse Operations	xxii
	Additional Sources of Information	xxii
	Printed Documentation	xxii
	Online Documentation	xxiii
	Obtaining Technical Support	xxiv
	Before Contacting Technical Support	xxiv
	How to Contact Technical Support	xxiv
Chapter 1	Introduction	1
	About VisualWorks	1
	Building Applications	1
	Read-and-Apply Tools	2
	Visual Reuse	2
	Starting VisualWorks	2
	Saving Your Image	3
	Exiting VisualWorks	4
Part I	Smalltalk Language	5
Chapter 2	Object Orientation	7
	Procedures vs. Objects	7
	Objects and Methods	8

- Composite Objects 9
- Variables and Methods 11
- Method Grouping 12
- Classes and Instances 13
 - Class Variables 13
 - Class Methods vs. Instance Methods 14
 - Class Grouping 15
 - Class Inheritance 16
- Looking up a Method 17
 - Overriding an Inherited Method 18
- Abstract Classes 19
 - Nesting Abstract Classes 20
- Choosing a Superclass 20

Chapter 3

Syntax

23

- Naming Conventions 23
 - Capitalization Rules and Conventions 24
- Literal Constants 24
 - Numbers 24
 - Characters 26
 - Strings 26
 - Symbols 27
 - Byte Arrays 27
 - Arrays 27
 - Booleans 27
- Variables 28
 - Temporary Variables 29
 - Instance Variables 31
 - Class Instance Variables 31
 - Class Variables 32
 - Pool Variables 33
 - Global Variables 34
 - Special Variables 34
 - Undeclared Variables 36
- Message Expressions 36
 - Unary Messages 37
 - Binary Messages 37
 - Keyword Messages 39
 - Messages in Sequence 40
- Block Expressions 42
 - Formatting Conventions 44
- Syntactic Elements Summary 44

Chapter 4	Control Structures	47
	Branching 47	
	ifTrue:ifFalse: 47	
	Looping 48	
	Conditional Looping 48	
	Number Iteration 49	
	Collection Iteration 50	
Chapter 5	Numeric Operations	55
	Integers 55	
	Floating Point Numbers 57	
	Fractions 59	
	Random Numbers 59	
	Dates 60	
	Time 63	
	Time Zone 66	
	Abstract Superclasses 67	
Chapter 6	Collection Operations	69
	Choosing the Appropriate Class 69	
	Set 70	
	Bag 70	
	Array 70	
	Interval 71	
	OrderedCollection 71	
	SortedCollection 71	
	LinkedList 71	
	Dictionary 72	
	Creating an Instance 73	
	Adding, Removing and Replacing Elements 74	
	Comparing Collections 76	
	Counting and Finding Elements 77	
	Copying a Collection 78	
	Converting and Printing 78	
	The Collection Hierarchy 78	
Chapter 7	String Operations	81
	Creating a Character 81	
	Character Operations 82	
	Creating a String 83	
	Substring Manipulations 83	

Pattern Matching 85
 The String Hierarchy 85

Chapter 8 Processes and Exception Handling 87

Creating a Process 87
 Scheduling a Process 88
 Setting the Priority Level 89
 Coordinating Processes with a Semaphore 90
 Passing Data Between Processes 92
 Using a Delay 92
 Using a Signal to Handle an Error 92
 Choosing or Creating a Signal 94
 Creating an Exception 96
 Setting Parameters 97
 Passing Control From the Handler Block 98
 Using Nested Signals 99
 Unwind Protection 100

Part II VisualWorks Tools 101

Chapter 9 Environment Tools 103

VisualWorks Main Window 103
 Settings Tool 104
 File List 105
 File List Views 106
 Display Options 106
 File List Commands 107
 Change List 109
 File Editor 113
 Project 113

Chapter 10 Smalltalk Programming Tools 115

System Browser 115
 Structure 115
 Class Categories 116
 Classes 116
 Protocols 116
 Methods 116
 Code 116
 Workspace 124

	System Transcript	125	
	Debugger	126	
	Inspector	130	
Chapter 11	Application Building Tools		131
	Resource Finder	131	
	Canvas Tool	132	
	Palette	133	
	Image Editor	135	
	Menu Editor	135	
	Enhanced Menu Editor	136	
	Properties Tool	140	
	Basics Properties	142	
	Details Properties	143	
	Validation Properties	144	
	Notification Properties	146	
	Color Properties	147	
	Position Properties for Bounded Widgets	149	
	Drop Source Properties	150	
	Drop Target Properties	151	
	Define Dialog	154	
Chapter 12	Database Application Building Tools		157
	The Data Modeler	157	
	Canvas Composer	157	
	VisualWorks Painting Tools	157	
	Embedded and Linked Data Forms	158	
	Mapping Tool	158	
	The Query Editor	158	
	Menu Queries	158	
	Ad Hoc SQL Editor	159	
Chapter 13	Application Delivery Tools		161
	Parcel List	161	
	Parcel Menu Commands	162	
	Utility Menu Commands	163	
	Parcel Browser	163	
	Structure	164	
	Parcel View	165	
	Category View	166	
	Class View	166	

	Protocol View	167	
	Method View	168	
	Code View	168	
	Image Maker	169	
Chapter 14	Debugging Techniques		171
	Reading the Execution Stack	171	
	Tracing the Flow of Messages	173	
	Inspecting and Changing Variables	175	
	Inserting Status Messages	176	
	Interrupting a Program	177	
	Restarting a Program	177	
Chapter 15	Managing Projects and Versions		179
	Entering and Exiting a Project	179	
	Summarizing Project Changes	180	
	Reverting to a Prior Version	182	
	Sharing Code	184	
	Condensing the Changes File	185	
Chapter 16	Accessing Databases		187
	Overview	187	
	Data Interchange	189	
	Establishing a Connection	190	
	Securing Passwords	190	
	Getting the Details Right	191	
	Setting a Default Environment	191	
	Default Connections	192	
	On the Importance of Disconnecting	193	
	Using Sessions	193	
	Variables in Queries	194	
	Named Input Binding	196	
	Getting Answers	197	
	Handling Multiple Answer Sets	198	
	What Happens when you Send an Answer Message	198	
	Waiting for the Server	199	
	Did the Query Succeed?	199	
	How Many Rows were Affected?	199	
	Describing the Answer Set	200	
	Buffers and Adaptors	200	
	Processing an Answer Stream	201	

	Using an Output Template	202
	Setting a Block Factor to Improve Performance	204
	Cancelling an Answer Set	205
	Disconnecting the Session	205
	Catalog Queries	206
	Controlling Transactions	207
	Coordinated Transactions	207
	Releasing Resources	207
	Tracing the Flow of Execution	208
	Directing Trace Output	208
	Setting the Trace Level	209
	Disabling Tracing	209
	Adding Your Own Trace Information	209
	Error Handling	210
	Signals and Error Information	210
	Exception Handling	211
	The Database Signal Hierarchy	212
	Choosing an Exception to Handle	212
	Image Save and Restart Considerations	213
Chapter 17	Troubleshooting	215
	Recovering from a System Failure	215
	Start-up Errors	216
	Source Code Unavailable in Browser	217
	Low Space	217
	No VisualWorks Main Window	217
	Can't Exit from VisualWorks	218
	UNIX	218
	Macintosh	218
	Windows	219
	Emergency Exit (all platforms)	219
	When You Need Assistance	219
Part III	Application Components	221
Chapter 18	Application Framework	223
	Overview	224
	Domain Model Is Separate From User Interface	224
	ApplicationModel Acts as Mediator	225
	Value Model Links Widget to Attribute	226
	Builder Assembles User Interface	227

- Widget Has Visual Component and Optional Controller 228
- About the Example Application 229
- Domain Model 231
 - Overview 231
 - Data Storage 232
 - Data Processing 233
- Application Model 234
 - Overview 234
 - Storage of Reusable Labels and Images 235
 - Storage of Interface Specs 236
 - Storage of Value Models 236
 - Dependent Notification 237
 - Application Startup 245
 - Application Cleanup 247
- Builder 248
 - Overview 248
 - Storage of UI Bindings 249
 - Interface Assembly 251
 - Interface Opening 252
 - Window Access 253
 - Named Component Access 253
- Window 254
 - Overview 254
 - Damage Repair 256
- Visual Component 257
 - Overview 257
 - Passive vs. Active Components 258
 - Autonomous vs. Dependent Components 259
 - Controller Linking 261
 - Model Linking 261
 - Redisplaying 261
 - Composite Visual Component 262
 - Wrapper 264
- Controller 264
 - Polling vs. Event-Driven Controllers 266
 - Flow of Control (Polling Controller) 267
 - Flow of Events (Event-Driven Controller) 270
 - Selection Tracking 271

Chapter 19 Graphic Operations

273

- Background 274
 - Coordinate System 274

- Points 276
- Rectangles 277
- Display Surfaces 280
 - VisualWorks Windows 281
 - Pixmap 281
 - Masks 282
 - Host Residency of Display Surfaces 283
 - Graphics Context 284
- Graphic Objects 289
 - Texts 289
 - Lines, Polylines and Polygons 290
 - Splines and Bezier Curves 291
 - Arcs, Circles and Wedges 291
 - Graphical Images 293
 - Image Processing 296
 - Bit Processing 297
 - CachedImage 297
 - Cursors 298
 - Icons 299
 - Animation 299
- Integrating Graphics into an Application 300
 - Integrating a Static Graphic 301
 - Integrating a Dynamic Graphic 302

Chapter 20 Color 303

- Types of Color 303
 - Pattern 303
 - Coverage 304
 - Color 304
- Palettes 307
 - Coverage Palettes 307
 - Color Palettes 308
 - Device Color Map 310
- Policies for Rendering Color 311
 - NearestPaint 312
 - OrderedDither 313
 - ErrorDiffusion 313

Chapter 21 Weak Arrays and Finalization 315

- Weak Arrays 315
- Finalization 316
- WeakDictionary 319

	HandleRegistry	319
	Finalization Example	320
Chapter 22	Parsing and Compiling	323
	Scanner	323
	Parser	324
	Compiler	325
Chapter 23	Memory Management	327
	Memory Layout	327
	Fixed-size OE Spaces	328
	Smalltalk Object Memory	333
	Facilities for Reclaiming Space	335
	Generation Scavenger	336
	Incremental Garbage Collector	336
	Compacting Garbage Collector	338
	Global Garbage Collector	338
	Data Compactor	339
	Memory Policy Classes	339
	ObjectMemory	339
	MemoryPolicy	340
Part IV	Application Delivery	343
Chapter 24	Overview of Application Delivery	345
	Different Ways to Deliver an Application	345
	Single Image File	345
	Parcels	346
	Development and Deployment Life-Cycle	346
	Method 1: Delivery Combined with Development	346
	Method 2: Delivery After Development	347
	More Information	347
Chapter 25	Parceling an Application	349
	What Are Parcels?	349
	Characteristics	350
	Contents	350
	Restrictions	350
	Parcel Files	351
	Creating Parcels	351

Deciding What to Parcel	351	
Specifying Parcels and their Contents	352	
Loading Parcels	356	
At Start Up	356	
From within an Application	356	
Behavior at Load Time	358	
Load Order	358	
Load Errors	358	
Filing Parcel Contents In and Out	359	
Tips for Working with Parcels	359	
Keeping Source Code and Parcels in Sync	359	
Testing Parcel Files and Source Files for Matches	360	
Chapter 26	Creating a Deployment Image	361
Setting Up a Deployment Image	362	
Handling Errors	362	
The Transcript	362	
Undeclared Variables	362	
Creating a Deployment Image	363	
Operations Performed by Image Maker	366	
Removal of Development Facilities	366	
Optional Removal of Other Facilities	366	
Preservation of Certain Facilities	370	
Optimization of Memory Usage	370	
Other Changes	371	
Saving the State of Image Maker	371	
Starting Up a Deployed Image	371	
Debugging a Deployed Image	372	
Exiting a Deployed Image	373	
Chapter 27	Creating Applications without Graphical User Interfaces	375
Key Concepts	375	
Setting Up a Headless Image	376	
Running an Application in Headless Mode	377	
When an Image Starts	377	
If an Application Attempts to Access a Display	378	
Debugging a Suspended Process	378	
Creating a Headful Copy of a Headless Image	379	
Tips for Programming a Headless Application	379	
Techniques for Starting a Headless Application	379	

Techniques for Communicating with a Headless Application	380
Terminating a Headless Application	380
Preventing Access to the Display	380
Delivering a Headless Application	381

Part V Appendixes 383

Appendix A Protocol Reference 385

Common Class Protocols	385
Common Instance Protocols	386

Appendix B Syntax Descriptions 387

Lexical Primitives	387
Character Classes	388
Numbers	389
Other Lexical Constructs	390
Atomic Terms	391
Expressions and Statements	392
Methods	394

Appendix C Special Characters 395

Composed Characters	395
Diacritical Marks	399

Appendix D Implementation Limits 401

Size Limitations	401
Open-coded Blocks	402
Shared Context	403
Browser Visibility	404
Block Optimization	404
The Debugger	406
Performance	406
Non-overrideable Methods	407
Special Treatment Only at Compile Time	407
Special Treatment at Compile Time and Translation Time	409

Appendix E Keyboard Shortcuts 411

Editing Text and Components	411
Displaying Tools and Dialogs	411
Selecting Components	411

Moving Components	412
Aligning Components	412
Grouping Components	412
Changing Layouts	412
Changing Tool Focus	412

Appendix F User-Defined Primitives

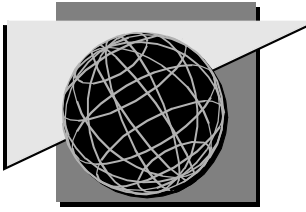
413

Theory of Operation	413
Basic Capabilities	414
Defining a New Primitive	414
Installation and Access	416
Primitive Numbers	417
Arguments	417
Data Types	417
Failure Codes	418
General Advice	419
C Conversion	421
String to String	421
Byte Array to Byte Object	421
Integer Array to Array	421
Float Array to Array	422
Integer to Integer	422
Float to Float	422
Double Float to Double	422
Boolean to Boolean	423
Character to Character	423
Return nil	423
Smalltalk Conversion	423
String to String	423
Byte Array to Byte Array	424
Integer Array to Array	424
Float Array to Array	424
Integer to Integer	425
Float to Float	425
Double Float to Double	425
Character to Character	425
Boolean to Boolean	425
Success Return	426
Any Value	426
Nil	426
True	426
False	426

- Failure Return 426
 - Coded Failure 427
- Type Checking 427
 - Character 427
 - String 427
 - Integer 428
 - Float 428
 - Double 428
 - Array of Integers 429
 - Array of Floats 429
 - Byte Array 429
 - Byte-like 430
 - Boolean 430
 - Immediate 430
 - Class Check 431
- Object Allocation 431
 - String 431
 - Byte Array 431
 - Array 431
 - Other Object Types 432
- Indexed Access 432
 - Indexed Variable 432
 - Instance Variable 433
 - Indexed Byte 433
 - Indexed Float 434
- Sizing 434
- Initializing 434
- Other Support Routines 435
- Registering Long-lived Objects 436
- Interrupts and Poll Handlers 437
- Unsafe Primitives 439
- Example 440
 - C Code 440
 - Smalltalk Code 442

Index

443



About This Book

The purpose of the *VisualWorks User's Guide* is to show you how to use VisualWorks® to quickly create applications that employ graphical user interfaces. It also provides detail about the ParcPlace Smalltalk™ syntax, programming tools such as the Debugger, and advanced facilities such as exception handling.

Descriptions are also included that tell how to leverage your development resources further by implementing embedded interfaces and applications. We also show how to build database access into your applications. In the appendix, you will find keyboard shortcuts, message categories reference, syntax descriptions, and special characters.

Audience

This guide address two primary audiences:

- n developers of user interfaces
- n developers of the models that support those interfaces

Both kinds of developers should at least be familiar with Smalltalk syntax and object-oriented programming concepts, as described in the *VisualWorks Tutorial* and the first part of the *VisualWorks User's Guide*. If you intend to develop complex application models or customize the standard components, you will need a more thorough understanding of the Smalltalk class library and the basic programming tools, which you can get from the *VisualWorks User's Guide*.

Organization

This *VisualWorks User's Guide* provides comprehensive instructions for using Smalltalk and VisualWorks. It is divided into five parts:

- n Smalltalk Language
- n VisualWorks Tools
- n Application Components
- n Application Delivery
- n Appendixes

The first part, Smalltalk Language, provides a detailed discussion of the Smalltalk language. The language is largely implemented via Smalltalk classes, as are user-interface components. It begins with an overview of object-oriented programming, providing a bridge that links conventional programming concepts to the sometimes unfamiliar terminology of Smalltalk. Subsequent chapters examine Smalltalk syntax, control structures, fundamental data structures, processes and exception handling.

The second part, VisualWorks Tools, provides detailed descriptions of the tools that are available in the VisualWorks environment. These tools help you manage projects, edit and compile code, trace bugs, edit text files, and more.

The third part, Application Components, provides detailed instructions for using the many reusable software modules that are available in VisualWorks. While you can reuse any part of the system in your applications, this manual selects the more commonly reused components for inspection, such as views and dialogs.

The fourth part, Application Delivery, describes the process for extracting applications from VisualWorks in a form that makes them ready for use by your intended end users. It includes information about breaking applications into separately-loadable units called parcels and creating a deployment image.

Conventions

This section describes the notational conventions used to identify technical terms, computer-language constructs, mouse buttons, and mouse and keyboard operations.

Typographic Conventions



This book uses the following fonts to designate special terms:

Example	Description
<i>template</i>	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
cover.doc	Indicates filenames, pathnames, commands, and other C++, UNIX, or DOS constructs to be entered outside VisualWorks (for example, at a command line).
<i>filename.xwd</i>	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File?New command	Indicates the name of an item on a menu.
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.

	Examples	Description
	Float class>>pi	Indicates a class method defined in a class.
	Caution:	Indicates information that, if ignored, could cause loss of data.
	Warning:	Indicates information that, if ignored, could damage the system.

Screen Conventions

This manual contains a number of sample screens that illustrate the results of various tasks. The windows in these sample screens are shown in the default Smalltalk look, rather than the look of any particular platform. Consequently, the windows on your screen will differ slightly from those in the sample screens.

Mouse Buttons

Many hardware configurations supported by VisualWorks have a three-button mouse, but a one-button mouse is the standard for Macintosh users, and a two-button mouse is common for OS/2 and Windows users. To avoid the confusion that would result from referring to <Left>, <Middle>, and <Right> mouse buttons, this book instead employs the logical names <Select>, <Operate>, and <Window>.

The mouse buttons perform the following interactions:

<Select> button	<i>Select</i> (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of <i>operations</i> that are appropriate for the current view or selection. The menu that is displayed is referred to as the <i><Operate> menu</i> .
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks <i>window</i> (except dialogs), such as move and close . The menu that is displayed is referred to as the <i><Window> menu</i> .

Three-Button Mouse

VisualWorks uses the three-button mouse as the default:

- n The left button is the <Select> button.
- n The middle button is the <Operate> button.
- n The right button is the <Window> button.

Two-Button Mouse

On a two-button mouse:

- n The left button is the <Select> button.
- n The right button is the <Operate> button.
- n To access the <Window> menu, you press the <Control> key and the <Operate> button together.

One-Button Mouse

On a one-button mouse:

- n The unmodified button is the <Select> button.
- n To access the <Operate> menu, you press the <Option> key and the <Select> button together.
- n To access the <Window> menu, you press the <Command> key and the <Select> button together.

Mouse Operations

The following table explains the terminology used to describe actions that you perform with mouse buttons.

When you see:	Do this:
click	Press and release the <Select> mouse button.
double-click	Press and release the <Select> mouse button twice without moving the pointer.

When you see:	Do this:
<Shift>-click	While holding down the <Shift> key, press and release the <Select> mouse button.
<Control>-click	While holding down the <Control> key, press and release the <Select> mouse button.
<Meta>-click	While holding down the <Meta> or <Alt> key, press and release the <Select> mouse button.

Additional Sources of Information

Printed Documentation

In addition to this User's Guide, the core VisualWorks documentation includes the following documents:

- n *Installation Guide*: Provides instructions for the installation and testing of VisualWorks on your combination of hardware and operating system.
- n *Release Notes*: Describes the new features of the current release of VisualWorks.
- n *Tutorial*: This manual provides an introduction to the concepts and skills needed by the new VisualWorks tasks.
- n *Cookbook*: Provides step-by-step instructions for performing hundreds of common VisualWorks tasks.
- n *International User's Guide*: Describes the VisualWorks facilities that support the creation of nonEnglish and cross-cultural applications.
- n *Object Reference*: Provides detailed information about the VisualWorks class library.

The documentation for the VisualWorks database tools consists of the following documents:

- n *VisualWorks' Database Tools Tutorial and Cookbook*: Introduces the process and tools for creating applications that access relational databases. The "Cookbook" chapter describes how to programmatically customize various aspects of a database application.
- n *Database Connect User's Guide*: Provides information about the external database interface. Versions of it exist for SYBASE, ORACLE7, and DB2 databases.

Online Documentation

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select **Open Online Documentation**. Your choice of online books includes:

- n *Database Cookbook*: Online version of the “Cookbook” part of the *VisualWorks’ Database Tools Tutorial and Cookbook* described above.
- n *Database Quick Start Guides*: Describes how to build database applications. It covers such topics as data models, single- and multiwindow applications, and reusable data forms.
- n *International User’s Guide*: Online version of the *International User’s Guide* described above.
- n *VisualWorks Cookbook*: Online version of the *Cookbook* described above.
- n *VisualWorks DLL and C Connect Reference*: Describes C data classes, object engine access functions, and user-primitive functions.

Obtaining Technical Support

If, after reading the documentation, you find that you need additional help, you can contact ParcPlace-Digitalk Technical Support. ParcPlace-Digitalk provides all customers with help on product installation. ParcPlace-Digitalk provides additional technical support to customers who have purchased the ObjectSupport package. VisualWorks distributors often provide similar services.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- n The *version id*, which indicates the version of the product you are using. Choose **Help?About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id**.
- n Any modifications (*patch files*) distributed by ParcPlace-Digitalk that you have imported into the standard image. Choose **Help?About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches**.

- n The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

How to Contact Technical Support

ParcPlace-Digitalk Technical Support provides assistance by:

- n Electronic mail
- n Electronic bulletin boards
- n World Wide Web
- n Telephone and fax

Electronic Mail

To get technical assistance on the VisualWorks line of products, send electronic mail to **support-vw@parcplace.com**.

Electronic Bulletin Boards

Information is available at any time through the electronic bulletin board CompuServe. If you have a CompuServe account, enter the ParcPlace-Digitalk forum by typing **go ppdforum** at the prompt.

World Wide Web

In addition to product and company information, technical support information is available via the World Wide Web:

1. In your Web browser, open this location (URL):
<http://www.parcplace.com>
2. Click the link labeled “Tech Support.”

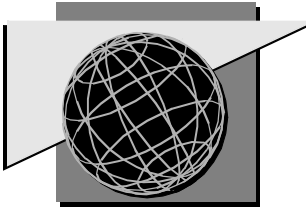
Telephone and Fax

Within North America, you can:

- n Call ParcPlace-Digitalk Technical Support at 408-773-7474 or 800-727-2555.
- n Send questions and information via fax at 408-481-9096.

Operating hours are Monday through Thursday from 6:00 a.m. to 5:00 p.m., and Friday from 6:00 a.m. to 2:00 p.m., Pacific time.

Outside North America, you must contact the local authorized reseller of ParcPlace-Digitalk products to find out the telephone numbers and hours for technical support.



Chapter 1

Introduction

About VisualWorks

VisualWorks is a fully object-oriented environment for constructing applications, using Smalltalk as the scripting language. It enables application developers to build graphical user interfaces rapidly for both new and existing applications, augmenting Parcplace Smalltalk development facilities. In addition, VisualWorks provides convenient linkages to many popular databases such as ORACLE and SYBASE.

Building Applications

You begin building an application by using a VisualWorks painter to place visual components on a canvas. The characteristics of components on the canvas are described by a variety of property-setting, menu-building, aligning and positioning tools. The VisualWorks painter creates the user interface.

A Definer and object script browsers work together to create the application logic. This is the “glue” that connects the components on the canvas with one another, and with information obtained from an underlying domain model. The domain model can reside in an external database.

A Builder connects the specifications for the user interface to the information in the domain model. The Builder is embodied in an application model, which serves as the coordinator between the user interface and the domain model. The Builder creates the executable system, applying the look of any of several different window managers.

Read-and-Apply Tools

The tools for painting and defining utilize a powerful read-apply metaphor when operating on a user interface canvas. Each tool is linked to the structure of the canvas so that it can “read” the characteristics of the components,

which enables you to understand current attributes. You can then refine the properties and “apply” the refinement.

Visual Reuse

A key feature of VisualWorks is that it enables you to organize your applications so as to share with and inherit from one another. This sharing occurs at several levels—we use the phrase *visual reuse* to encompass the varieties of sharing, including:

- n Direct interface reuse—The same user interface can be connected to different domain models.
- n Interface nesting—A “larger” interface can incorporate selected parts of a “smaller” interface, or all of it.
- n Interface inheritance—You can arrange a hierarchy of application models that refine the user interfaces they inherit from their ancestors.
- n Direct application reuse—One application can invoke another, establishing interdependencies as needed.

This visual reuse augments the traditional benefits of reuse provided by an object-oriented language such as Smalltalk. Every object in the environment—including the tools and the application frameworks—is built up from a hierarchy of classes that has surprisingly simple roots. Because VisualWorks is an open environment, you can reuse any class of object in your application. You can think of VisualWorks as a generic family of objects whose evolution you shape to fit your needs.

Starting VisualWorks

VisualWorks runs on a variety of computer systems, under several different window managers. Starting VisualWorks requires a slightly different procedure in each windowing system, as detailed in your *VisualWorks Installation Guide*.

In general, if your system provides a special mechanism for launching applications, such as double-clicking on an application’s icon, use that method for VisualWorks. If you normally launch an application by entering a command string, enter a string of the following form:

```
oe-path image-path
```

For **oe-path**, substitute the pathname of the object engine. For **image-path**, substitute the pathname of the standard system image (**visual.im**) or of a custom image.

Saving Your Image

From using other applications, you may be accustomed to saving a *file* that you have created. As a programming environment that permits you to modify virtually any aspect of it, however, VisualWorks lets you save the entire working environment. You accomplish this in a simple step known as *saving an image* (also called *making a snapshot*). To do so, choose **File?Save As...** in the VisualWorks main window. The current image name is provided as a default.

If you would like to save the image with another name, enter a new name to create a new image and leave the existing one as is. (Each image file requires multiple megabytes of disk space, depending on how much code you have added to the system—so make sure your disk can accommodate a new image.)

The system appends the extension **.im** to the image file, except on a Macintosh platform, where no extension is attached. Your image, or snapshot, will contain not only code modifications you have made, but also the current state of every window you have opened.

Exiting VisualWorks

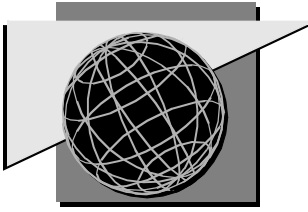
To exit from VisualWorks, choose **File?Exit VisualWorks...** in the VisualWorks main window. A dialog will be displayed, offering the following options:

Exit

Save then Exit

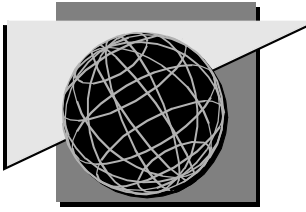
Cancel

For best results, always exit from VisualWorks by using the VisualWorks main window's **File?Exit VisualWorks...** command. If you cannot exit via the **File?Exit VisualWorks...** command, refer to “Emergency Exit (all platforms)” on page 219.



Part I

Smalltalk Language



Chapter 2

Object Orientation

Much of the literature on object-oriented programming (OOP) tends to emphasize how it differs from procedural programming. And it is different, in many important respects. Working with objects requires some new ways of thinking, just as touch typing requires that you unlearn hunt-and-peck habits.

Unfortunately, too often the strangeness of it all is overemphasized. This chapter attempts to present object-oriented terms and concepts in a familiar context, using your programming expertise as a bridge to the new world of objects.

Procedures vs. Objects

In a conventional programming language, a procedure typically performs multiple operations and handles several items of data. For example, when a user inputs a customer record in an accounts receivable system and then executes a 'save' command, a procedure might be invoked to validate the dozen or more fields of information in a customer record.

What happens when the five-digit field for a postal code in an application has to be changed to accommodate the six-character Canadian format? Three sources of inefficiency become apparent immediately.

First, what amounts to a single conceptual change (modify postal code) has to be programmed in two locations (database structure and procedure code, as shown in part A of the illustration). Wouldn't it be nice if the data were somehow bound more tightly to the code, so that only one system element had to be changed?

Second, there are likely to be multiple procedures that handle postal codes—besides customer data maintenance, there may be supplier maintenance, distributor maintenance, and so on (part B). In each such procedure, the postal code validation routine has to be modified. In an ideal system, such a change would affect all pertinent procedures simultaneously.

Third, although only the portion of a procedure's code pertaining to postal codes is affected by the change, the entire procedure has to be scanned by the programmer and recompiled (part C).

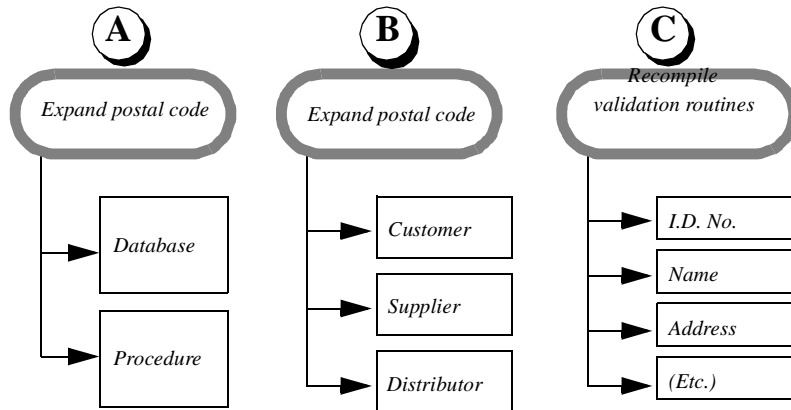


Figure 2-1 Modifying zip code in procedural programs

Objects and Methods

There has to be a way to isolate the changes more intelligently. In an ideal programming language, each field in the database would be a separate entity for the purpose of changing its attributes. Each atomic routine in a program would be a separate entity for the purpose of maintaining the code. So now we have a set of atomic data elements and a set of atomic procedures. It turns out that the procedures cluster very naturally around the data. The procedure for validating a postal code is something that only the postal code object needs to know. Likewise, only the address object needs to know what its valid inputs are. So if we can make each data object smart enough to perform the useful operations on itself, we no longer need separate procedures at all.

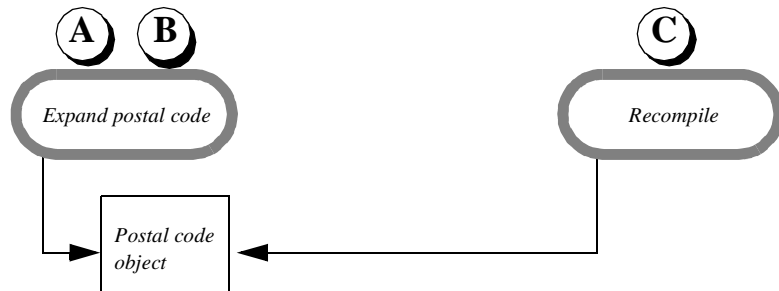


Figure 2-2 *Modifying postal code in Smalltalk*

The simple strategy of making data smart is at the core of Smalltalk. An application is no longer a collection of procedures that act on a database, but a collection of data objects that interact with one another via built-in routines called *methods*. The language is *object-oriented* rather than *procedure-oriented*.

In fact, because Smalltalk variables are not bound to specific data types, no change is required for client programs to be able to store a string rather than an integer in a postal code.

To expand the definition of a postal code in Smalltalk, all you need to do is broaden the postal code object's validation routine. When another object, such as the customer or supplier object, needs to know whether a postal code is valid, it passes the proposed value to a postal code object, which uses its built-in mechanisms to do the testing.

Composite Objects

Some objects, called *composite* objects, contain several other objects. For example, a customer object would contain identifying objects such as customer number, name, address, city, state, postal code and telephone number. Why have a customer object at all? Because some procedures have to be performed for a customer rather than a postal code or a telephone number.

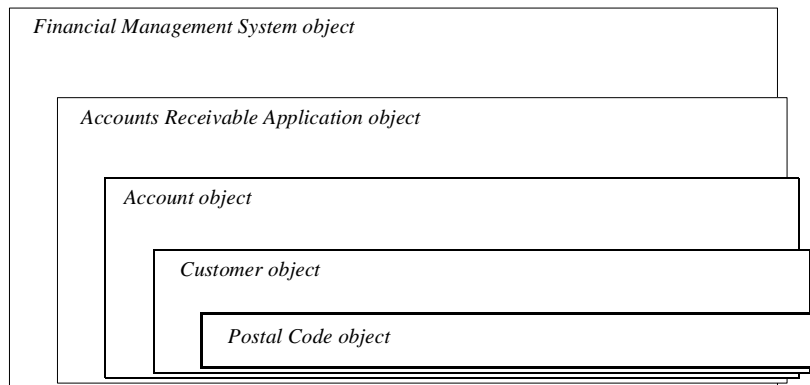


Figure 2-3 *Hierarchy of Objects*

The ‘create’ command, for example, is best centralized up at the customer level of abstraction, because it is an operation that affects all of the data objects that make up a customer. What does that ‘create’ operation consist of? In our example, the customer object simply fires off the same message to each member of its collection: ‘Here’s your input—validate it and store it. Let me know if there’s a problem.’

Theoretically, the customer object would provide the customer-identification part of an ‘account’ object that handles requests related to a customer’s account status. A collection of account objects would make up the accounts-receivable system, itself an object that knows how to answer questions about its collection of accounts. And the accounts-receivable object joins an accounts-payable application and a general-ledger application as parts of a financial-management package. Hence, programming an application in Smalltalk consists of building a hierarchy of objects. Another way of looking at it is that you’re creating a single object (the application) that contains component objects, each of which may contain smaller components, and so on. Figure 2-3 illustrates a portion of such a hierarchy.

Variables and Methods

An object typically is made up of one or more private variables (the data) combined with a set of methods for manipulating that data. Each method is a specialized subroutine.

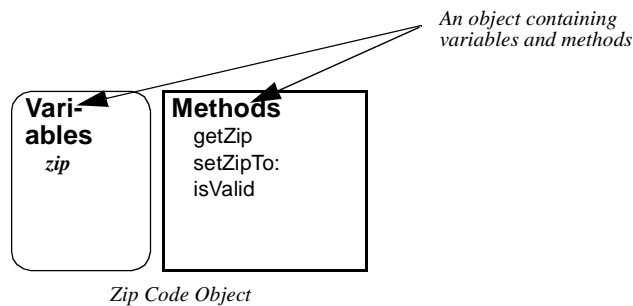


Figure 2-4 Variables and methods of an object

The two parts of an object are also known as *state* and *behavior*. The values held by an object's variables define its state. Its methods—what it knows how to do—define behavior.

For example, a postal code object might have a variable called *zip* to hold the postal code string. It needs at least two methods to be a civilized object, as listed in Table 2-1.

Table 2-1 Accessing Methods for the Postal Code Object

Method name	Description
<i>getZip</i>	Return a string containing the postal code
<i>setZipTo:</i>	Replace the contents of the zip code variable with the string that follows the colon

As you can see, each variable typically generates two accessing methods, one for inquiry and one for update. Even a simple postal code object will often have other methods. For example, it might have a method called *isValid*, which checks to make sure the string conforms to a recognized postal code format.

Method Grouping

The method name is used by other objects to select that operation. Thus, a method name is known as a method *selector*. A method selector is sometimes also called a *message*, though technically a message consists of the selector plus any arguments. In this manual set, we frequently use the method name

as an adjective, as in “a `getZip` message,” which is shorthand for “a message involving the `getZip` method selector.” The fundamental unit of any Smalltalk expression is an object reference followed by a message, as in `postalCode getZip`. This expression asks the `postalCode` object to return the value stored in its zip code variable.

Method names may contain letters, numbers, and underscores, but may not begin with a number. When two or more words are combined to form a name, as in this case, second and later initials are capitalized to improve readability—this convention applies to all names in the system: objects, variables and methods. For global variables, the first letter is also capitalized. All method names begin with a lower-case letter.

It is not uncommon for an object to have dozens of methods. From class to class, methods tend to cluster in recurring groups—for example, objects that have data also have a set of methods for accessing the data. Collectively, such methods are known as *accessing* methods. You may encounter the phrase “accessing protocol,” which refers to the set of methods for accessing data within an object.

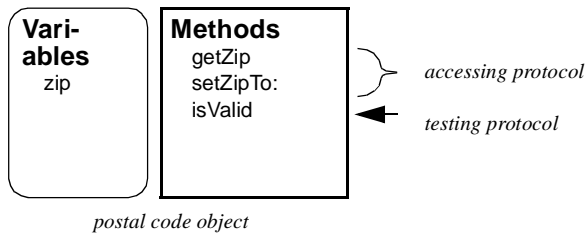


Figure 2-5 Two message categories in a postal code object

A message category, also called a *protocol*, is a convenient grouping of related methods, much as a file folder holds related documents. The Smalltalk programming community follows informal standards in choosing protocol names—Appendix A *Message categories reference* lists the more common protocol names and describes their usage. The System Browser uses protocol names to help you search the code library.

Classes and Instances

The question arises: How can there possibly be only one postal code object that serves both a customer and a supplier when the real-world customer and

supplier might reside in different zip zones? For that matter, each new customer might have a different postal code.

Obviously, there is a separate postal code object in each instance because the values stored in the variables are different. On the other hand, it would be silly to duplicate the postal code object's methods for each instance, so there must be one postal code object that is unique in that it knows how a postal code ought to behave. The data-only object is known as an *instance*; the method-holding object is called a *class*.

Class names may contain letters, numbers, and underscores, but may not begin with a number. The first letter of a class name is capitalized to distinguish it from an instance name. So `ZipCode` is a class; `zipCode`, `aZipCode` and `customerZip` are all instances.

A class can be thought of as the object behavior affixed to a data template. An instance is created by cloning the template so a new set of variables can be stored. The `ZipCode` class has a template specifying that each instance of `ZipCode` will have one variable named `zip`. Any given instance of that class consists of a value for that variable.

Class Variables

A class can also have its own variables, which serve as system constants. For example, the built-in class `Date` has a class variable called `MonthNames`, which stores an array containing names for the 12 months. Our `ZipCode` class might have a class variable called `Formats`, to store a collection of known formats. In either of these examples, it would be wasteful to store a new copy of the class variable in every instance that is cloned from it because the value of the variable remains constant for all instances.

Like class names, class variable names begin with a capital letter. The class variables are not part of the template used to create an instance—only instance variables belong to the template.

Class Methods vs. Instance Methods

If an instance doesn't have its own copy of the methods on board, how can it respond to messages? In a manner that is transparent to the programmer, the system looks for the appropriate method in the class from which the instance was spawned.

The expression `zipCode.getZip` is equivalent to “ask the `ZipCode` class to execute its instance method called `getZip` using the variables in the instance called `zipCode`.” Thus, though each instance does not use up unnecessary

memory space by creating a copy of the instance methods, the effect is the same.

A message can also be sent to a class, which is also an object. Each class has two different sets of methods, one for itself and one for its instances. When a class receives a message directly, it looks for the corresponding method among its class methods.

Thus, the expression `zipCode getZip` executes an instance method that returns the value of the instance variable. On the other hand, the expression `ZipCode formats` causes a class method to be performed and the value of a class variable (i.e., a constant) to be returned.

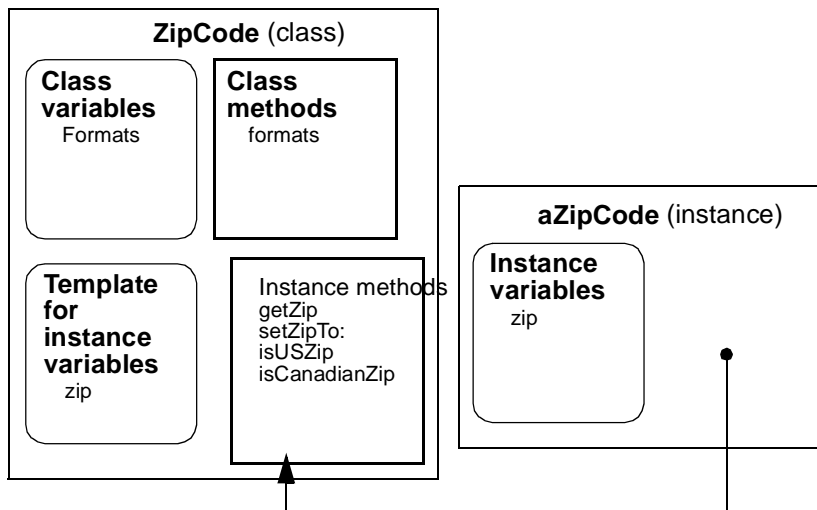


Figure 2-6 The parts of a class and an instance, and their interconnections

To summarize, the Smalltalk language consists of thousands of subroutines called methods that are organized as a library of class objects. The typical class object consists of class variables, class methods, instance methods, and a template for instance variables.

Class Grouping

VisualWorks contains a library of class objects, more than 1000 of them. As you might expect, they have been herded into categories to help you find the one you need for any given purpose. For example, numeric classes such as

`Integer` and `Fraction` belong to a category called `Magnitude-Numbers`. Classes `Character` and `String` are members of the category called `Collections-Text`. Category names typically include a subcategory name—there are two categories of magnitude-like classes: `Magnitude-General` and `Magnitude-Numbers`. Collection-like objects are grouped into eight categories, all of which begin with the word “Collections.”

Category names have no impact on your program’s functionality. The `Integer` class could be moved to the `Collections-Text` category or to an entirely new category without affecting any program. Try it, if you’ve mastered the `System Browser`, which uses category names to help you navigate the class library. In the sense that they are group labels, categories are to classes as protocols are to methods.

Programming in Smalltalk amounts to building new class objects and reusing the existing ones. You can examine, use and even modify any class in the system though, in practice, there are many that you will never need to think about because they provide low-level support for other classes. (Changing system classes except by extension is not recommended—it’s easy to introduce serious system bugs by doing so.)

Class Inheritance

The class library is organized in a hierarchy of specialization, very much like the taxonomy applied to the animal kingdom. At the root of the tree is class `Object`. One kind of `Object` is a class called `Magnitude`. If you dig down through a few more levels of specialization within the `Magnitude` subhierarchy, you come to a class called `SmallInteger`. An instance of class `SmallInteger` is an integer such as 3.

If you execute the expression `3 raisedTo: 4`, the correct result (81) will be returned. A `raisedTo:` message with an argument of 4 is being sent to 3, which is an instance of `SmallInteger`. From the prior discussion about

instance methods, one would assume that the class `SmallInteger` has an instance method called `raisedTo:`, but that is not the case.

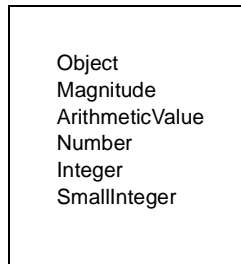


Figure 2-7 Inheritance hierarchy for the `SmallInteger` class

Looking up a Method

Smalltalk provides a method-lookup mechanism that starts its search for a given method in the obvious place—the class of the object to which the message was sent. If no such method exists there, the method finder climbs up through the hierarchy, stopping at each level to look for the method. In our example, the method finder has to go up two levels, past the `Integer` class to its parent, `Number`. There it finds the `raisedTo:` method.

`SmallInteger` is a subclass of `Number`, because it provides specialized variables and/or methods. `Number` is a superclass of `SmallInteger`, as is the class that sits between them in the hierarchy, `Integer`. Class `Object` is the top-level superclass of all other objects.

It's important to remember that the method finder has two ladders at its disposal, one for finding class methods and the other for locating instance methods. As it climbs upward through the superclasses, it uses only one ladder or the other, but not both. Its choice of ladder is determined by the message recipient. If the message is sent to an instance (3, in our example), only instance methods are searched. A message sent to a class such as `SmallInteger` would push the method finder onto the class-method ladder. The

expression `SmallInteger raisedTo: 4` would cause a fruitless search resulting in an error.

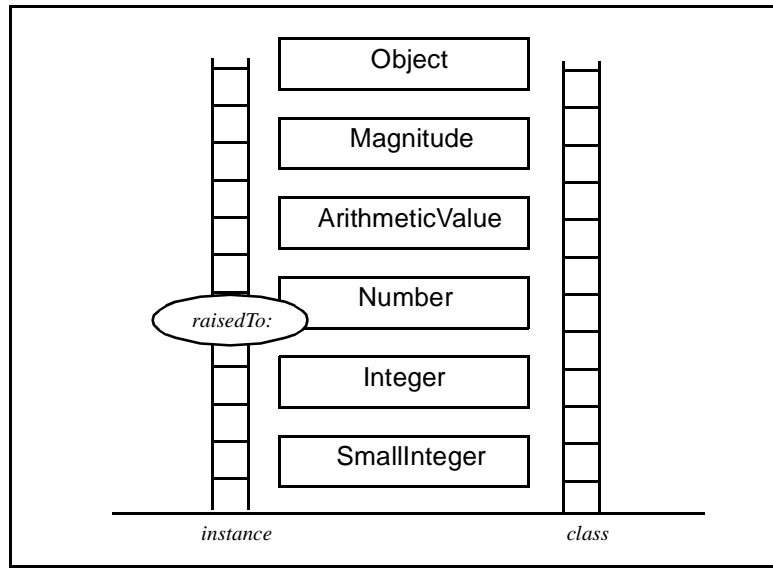


Figure 2-8 The upward search path of the object hierarchy

Overriding an Inherited Method

An instance of any subclass of `Number` can respond to a `raisedTo:` message, but that doesn't mean they all use `Number`'s version of it. The subclass `Float`, for floating point numbers such as 3847.029, has its own instance method called `raisedTo:` because floating-point numbers require a specialized algorithm for exponentiation. When the method finder goes to work on the expression `3847.029 raisedTo: 4`, it stops at class `Float` and never gets as high as `Number`.

Inheritance also applies to variables. Thus, each class inherits all of the methods and variables of its superclasses.

For example, the `ApplicationModel` class provides variables and methods that support a mechanism for notifying dependent objects of a change in state. This mechanism is inherited by all subclasses of `ApplicationModel`. The `Customer` class that we mentioned earlier might well be created as a subclass of `ApplicationModel`. Then, if we create a `View` that displays the values in the `Customer` object, the `Customer` inherits methods for keeping that `View`

in sync with the data changes. We don't have to write any code for such dependency coordination.

Abstract Classes

The class `Object` is the ultimate superclass of all other classes, whether built into the system or newly created by an application developer. But `Object` has an empty template for instance variables. This may seem odd considering that instance variables hold the actual data. What would an instance of class `Object` hold as its nugget of data? The answer is that `Object` is not intended to have instances. Its behavior is inherited and used by its subclasses and their instances.

When a class is not intended to be used to create concrete instances, it is called an *abstract* class. An abstract class is frequently useful as a repository for variables and methods that are useful to two or more classes, none of which is a logical subclass of the other. Another way of looking at it is that the similarities shared by a group of objects are squeezed up from their separate locations and into a common superclass.

The postal code can serve as an example once again. Until now, we have been trying to make a single `ZipCode` class handle two very different postal code formats. Presumably, as the customer base expands, more methods would have to be added to handle other postal systems. Eventually, a plain old United States numeric zip code would have to be stored in a class that had more irrelevant methods than relevant ones—and that's the sort of awkwardness this object-oriented technology is supposed to avoid.

Let's make `ZipCode` an abstract superclass, with two new subclasses: `USZip` and `CanadianZip`. They can both inherit the `zip` variable and the accessing methods (`getZip` and `setZipTo:`) as well as any class variables and class methods. The `isValid` method must be re-implemented in each of the subclasses, to handle their specific formats. The `ZipCode` class's version of `isValid` can then hand off the validation request to the appropriate subclass. To `Customer`, `Supplier` and any other objects that interact with `ZipCode`, the mechanism for finding out whether a zip code is valid has not changed.

Nesting Abstract Classes

A subclass of an abstract class can be abstract itself. One might make `USZip` abstract, for example, and create one subclass representing the five-digit format (`OldUSZip`) and another for the hyphenated-nine-digit format (`SlowToBeAdoptedUSZip`).

Choosing a Superclass

When you create a new class, choosing its superclass is an important design decision. The choice is made easier when you employ an architecture that has been proven in many diverse applications.

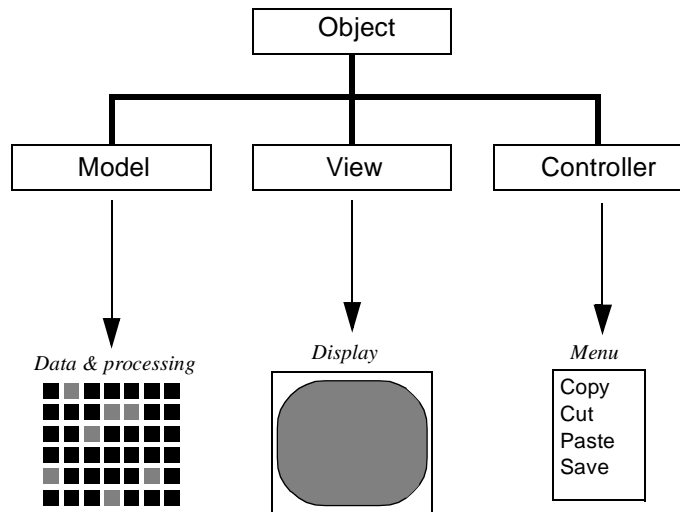


Figure 2-9 The containment hierarchy of the class library

The key to this architecture is to divide your application into two parts. First develop the data structure and the attendant processing, then invent the user interface. The user interface is further subdivided into input and output modules. The data-and-processing module is referred to as the *model*. The output module usually consists of the screen displaying mechanisms—it’s called the *view*. The input module is called the *controller* because it enables the user to control the sequence of events by entering data and commands.

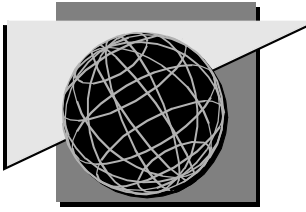
Not surprisingly, Smalltalk provides an abstract class as the intended starting point for each of these three modules: **Model**, **View** and **Controller**. Thus, the architecture is known as model-view-controller, or MVC, programming. For detailed information about MVC design, see “Application Framework” on page 223.

We use the term “application” broadly here—an object as lowly as a postal code can be regarded as a self-contained model that can have an associated

view (a box on the screen in which the postal code is displayed) and controller (for accepting keyboard input to the model in the form of data entry). This implies that an MVC application can be a component of a larger MVC application, and so on. That is indeed the case, furthering the cause of reusability by segmenting any given program into easily separated components. In this sense, a model-view-controller triad is the fundamental unit of design just as an object is the fundamental unit of implementation.

When you choose a superclass for a new class, you are selecting an inheritance hierarchy—positioning the method finder’s ladder in the class library, so to speak. **Model**, **View** and **Controller** head three major subhierarchies within the library. Your choice of superclass typically resolves to a class within one of those subhierarchies, and often to the head classes themselves.

Many of the user-interface components that have been layered on top of Smalltalk to form VisualWorks are subclassed from **Model**, **View** or **Controller**. The remaining classes are typically subclassed from **Object**, because as linguistic elements they stand apart from the MVC machinery.



Chapter 3

Syntax

ParcPlace Smalltalk employs syntactic rules that are unique in some respects, as well as unusual naming conventions. We begin with a discussion of naming style, then proceed to explore the syntax of literal constants, variables, message expressions and block expressions.

Naming Conventions

Names of classes, variables, methods and other expression elements tend to be lengthy in Smalltalk by comparison with most languages. For descriptive purposes, a name is frequently made up of two or more words. When this is the case, convention dictates that the first letter of each *embedded* word is capitalized. This convention is not enforced by the language or by any of the development tools provided with ParcPlace Smalltalk, but it does improve readability.

In conformance with the draft ANSI standard, VisualWorks does not allow the use of periods in class names or other identifiers.

Capitalization Rules and Conventions

Table 3-1 provides rules and conventions that apply to the first letter of a name.

Table 3-1 Capitalization Rules and Conventions

Type of name	Initial capital	Example
Class category	Yes (suggested)	Magnitude-General
Class	Yes (mandatory)	Date
Class variable	Yes (mandatory)	MonthNames
Global variable	Yes (mandatory)	MaximumUsers

Table 3-1 Capitalization Rules and Conventions

Type of name	Initial capital	Example
Pool variable	No (suggested)	cr
Instance variable	No (suggested)	year
Temporary variable	No (suggested)	aDate
Method protocol	No (suggested)	accessing
Method	No (suggested)	monthName

Literal Constants

Smalltalk provides six types of literal constants:

- n number
- n character
- n string
- n symbol
- n byte array
- n array of literals (including other arrays)

In addition, three special literals are recognized by the compiler: `nil`, `true` and `false`. These are discussed following the six types listed above.

Numbers

Numbers are represented in the usual way, using a preceding minus sign and embedded decimal point as required.

Integers

Integers are expressed as numeric literals such as `101`, or as the result of arithmetic operations involving one or more integers such as `55 + 46`.

Floating Point Numbers

Floating point numbers must have at least one digit to the left of the decimal point, so the compiler can distinguish a decimal point from a period used as an expression delimiter. Thus, `0.005` is legal, but `.005` is not. In scientific notation, the *e* is replaced by a *d* in a `Double` and a *q* for quad-precision.

Fixed-Point Numbers

A fixed-point number is useful for business applications in which a fixed number of decimal places is required. Fixed-point numbers are expressed by placing the letter *s* after a literal integer or a floating-point number. The number of decimal places preceding the *s* implicitly specifies *scale* of the number (the number of decimal places to be preserved). Note that an explicit scale takes precedence over an implicit one, so that 99.95s4 is the same as 99.9500s, while 99.9500s2 is an error.

Nondecimal Numbers

Number literals can also be expressed in a nondecimal base by prefixing the number with the base and the letter *r* (for *radix*). For example:

Octal	Decimal
8r377	255
8r34.1	28.125
8r-37	-31

When the base is greater than ten, the capital letters starting with “A” are used for digits greater than nine. For example, the hexadecimal equivalent of the decimal number 255 is 16rFF.

Numbers in Scientific Notation

Numbers can also be expressed in scientific notation by including a suffix composed of *e* (for *exponent*) or *d* (for *double-precision*) plus the exponent in decimal. Note that you can also use the letter *q* instead of *d*. The letter *q* stands for quad-precision, and is available for portability to other Smalltalk systems; however, in VisualWorks, *q* has the same effect as *d*.

The base is raised to the power specified by the exponent and then multiplied by the number. For example:

Scientific Notation	Decimal
1.586d5	158600.0

Scientific Notation	Decimal
1586e-3	0.001586
8r3e2	192
2r11e6	192

Characters

A character literal is always prefixed by a dollar sign. For example:

```
$a  
$M  
$-  
$$  
$1
```

Strings

A string literal is enclosed in single quotes (double quotes are used to delimit a comment). Any character can be included in a literal string. If a single quote is to be included, it must be preceded by a single quote, as in:

```
'I won't fail'
```

Symbols

A symbol is a label that conveys the name of a unique object such as a class name. There is only one instance of each symbol in the system. A symbol literal is preceded by a number sign, and optionally enclosed in single quotes. For example, `#Float` and `#'5%'` are legal symbols. If a symbol is enclosed in an array, it must still be preceded by a number sign.

Byte Arrays

A literal byte array is enclosed in square brackets and preceded by a number sign. Elements of the array must be integers between 0 and 255. They are

separated by one or more spaces. The result, as in the following example, is an instance of class `ByteArray`:

```
#[255 0 0 7]
```

Arrays

An array literal is enclosed in parentheses and preceded by a number sign. Elements of the array are separated by one or more spaces (extra spaces are ignored). An array literal embedded in another array must still be preceded by a number sign. The following example contains a number, a character, a string, a symbol and another array (of three characters):

```
 #(1586.01 $a 'sales tax' #January #($x $y $z))
```

***Note:** The mutability of arrays and strings is a source of possible error in using literals. When you change an element in a nonatomic literal constant (a String, an Array, or a ByteArray), the change is reflected globally. For that reason, experienced Smalltalk programmers rarely pass a mutable literal constant from one method to another, but pass a copy instead.*

Booleans

The boolean constant `true` is the sole instance of class `True`, which is a subclass of `Boolean`.

The constant `false` is the sole instance of class `False`, also a subclass of `Boolean`. Unlike most instances, the values of `true` and `false` are hard-wired in the compiler—which qualifies them as constants.

Their behavior, however, is defined in the instance methods of the classes `True` and `False`. They implement logical operations such as `not`.

The `nil` object is the sole instance of class `UndefinedObject`. As the class name implies, `nil` is the null value given to variable slots that have not yet been assigned a more interesting value. Like the booleans, `nil` is hard-wired in the compiler. Its behavior is defined in `UndefinedObject`—for example, it overrides the `isNil` method implemented by `Object` (answering `true` instead of `false`).

Variables

Six kinds of variable are available in Smalltalk. Listed in order of increasing scope, they are as follows:

- n temporary
- n instance
- n class instance
- n class
- n pool
- n global

Temporary and instance variables are *private* variables because their scope is local to a method (for temporaries) or to an instance (for instance variables). Class, class instance, pool and global variables have wider clienteles, as described below. In addition, there are three special variables, which are discussed after the section on global variables.

Variable names may contain letters, numbers, and underscores, and may not begin with a number. By convention, the first letter is lowercase for local variables and uppercase for nonlocal variables.

The default value for any variable is the nil object. To assign a new value to a variable, use a colon followed by an equal sign (:=, pronounced “gets”), as in the expression:

```
prompt := 'Enter your name'
```

The expression on the right-hand side of the assignment can be any legal Smalltalk expression. The following examples are all valid assignment expressions. They have the effect of creating an array of ice cream flavors and selecting one of those flavors at random:

```
flavors := #('chocolate' 'vanilla' 'mint chip').  
index := (Random new next)* 3.  
flavorChoice := flavors at: index truncated + 1
```

Assignments can be chained when two or more variables are to store the same value, as in:

```
majorLoopCounter := minorLoopCounter := 1
```

Chained assignments should only be used with literal or read-only values—otherwise, updating one variable has the side effect of changing the value of the other variable similarly.

Temporary Variables

A temporary variable is most often encountered in a method, where it provides temporary storage for an argument or a calculated value. Its lifetime begins when it is declared by the method, or a block expression within the method, and ends when the block or method finishes processing and returns control to the calling object.

For example, the class `Time` provides an instance method called `hours:minutes:seconds:`. This method declares three temporary variables to hold its arguments, and names them `hourInteger`, `minInteger` and `secInteger`. The first line of the method consists of the method name with these argument names inserted, as follows:

```
hours: hourInteger minutes: minInteger seconds: secInteger
```

When a client object sends this message to an instance of `Time`, appropriate integers are provided. Here is what the message expression might look like:

```
aTime hours: 11 minutes: 42 seconds: 15
```

The result of this expression is that, for as long as the method continues processing, `hourInteger` is equal to 11, `minInteger` is 42, and `secInteger` is 15. Argument variables, unlike other temporaries, do not accept new values by assignment. As a documentation convention, a temporary variable is usually named so as to indicate the object type it is intended to hold. However, any object can be stored in any variable.

A temporary variable can be used for dynamic storage as well as argument storage. For example, the `Dictionary` class provides an instance method called `occurrencesOf:`, for counting the number of entries in a dictionary that equal the argument. The method declares a temporary variable in which

to store the total. A temporary variable is declared by enclosing its name between vertical bars. The declaration must follow the message definition, and usually follows a comment explaining the method. The first three lines of the `occurrencesOf:` method look like this:

```
occurrencesOf: anObject
    "Answer how many of the receiver's elements are equal to
    anObject."
    | count |
```

One or more white-space characters (space, tab, etc.) are used to separate variable names when multiple temporaries are declared between the vertical bars.

Instance Variables

An instance variable is used to store data in an instance. It is declared as part of the class definition. The following definition of the `Set` class shows the form of an instance variable declaration (on the second line):

```
Collection variableSubclass: #Set
    instanceVariableNames: 'tally '
    classVariableNames: "
    poolDictionaries: "
    category: 'Collections-Unordered'
```

When an instance of class `Set` is created, the `tally` variable is initialized to `nil`. Each time an element is added to or removed from the set, its `tally` is updated with a new count of the set's elements.

Thus, an instance variable provides a place to store a value that can be used by multiple methods (whereas a temporary variable is local to a specific method). Its lifetime is that of the instance itself.

Instance variables are inherited, so an instance has its own copy of the instance variables declared by all of its superclasses. For example, the class `SystemDictionary` is a subclass of `Set`, so it does not need to declare its own `tally` variable because it can use the `tally` that is declared in its superclass.

Class Instance Variables

A class instance variable is used to store data that varies with each subclass in a hierarchy. It can only be accessed by a class method.

For example, suppose you have an abstract `LanguageDictionary` class that has methods for looking up words to verify spelling, etc. You give `LanguageDictionary` a class instance variable named `wordCollection`. Now you create a series of subclasses corresponding to the English language, the Polish language, and so on. The `EnglishLanguage` class can initialize `wordCollection` to hold English words. The other subclasses can initialize it differently. Then when an instance of any subclass asks for `wordCollection`, it gets the appropriate language-specific version.

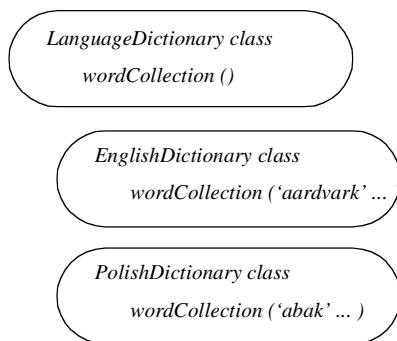


Figure 3-1 Class instance variable

The advantages of this approach are that you still only have to initialize the `wordCollection` once for each subclass (unlike instance variables) and all subclasses can reuse methods that employ a common variable name (unlike class variables).

Class Variables

A class variable is used to store data that is useful to all instances of the host class and its subclasses. Because it is a shared variable (accessible by multiple objects), the first letter of a class variable name is capitalized. Its lifetime is that of its host class.

The initial value of a class variable is usually assigned in a class method (normally named `initialize`) and that method is typically invoked as the final act of creating the class.

For example, class `Date` has five class variables, which are declared in the class definition (third and fourth lines) as follows:

```
Magnitude subclass: #Date
  instanceVariableNames: 'day year '
  classVariableNames: 'DaysInMonth FirstDayOfMonth
  MonthNames SecondsInDay WeekDayNames'
poolDictionaries: ''
category: 'Magnitude-General'
```

The instance variables change with each instance, so they can only be accessed directly by the same object. The class variables, however, keep the same values across instances. So when an instance wants to access the array of integers contained in the `DaysInMonth` variable, for example, it does not have to send a message to `Date`. It can use the variable in its methods just as naturally as it would use an instance variable. Objects that are not in the inheritance chain would have to query `Date` for the information.

Pool Variables

A pool is a dictionary of global variables that is intended for use by a specific set of classes. Its purpose is to provide quick access to the contents of that dictionary (short-cutting the usual dictionary-lookup machinery). Each element in the dictionary is a pool variable, and is available to any class that declares the pool in its definition. For example, class `Text` declares a pool dictionary in its class definition:

```
CharacterArray subclass: #Text
  instanceVariableNames: 'string runs '
  classVariableNames: ''
  poolDictionaries: 'TextConstants'
  category: 'Collections-Text'
```

The `TextConstants` dictionary provides keyboard mapping support for various text-manipulating objects. Each key in its dictionary names a textual element or characteristic such as `Tab` and `Underline`. Each key's associated value is the character sequence that invokes that property.

Note: *Pool dictionaries are not inherited, so you must add them to each class that is to use them, even if they are declared in its superclass.*

Global Variables

A global variable is accessible by any object. It must begin with a capital letter. Its lifetime is that of the system, unless it is explicitly removed from the system dictionary.

All class names are global variables, as obvious examples of objects that must be accessible to all other objects. (Removing a class name from the image by deleting its entry in the Smalltalk dictionary is not recommended.) Object-oriented programming style discourages the creation of globals other than class names. In fact, only a handful of globals other than class names and pool dictionaries exist in the system. **Smalltalk**, **Transcript** and **Processor** are examples. They could as well be implemented in the form of class variables, with class methods to return the values of those variables.

Special Variables

For three special variables, the value changes according to the execution context but cannot be changed by assignment: **self**, **super** and **thisContext**.

The most prevalent of these special variables is **self**, which holds a reference to the object that is executing the current message.

In the simplest case, **self** merely allows the programmer to direct a new message to the specific instance that is executing the current method. In effect, an object can execute another of its own methods. A hypothetical **doSomething** method could use a **computeX** method to calculate a number, for example, with the expression **self computeX**.

A more complicated case arises when inheritance is involved. Suppose the **doSomething** method is located in the superclass of the object that received the **doSomething** message. But **computeX** is implemented by the subclass.

How do we send the method finder back to the bottom of the ladder to search for `computeX`, rather than just starting from its superclass location?

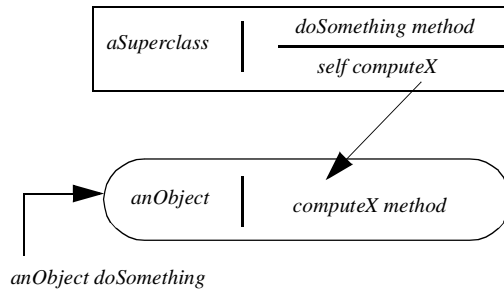


Figure 3-2 The special variable `self` is a pointer to the object (in this case, `anObject`) that received the message being executed (`doSomething`)

The surprising but pleasing answer is that the expression `self computeX` still works. The new message (`computeX`) is directed at `self`, which refers to the object that received the previous message (`doSomething`).

It's important to remember that `self` does not necessarily point to an instance of the class whose method is being executed. In our example, the word "self" is used in the parent's method but it refers to the child. Thus, using `self` in a method automatically provides for downward growth in the hierarchy.

The `super` variable is very similar to `self`, except `super` tells the method finder to begin its search one level above the executing method in the class hierarchy. This is useful when a subclass wants to add operations to its parent's method without having to duplicate the parent's code. Note that

`super` is in the nature of a qualifier applied to the method finder, so it cannot be assigned to a variable (as `self` can).

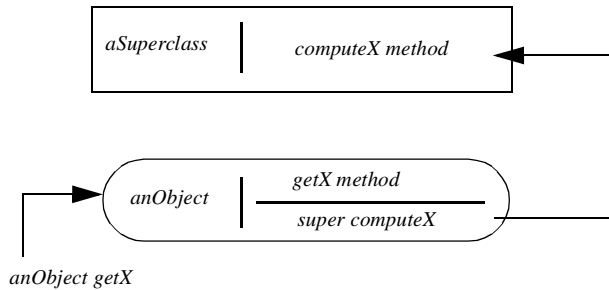


Figure 3-3 Special variable `super`

The third special variable, `thisContext`, is a reference to the stack context of the current process. While `self` and `super` are commonly used by Smalltalk programmers, `thisContext` is rarely needed by application developers. It is used by the system's exception handler and debugger.

Note: In some of the literature on Smalltalk, `self` and `super` are referred to as *pseudovariables*. However, other objects have also been called *pseudovariables*, so the term is ambiguous—we call them *special variables* instead.

Undeclared Variables

When a variable is deleted while references to it still exist, its name is automatically entered in a dictionary of **Undeclared** variables. This dictionary is maintained by the system and need not concern you under normal circumstances—but it can provide useful clues to certain kinds of program errors.

Message Expressions

A message expression is the fundamental unit of programming in Smalltalk. It has three kinds of components: a receiver, a method name and zero or more arguments. In `9 raisedTo: 2`, the receiver is `9`, the method name is `raisedTo:`, and the argument is `2`. The term message technically refers to the method selector and arguments, while a message expression includes the receiver.

Every message returns an object to the message sender. In the example just given, the `raisedTo:` method returns an instance of `SmallInteger`—

specifically, 81. There are three ways to denote the object to be returned from a method:

- n By default, the message receiver (**self**) is returned to the sender.
- n A return operator (^, entered as <Shift-6> on most keyboards) preceding a variable name causes that object to be returned. For example, the expression `^anObject` causes `anObject` to be returned.
- n A return operator preceding a message expression returns the value of that expression. For example, the expression `^3 + 4` causes the object 7 to be returned.

Note: In prior versions of *ParcPlace Smalltalk*, an up-arrow symbol was displayed as the return operator, though it was typed with the same <Shift-6> key used currently.

A period is used to separate message expressions. No period is necessary after the final expression in a series.

There are three types of message: unary, binary and keyword expressions. In addition, two or more messages can be joined in sequence. Each of these constructs is described below.

Unary Messages

A unary expression has a receiver and a method name but no argument. The following are all unary expressions:

1.0 sin.	"Returns the sine of 1.0."
Random new.	"Returns a random number generator."
Date today.	"Returns today's date."

Binary Messages

binary expression uses a special character such as a plus sign (+) as its method name and requires one argument. Some binary selectors are combinations of two special characters, such as the comparison selector `>=` (greater than or equal to). If you create a new binary, the second character of its name cannot be a minus sign (-).

The most common binary messages have to do with arithmetic operations, comparisons and string concatenation. Table 3-2 describes all of the valid binary selectors. One or more white-space characters before and after the selector are optional.

Table 3-2 *Binary Method Selectors*

Selector	Example	Description
+	counter + 1	Add
-	100 - 50	Subtract
*	index * 3	Multiply
/	1 / 4	Divide
**	4 ** 3	Raised to
//	13 // -2	Integer divide (round the quotient to the next lower integer; in the example, -7). An instance of <code>Point</code> can also be rounded via this operator.
\%	13 \% -2	Modulo (return the remainder after division; in the example, -1).
<	counter < 10	Less than
<=	index <= 10	Less than or equal
>	clients > 5000	Greater than
>=	files >= 2000	Greater than or equal
=	counter = 5	Values are equal
~=	length ~= 5	Values are not equal
==	x == y	Same object (receiver and argument are the same object or point to the same object)
~~	x ~~ y	Not the same object
&	(x>0) & (y>1)	Logical AND (return true if both receiver and argument are true, otherwise false).
	(x>0) (y<0)	Logical OR (return true if either receiver or argument is false).

Table 3-2 Binary Method Selectors

Selector	Example	Description
,	'abc','def'	Concatenate two collections.
@	200 @ 300	Return an instance of Point whose x coordinate is the receiver and whose y coordinate is the argument.
->	#Three -> 3	Return an instance of Association whose key is the receiver and whose value is the argument.

The assignment expression (:=) is not a method selector, so it is not listed here even though it looks like a binary selector. Also not listed is the linking symbol (>>) used in the debugger, which is also not defined as a selector. It provides a shorthand way of referring to a method and its implementing class together. Thus, `Set>>size` refers to the `Set` class's instance method called `size`.

Keyword Messages

A keyword expression has a receiver, one or more argument descriptors (keywords) and one argument for each keyword. Each keyword ends in a colon. The following are valid keyword expressions:

```
aDate addDays: 5 "Add five days to aDate."
```

```
anArray copyFrom: startIndex to: stopIndex
    "Return a copy of that portion of anArray
    that begins at startIndex and ends at
    stopIndex."
```

When there is more than one keyword, the method name is formed by concatenating the keywords. In the second example above, the method name is `copyFrom:to:` (formally pronounced “copyFrom colon to colon”). There is no limit on the number of keywords in a method name.

Messages in Sequence

Frequently, the receiver of a message is the object returned by the previous message expression. To avoid creating a temporary variable to store the

returned object, you can create a caravan of messages. For example, the first set of expressions below can be compressed into the form of the second set:

```
interest := principal * interestRate.  
principal := principal + interest.
```

```
principal := principal + (principal * interestRate).
```

This technique reduces the wordiness of the code, though sometimes at the expense of readability. Parentheses can be inserted, as shown in the example, to improve the readability and to assure that the intended parsing order is followed.

When two messages have the same parsing precedence, parentheses are sometimes required. For example, $3 + 4 * 5$ is very different from $3 + (4 * 5)$ because binary selectors are all evaluated from left to right.

Parentheses are also necessary when a keyword expression is in the argument expression for another keyword expression. For example, the first expression below is valid but in the second version the method selector is interpreted by the compiler as `readFrom:on:`, which does not exist.

```
Time readFrom: (ReadStream on: '10:00:00 pm').
```

```
Time readFrom: ReadStream on: '10:00:00 pm'.           "WRONG"
```

The following rules summarize the parsing order:

1. Parse parenthesized expressions before nonparenthesized expressions.
2. Parse multiple unary expressions left to right.
3. Parse multiple binary expressions left to right.
4. Parse unary expressions before binary expressions.
5. Parse binary expressions before keyword expressions.

The result of the following code fragment is that a number is printed in the System Transcript—can you trace the logic using the rules above?

```
| aSet nbr|
nbr := 207.
Transcript show: (aSet := Set new add: nbr + 3 * 5 sin) printString
```

In the first line, two temporary variables are declared. In the second line, one of the variables is assigned the number 207. In the third line, the following sequence of events takes place:

Event	Description
Set new	Create an instance of Set.
5 sin	Calculate the sine of 5 (-0.958924).
nbr + 3	Add 3 to nbr (210).
... *	Multiply 210 by -0.958924 (-201.374).
.. add: ...	Add -210.374 as an element in the set created in Step 1.
aSet :=	Assign the set to the variable aSet.
... printString	Convert the set to a printable string.
Transcript show:	Output the printable string to the System Transcript.

When two or more messages are to be sent to the same object, a semicolon can be used to *cascade* the messages. This avoids having to repeat the name of the receiver, though frequently at the expense of readability. For example, the first set of expressions below has the same effect as the final expression, in which the messages are cascaded:

```
Transcript show: 'This is line one.'.
Transcript cr.                                "Carriage return."
Transcript show: 'This is line two.'.
Transcript cr.
```

```
Transcript show: 'This is line one.'; cr; show: 'This is line two.'; cr
```

Block Expressions

A block expression represents a deferred sequence of operations. Blocks are used in control structures, so they will be discussed in more depth in Chapter 4, “Control Structures.” The syntactic characteristics of block expressions are discussed here.

A block expression is enclosed in square brackets, as in:

```
[index := index + 1.  
anArray at: index put: 0]
```

The messages inside the block are not sent until the block object receives the unary message `value`. The following expressions have the same effect:

```
index := index + 1.  
[index := index + 1] value.
```

Up to 255 separate arguments can be passed to a block. Argument names must be listed just inside the opening bracket. Each argument name must be preceded by a colon. The final argument name must be followed by a vertical bar. For example:

```
[:counter | counter := counter + 1]
```

The argument variables are private to the block. The values of the arguments are passed by using variants of the `value` message. There are four variants, to be used depending on the number of arguments:

```
value: anObject  
value: anObject value: anObject  
value: anObject value: anObject value: anObject  
valueWithArguments: anArray
```

Passing an argument to the example above would be arranged thus:

```
[:counter | counter := counter + 1] value: 3
```

Temporary variables can also be declared within a block. They must be enclosed in vertical bars and placed after the vertical bar that separates argument variables. They are local to the block.

The full syntax for a block is as follows:

```
[ :arg1 :arg2 |  
  |temp1 temp2 |  
  statement1.  
  statement2.  
  ...]
```

Formatting Conventions

The compiler ignores tabs, carriage returns and extra spaces. Formatting conventions vary but readability favors the following guidelines:

1. Start the message definition at the left margin and indent all other contents of the method one level.
2. Leave a blank line beneath the method comment and as a separator between sections of a long method.
3. Follow each period that ends an expression by a carriage return.
4. Indent as needed to visually identify each subordinate section of code.

The code browser provided with ParcPlace Smalltalk provides a `format` command for automatically applying these rules.

Syntactic Elements Summary

The following table summarizes the syntactic elements discussed in this chapter. The ellipsis (...) is used in examples when irrelevant elements are not shown.

Table 3-3 Syntactic Elements Summary

Element (punctuation)	Example
Character (\$)	\$a
String ('...')	'The address is', clientAddress
Symbol (#)	#Time

Table 3-3 Syntactic Elements Summary

Element (punctuation)	Example
ByteArray (#[])	#[255 76 0 49]
Array (#())	#('Three' #Three 3)
Comment ("... ")	"Multiply two numbers."
Nondecimal number	16r3F "Hexadecimal"
Scientific notation	1.586e5
Temp. variable declaration ()	index counter
Assignment (:=)	index := 0
Return (^)	^self
Parser grouping (())	3 + (4 * 5)
Block ([])	[index := index + 1]
Block argument (:...)	[:arg arg := arg + 1 ...]
Block variable (...)	[temp temp := 0 ...]
Unary message	'J.G. Kilhoon' size
Binary message	index < 10
Keyword message (:)	currentDate addDays: 3
Cascaded message (;)	Transcript; cr; cr; cr
Messages in sequence (.)	index := 0. counter := 1
Object-method pairing (>>)	Date>>addDays:

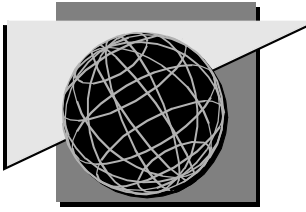
The following example illustrates a typical, fully assembled method. It is the Dictionary class's instance method called `includes`:

```
includesKey: key
    "Answer whether the receiver has a key equal to the
    argument, key."

    | index |
```

```
index := self findKeyOrNil: key.  
^(self basicAt: index) ~~ nil
```

Naturally, the ParcPlace Smalltalk class library contains thousands of other examples of methods.



Chapter 4

Control Structures

Control structures in Smalltalk are invoked by sending messages to various objects. The boolean objects `true` and `false` provide the if-then-else machinery, while numbers, collections and blocks provide the looping methods. These two types of control structure—branching and looping—are described in this chapter.

The `BlockClosure` class provides the machinery with which these control structures are implemented. You can use the same machinery to create new control structures. Block syntax is described in “Block Expressions” on page 41.

Branching

The boolean objects `true` and `false` implement methods for performing conditional selection (if statements). However, you will rarely see `true` or `false` mentioned explicitly in such an expression. Instead, an expression such as `index > 9` returns a boolean value, and that returned value is the receiver for the messages described below.

ifTrue:ifFalse:

The Smalltalk version of if-then-else is the `ifTrue:ifFalse:` method. It takes two blocks as its arguments, one to be executed if the receiver is `true` and the second to be executed if the receiver is `false`. In the following example, a prompt string is altered depending on whether the application user is a managerial employee:

```
(userType == #Manager)
  ifTrue: [prompt := 'Enter your password']
  ifFalse: [prompt := 'Access denied—sorry']
```

Either of the blocks can be left empty when no action is required. This is so often the case that `ifTrue:` and `ifFalse:` are provided as separate methods. In the example above, if no password were required, the `ifTrue:` portion of the expression could be dropped entirely. An `ifFalse:ifTrue:` method is also available, when the `false` condition is more prevalent.

Smalltalk has no equivalent of the *case* statement provided in many languages—a case statement tends not to be object-oriented.

Looping

Three types of iterative operation are available: conditional, number and collection looping. This section discusses the three types of looping.

Conditional Looping

Conditional looping involves a conditional test that determines whether to repeat the loop.

whileTrue: and whileFalse:

In the previous example, the expression (`userType == #Manager`) is evaluated just once. By contrast, the condition that drives a while loop has to be evaluated multiple times. In Smalltalk, it is enclosed in the square brackets that identify it as a block (an instance of class `BlockClosure`). The `whileTrue:` message causes that block to receive a `value` message, which triggers execution of the block's contents. If the expressions in the receiver block return a `true`, the argument block is executed. Then `value` is again sent to the receiver block to see if it is still `true`, repeating the cycle.

The following example might be used in a game that ends when there is only one player (the winner) left in the game:

```
[players > 1] whileTrue:  
  [nextPlayer takeTurn.  
   (nextPlayer outOfGame) ifTrue: [players := players - 1] ]
```

To reverse the logic of the test, use `whileFalse:`. For example, to process a stream of objects until the endpoint is encountered:

```
[self atEnd] whileFalse: [aBlock value: (self next) ]
```

For situations in which no argument block is needed, the unary messages `whileTrue` and `whileFalse` are available.

repeat

When a block of statements contains its own (reliable!) test for returning from the loop, the simple message `repeat` can be sent to the block.

Number Iteration

Number looping corresponds to the traditional *for* loop, and is implemented via messages to numbers.

timesRepeat:

To repeat a block of expressions a specific number of times, send a `timesRepeat:` message to a number and provide the repeatable block as an argument. For example, to send the string 'Testing!' to the Transcript `anInteger` times:

```
anInteger timesRepeat: [Transcript show: 'Testing!']
```

to:by:do:

A more elaborate sort of *for* loop comes in the form of the `to:by:do:` method, which lets you specify a starting integer, a stopping integer, the step increment and the block to be repeated. For example, to print something like a word processor's tab-setting ruler on the Transcript:

```
10 to: 65 by: 5 do: [ :marker |
  Transcript show: marker printString.
  Transcript show: '---'].
```

Here's a translation: Count by fives from 10 to 65. Pass each such value to the block, which converts it to a string and outputs it to the Transcript, followed by three hyphens. The output looks like this:

```
10---15---20---25---30---35---40---45---50---55---60---65---
```

Notice that, unlike `timesRepeat:`, the `to:by:do:` method automatically passes the value of the counter to the block (picked up by the argument named

marker in this case). The block must declare an argument variable to catch the passed value.

to:do:

When the counting increment is 1, you can use the simpler `to:do:`. The following example prints the ASCII equivalents of the numbers 65 through 122 in the Transcript.

```
65 to: 122 do: [ :asciiNbr |  
    Transcript show: asciiNbr asCharacter printString]
```

Collection Iteration

Collection looping supports scanning, counting and other operations involving one repetition for each member of a collection. It is frequently useful to repeat a series of operations for each element in a collection of objects (collections are discussed further in Chapter 6, “Collection Operations”). The integer iteration discussed above is a special case, dealing exclusively with numeric intervals—i.e., collections of integers. The iteration methods discussed in this section apply to other kinds of collections as well. All are implemented by the `Collection` class, which is the superclass of dictionaries, arrays, sets, strings, etc.

do:

The simplest method, `do:`, evaluates the block for each member of the collection. For example, to capture the contents of an array during program execution, we might want to convert each member to a printable string and output it to the Transcript:

```
anArray do: [ :anElement |  
    Transcript show: (anElement printString); cr ]
```

select:

To filter a collection and wind up with a desired subset, use `select:`. Each member of the collection that satisfies the conditions in the block is stored in a new collection of the same type, which is returned by the method. The following example counts the number of question marks in a string by gath-

ering the question marks into a new collection and then finding the size of that collection:

```
(aString select: [ :eachChar | eachChar == $? ] ) size
```

reject:

The **reject:** method is the opposite of **select:**. It gathers the members of the original collection that fail the test rather than those that pass it. Substituted for **select:** in the example above, it would create a collection of non-question-marks, which would then be sized.

detect:

The **detect:** method, like **select:**, tests each element of the collection. But instead of returning a subcollection of those elements that pass the test, it returns the first such instance (and stops testing at that point). The following example locates the first instance of the integer 8 in anArray:

```
anArray detect: [ :each | each == 8 ]
```

collect:

The **collect:** method performs a transformation on each element of the collection and returns a new collection containing the transformed objects. For example, to get an uppercase version of aString:

```
aString collect: [ :each | each asUppercase ]
```

inject:into:

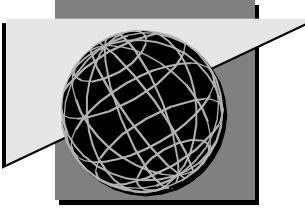
The **inject:into:** method enables you to pass an explicit argument to the block in addition to the collection's elements. This explicit argument (the **inject:** part of **inject:into:**) is used to initialize a counter for a cumulative operation such as summing. For example, to add the numbers in aSet:

```
aSet inject: 0 into: [ :subtotal :nextNbr | subtotal + nextNbr ]
```

Table 4-1 summarizes the branching and looping methods discussed in this chapter.

Table 4-1 Control Structure Methods

Method name	Description
ifTrue:	If the prior expression is true , execute the argument block.
ifFalse:	If the prior expression returns false , execute the argument block.
ifTrue:ifFalse:	If the prior expression is true , execute the first block; otherwise do the second block.
ifFalse:ifTrue:	Checks for a false condition first.
whileTrue:	Repeat the argument block until the receiver block is no longer true .
whileFalse:	Repeat the argument block until the receiver block is no longer false .
whileTrue	Repeat the receiver block until it no longer returns true .
whileFalse	Repeat the receiver block until it no longer returns false .
repeat	Repeat the receiver block until it executes a return or otherwise breaks the loop.
timesRepeat:	Repeat the argument block, using the receiving integer as a counter
to:by:do	Repeat for a specified interval, incrementing the counter by a specified value. Use the repetition counter as a block argument.
to:do:	Same as above, using 1 as the counter increment
do:	Repeat a block for each element in the receiver collection.
select:	Collect all elements that pass a test.
reject:	Collect all elements that fail a test.
detect:	Return the first element that passes a test.
collect:	Transform each element and return the transformed version of the collection.
inject: into:	Perform a cumulative operation such as summing the elements.



Chapter 5

Numeric Operations

ParcPlace Smalltalk provides several classes that represent elements in various kinds of linear series. They include various kinds of numbers as well as **Date** and **Time**. Operations involving these classes are discussed in this chapter. At the end of the chapter, the numeric classes are placed in the context of the class hierarchy with a discussion of their abstract superclasses.

Integers

The **Integer** class is an abstract superclass with three subclasses: **SmallInteger**, **LargePositiveInteger** and **LargeNegativeInteger**. The boundaries between **SmallInteger** and its larger neighbors occur at $2^{29}-1$ (536,870,911) and -2^{29} . Large integers have no size limit (other than memory availability). However, the system coerces integers into the proper subclass transparently, so you rarely need to pay attention to this issue.

Most of the behavior is defined in **Integer**, so in this section we will speak of integers generically.

instance creation and arithmetic

No specific methods are needed to create an instance of an integer (or any number) because they are typically created by calculations involving literals. Instances take the form of literals such as **101**, and are derived via arithmetic expressions such as **55 + 46**.

The usual arithmetic operations are supported, with three types of division:

- n Exact division (**/**), which returns a fraction if the result is not an integer (see “Fractions” on page 59)
- n Integer division (**//**), which returns the quotient rounded to the next lower integer (i.e., rounded toward negative infinity); use **\%** to get the corresponding remainder (modulo)

- n Truncated division (`quo:`), which returns the integer portion of the quotient; use `rem:` to get the corresponding remainder

Integers also support the following functions:

- n `abs` (absolute)
- n `factorial`
- n `gcd:`, which returns the greatest common divisor of two integers (the receiver and the argument)
- n `lcm:`, which returns the least common multiple of two integers (the receiver and the argument)
- n `negated` (reverse sign)
- n `raisedTo:`, which can also be written as double-asterisk (`**`); if the argument is an integer, `raisedToInteger:` is faster
- n `reciprocal`
- n `sqrt` (square root)
- n `squared`

testing

Integers return a `true` or `false` from the following methods:

- n `even`
- n `odd`
- n `negative`
- n `positive` (`>=0`)
- n `strictlyPositive` (`>0`)
- n `isInteger`
- n `isLiteral`
- n `isZero`

The `sign` method returns 1 if the receiver is positive, -1 if it is negative, and zero if it is zero.

comparing

Integer instances respond to the usual binary comparison messages (`=`, `==`, `<`, `<=`, etc.).

converting and printing

The `asCharacter` method returns a character whose ASCII value is the receiving integer. The expression `80 asCharacter` returns `$P`.

The `asFloat` method returns a floating-point representation of the integer, so `80 asFloat` returns `80.0`.

The `printString` method returns a string containing the integer. A radix integer is first converted to base 10, so the expression `16r11 printString` returns `'17'`. To print the integer in any base, use `printStringRadix:.`. Thus, `16r11 printStringRadix: 16` returns `'11'`.

Similarly, the `printOn:` method for printing an integer on a stream has a `printOn:base:` version for specifying a nondecimal base.

Floating Point Numbers

The `Float` class creates instances of single-precision floating point numbers between 1038 and -1038, with eight to nine digits of precision. The `Double` class creates double-precision floating point numbers between plus and minus 10307, with 14 to 15 digits of precision. Both `Float` and `Double` are subclasses of `LimitedPrecisionReal`, an abstract superclass that contains the behavior that is common to single- and double-precision floats.

In this section we will speak of floats generically.

instance creation and arithmetic

Instances take the form of literals such as `327.95`, and are derived via arithmetic expressions such as `301.50 + 26.45`. In scientific notation, a `Double` displays a `d` instead of `e`, as in `3.015d67`.

The usual arithmetic operations are supported, including the three types of division described above for integers.

Floats also support the following functions:

- n `abs` (absolute value)
- n `cos`, `sin`, `tan`, `arcCos`, `arcSin`, `arcTan`
- n `fractionPart`, which returns the fractional part of the number
- n `integerPart`, which returns the integer part of the number as a float
- n `ln` (natural log)
- n `negated` (reverse sign)

- n `raisedTo:`, which can also be written as double-asterisk (`**`); if the argument is an integer, `raisedToInteger:` is faster
- n `reciprocal`
- n `rounded`, which rounds to the nearest integer, and `roundTo:`, which lets you specify the rounding factor (such as 100)
- n `sqrt` (square root)
- n `squared`
- n `truncated`, which returns the integer part of the number as an integer

testing

Floats return a `true` or `false` from the following methods:

- n `evenNegative`
- n `oddPositive` (`>=0`)
- n `strictlyPositive` (`>0`)
- n `isLiteral`
- n `isZero`

The `sign` method returns 1 if the recipient is positive and -1 otherwise.

comparing

Floats respond to the usual binary comparison messages (`=`, `==`, `<`, `<=`, etc.)

converting and printing

Use `asInteger` to remove the fractional portion of the float and return an integer.

The `asRational` method converts a float to a rational number (integer or fraction). For example, the expression `7.5 asRational` returns the fraction `15/2`.

Floats also provide methods for converting `degreesToRadians` and `radiansToDegrees`.

The `printString` method returns a string containing the float.

Fractions

An instance of **Fraction** is a number with a numerator and a denominator, separated by a division slash, as in $3/4$. Fractions are always reduced to lowest terms.

Fractions respond to most of the messages described for integers and floats—exceptions are easily accessible to common sense and experimentation. Fractions come equipped with additional methods as follows:

- n An instance can be created via arithmetic or explicitly, as in the expression `Fraction numerator: 3 denominator: 4`.
- n `numerator` and `denominator`, for accessing the components
- n `asFloat` and `asDouble`, for converting to a floating point number

Random Numbers

An instance of class **Random** is a random number generator. The easiest method of generating a random number is with the expression `Random new next`.

The **Random** class comes with seven sets of parameters that correspond to seven generators. Each requires a starting value (called a “seed”) on which to perform arcane calculations resulting in a series of random floats. Thus, the more explicit means of creating an instance of **Random** is with the `fromGenerator:seededWith:` method, as in:

```
Random
  fromGenerator: 1
  seededWith: (Time millisecondClockValue)
```

The example, as it turns out, is what the `new` method does. It uses the first of the seven generators, seeded with the number of elapsed milliseconds on the system’s clock. (You can provide a literal seed number for a reproducible sequence of random numbers.) Once the generator has been created, send it the message `next` to get the next random number in the series. If multiple random numbers are needed, assign the generator to a variable and send `next` to the variable, as in the following loop:

```
| aGenerator |
aGenerator := Random fromGenerator: 7 seededWith: 234.
```

```
10 timesRepeat: [  
    Transcript show: ( aGenerator next) printString.  
    Transcript cr]
```

This code fragment prints ten random numbers in the Transcript. Each time it is executed, it prints the same ten numbers unless the generator or the seed is changed. That's why the seed is frequently derived from a near-random number itself, such as the current time.

Dates

In Smalltalk, a date is defined by a day and a year—so March 5, 1980 is day 65 in the year 1980. This is another way of saying that the `Date` class provides a template for two instance variables, called `day` and `year`. However, a date object comes equipped with methods for converting itself to and from standard month-day-year representations. Its public interface, therefore, makes it appear to be an object having a month, a day and a year.

instance creation

`Date` provides five class methods for creating a new instance, each using a different kind of input.

- n `newDay:month:year:` creates a date object from a day number, a month name and a year number. The month name must be a symbol, as in `#March`—only the unique first letters of the month name need to be given, so `#Mar` is sufficient in this example. If the century part of the year is omitted, the current century is assumed. To create an instance of `Date` with the value of March 5, 1980, use the expression `Date newDay: 1 month: #March year: 1980`.
- n `today` creates a date object with the current date as its value. The full expression is `Date today`.
- n `readFromString:` (inherited from `Object`) takes its input from a string, as in:

```
Date readFromString: 'March 5, 1980'
```

The string can begin with either the month or the day, though if both are integers the first will be assumed to be the month. The month can be a number or the (unique first letters of the) name. Any of the usual separators (space,

comma, hyphen, slash or nothing) can be employed. Thus, all of the following strings would be converted successfully:

- q 'March 5, 1980
 - q 'MAR 5 80'
 - q '3/5/80'
 - q '3-5-1980'
 - q '5 March 1980'
 - q '5MAR80'
- n Also available are `newDay:year`, which returns a date object that is a given number of days after the start of the specified year, and `from-Days:`, which is similar except that it uses 1901 as its starting year.

comparing

Date instances respond to the usual binary comparison messages (`=`, `==`, `<`, `<=`, etc.). Thus, the following expression returns true:

```
(Date today) > (Date readFromString: '4/1/01')
```

arithmetic

A number of days can be added to or subtracted from a date object, using `addDays:` and `subtractDays:`, as in `currentDate addDays: 7`, which returns a date seven days later than `currentDate`. The difference in days between two date objects can be found with `subtractDate:`.

accessing and inquiries

Table 5-1 lists methods for finding some atomic piece of information about a date.

Table 5-1 Date Methods

Method name	Object returned
<code>day</code>	Number of days since beginning of year
<code>weekDay</code>	Name of the day of the week, as a symbol
<code>dayOfMonth</code>	Number of days since beginning of month
<code>previous:</code>	Date of previous specified weekday

Table 5-1 *Date Methods*

Method name	Object returned
monthIndex	Month number
monthName	Month name
daysInMonth	Number of days in month
firstDayOfMonth	Number of days from beginning of year to first day of the month, inclusive
year	Year number
daysInYear	365 or 366 (until the calendar goes metric)
daysLeftInYear	Number of days to end of year
leap	1 if the date is in a leap year, otherwise 0

printing

The `printFormat`: method returns a string representation of a date object, using the format specified in the array that is passed as an argument. The array consists of six integers, as shown in Table 5-2.

Table 5-2 *Parts of a format array for printing date*

Array index	Purpose
1	Day's position in output (1, 2 or 3)
2	Month's position in output (1, 2 or 3)
3	Year's position in output (1, 2 or 3)
4	ASCII number of the separator character
5	Month format (1 = number, 2 = abbreviation, 3 = name)
6	Year format (1 = full number, 2 = last two digits)

For example, to print the current date in the format of March 5 1980:

(Date today) `printFormat: #(2 1 3 32 3 1)`

For most purposes, the `printString` method supplied by the `Object` superclass suffices. The expression `(Date today) printString` returns the current date in the format `5 March 1980`, though that format can be altered in the `printOn:` instance method supplied by `Date`.

`Date` also provides methods for printing a date object on a stream (`printOn:`, mentioned above, and `printOn:format:`), and for printing an executable expression on a stream (`storeOn:`) such that the expression will return the original date object.

Time

A time is defined by an hour, a minute and second relative to midnight. An instance of `Time` having `3:47:26 pm` as its value contains `15` in its `hours` variable, `47` in its `minutes` variable, and `26` in its `seconds` variable. `Time`'s public interface provides methods appropriate to this representation as well as a seconds-since-midnight representation.

instance creation

The `Time` class provides three methods for creating a time object:

- n `now` returns the current time, in the format `3:47:26`.
- n `fromSeconds:` returns a time that is a specified number of seconds past midnight. For example, `Time fromSeconds: 3661` returns a time object with the value `1:01:01 am`.
- n `readFromString:` takes its input from a string, as in:

```
Time readFromString: '3:47:26 pm'
```

The string can include leading zeros (`'03:47:26'`). The `'am/pm'` element can be in uppercase letters.

comparing

`Time` instances respond to the usual binary comparison messages (`=`, `==`, `<`, `<=`, etc.). Thus, the following expression returns `true` (except during the first hour after midnight):

```
(Time now) > (Time readFromString: '1:01:01 am')
```

arithmetic

A time object can be added to either a time object or a date object, using `addTime:.` The argument is converted to seconds past midnight (if it's a time object) or seconds since the beginning of 1901 (if it's a date object). That number of seconds is then added to the receiver, and the new instance of `Time` is returned in the usual hours-minutes-seconds format. Hours are not returned modulo 24—for example, the following expression returns a time of 694035:47:26 pm.

```
(Time readFromString: '3:47:26 pm')
  addTime: (Date readFromString: 'March 5, 1980')
```

Time can be subtracted in the same way, using `subtractTime:.`

accessing and inquiries

To find out some atomic piece of information about a time, use one of the methods in Table 5-3.

Table 5-3 Time Methods

Method name	Object returned
<code>hours</code>	Number of hours
<code>minutes</code>	Number of minutes
<code>seconds</code>	Number of seconds
<code>totalSeconds</code>	Number of seconds from the beginning of 1901 to the current time
<code>dateAndTimeNow</code>	An array containing current date and time
<code>millisecondClockValue</code>	Number of milliseconds since the system's clock was last reset

The last three methods listed above are class methods. `Time` also provides “general inquiries” that report the number of seconds since the beginning of 1901 as a four-element `ByteArray` (`timeWords`), and the milliseconds that transpire during execution of a block provided as its argument (`millisecondsToRun:`).

converting and printing

The `asSeconds` method converts a time object into a number representing the value of $(\text{hours} * 3600) + (\text{minutes} * 60) + \text{seconds}$.

The `printString` method inherited from `Object` returns a string version of the time object, so `Time now printString` returns a string containing the current time.

`Time` also provides methods for printing a time object on a stream (`printOn:`), and for printing an executable expression on a stream (`storeOn:`) such that the expression will return the original time object.

Time Zone

The `Time` class converts Greenwich Mean Time (GMT) to local time with the help of another class, `TimeZone`, on machines that report GMT rather than local time. `TimeZone` stores an offset from GMT for local time. In some parts of the world, this offset is not an integral number of hours, which is supported.

`TimeZone` provides an algorithm for determining whether DST is in effect. The algorithm relies on parameters that can be changed to suit local custom—by default, Daylight Savings Time is in effect from 2 a.m. on the first Sunday preceding April 7 to 2 a.m. on the first Sunday preceding October 31.

To change the day of the week from the Sunday preceding April 7 and October 31 to some other day, substitute the desired day of the week for `#Saturday` in the following expression:

```
TimeZone default weekDayToStartDST: #Saturday
```

To change other parameters in the default `TimeZone`, create a new instance of `TimeZone` with the desired parameters, then pass that instance as an argument to the `setDefaultTimeZone:` method, as follows:

```
| newTZ |
newTZ := TimeZone
  timeDifference: -8
  DST: 1
  at: 4
  from: 97
  to: 305 .
"Create instance with parameters..."
"Offset 8 hours from GMT"
"DST is different by 1 hour"
"Start/end DST at 4 a.m."
"Start DST on 97th day of year"
"End DST on 305th day"
```

```
newTZ weekDayToStartDST: #Tuesday.      "Day to start/end DST"  
TimeZone  
setDefaultTimeZone: newTZ              "Install new default"
```

In a few locations, the algorithm for determining the beginning and ending of Daylight Savings Time is different from the algorithm described above. To accommodate such a time zone, you will need to alter the code in the `TimeZone` instance method called `convertGMT:do:`.

Abstract Superclasses

The concrete classes discussed in this chapter all have `Magnitude` as a common superclass. For `Date` and `Time`, it is the direct parent.

The number classes have `ArithmeticValue` as an intermediate superclass, implementing much of the shared behavior such as arithmetic operations.

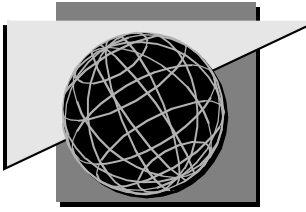
Yet another intermediary superclass, called `Number`, implements much of the behavior specific to scalar quantities.

The following hierarchy illustrates the relationships of the abstract and concrete classes described in this chapter. It omits classes such as `Point` that exist in the hierarchy but are described elsewhere in the documentation. Abstract classes are underlined.

```
Object  
  TimeZone  
  Magnitude  
    Date  
    Time  
    ArithmeticValue  
      Number  
        Fraction  
        Integer  
          LargeNegativeInteger  
          LargePositiveInteger  
          SmallInteger  
        LimitedPrecisionReal  
          Double  
          Float
```

When accessed via the System Browser, all of the classes discussed in this chapter occur within one of two class categories:

- n Magnitude-General (Magnitude, Date, Time, TimeZone)
- n Magnitude-Numbers (the remainder)



Chapter 6

Collection Operations

ParcPlace Smalltalk provides a wide variety of classes for operations involving collections of objects. In addition to the conventional arrays, there are bags, dictionaries, sets, linked lists, and more. Operations involving these classes are discussed in this chapter.

The first section presents the distinguishing features of the collection classes, with the aim of helping you choose the best class for a specific use. A decision tree (see Figure 6-1) provides a quick reference when making such a choice.

Each of the subsequent sections deals with a specific task, such as adding elements to a collection. The usual behavior is described first in each section, followed by notes about specialized behavior.

By way of summary, the final section describes the collection classes in the context of the class hierarchy, with a discussion of their abstract superclasses.

Iterative operations involving collections are discussed in detail in Chapter 4, “Control Structures.” A string of characters is also a collection and shares much of the behavior of other collections. It is discussed as a special case in Chapter 7, “String Operations.”

Choosing the Appropriate Class

There are nine main kinds of collections. Three of them have specialized variations. A brief description of each collection class follows, proceeding from the simplest to the more complex. As a rule of thumb, choose the simplest class that suits your purpose.

Set

A **Set** is about as close to a generic collection as you can get. No index. No sorting. It does discard duplicates, which is often useful. The fact that an instance of **Set** has only one special capability should not distract you from

the fact that the generic behavior it inherits, as described in later sections of this chapter, includes powerful mechanisms for manipulating elements of a data set.

An `IdentitySet` is identical in all respects, except that it uses `==` for comparisons instead of `=`.

Bag

An instance of `Bag` is just like a `Set`, except that it counts the duplicates as it discards them. Thus, for each element in a `Bag` there is also a tally of the occurrences of that object. If each character in the word collection were an element in a `Bag`, for example, the tally for the element `$c` would be 2.

Another way of looking at a `Bag` is that it is a `Set` that does not discard duplicates. But since the elements are not ordered in any particular way, the most we can hope to know about an element such as `$c` is how many times it occurs. `Bag` does not waste memory by creating a new element for a duplicate, but increments a counter instead.

Array

`Array` allows you to maintain relative positions of elements, via an integer index. In our collection example, `$e` can be identified by its external key, the integer 5. (In a `Set` or a `Bag`, by contrast, the position of `$e` is unpredictable.) As another example, if a customer name were to be stored as a collection of three elements—first, middle and last names—it would make sense to use an `Array` rather than a `Set` because the relative positions of the elements must be preserved.

A `RunArray` provides efficient storage for situations in which a value is repeated consecutively over long stretches of an array. For example, the font information for a block of text is a likely candidate—a roman font would be used for many sequences of elements in the array (letters in the text), with occasional bursts of italic, bold, etc. Although `RunArray` responds to the same messages as `Array`, its internal representation avoids waste by storing an element only if it differs from the preceding element, along with a tally of that element's repetitions.

A `ByteArray` provides space-efficient storage for bytes. Its elements are restricted to the set of `SmallIntegers` from 0 to 255. `WordArray` is for manipulating 16-bit words; its elements can be integers from 0 to 65535.

Interval

An **Interval** is a finite arithmetic progression, such as the series 2 4 6 8. It is typically used to control an iterative loop, as described in Chapter 4, “Control Structures.”

OrderedCollection

An **OrderedCollection**, like an **Array**, has an integer index and accepts any object as an element. Unlike **Array**, however, an **OrderedCollection** permits elements to be added and removed freely. It is frequently used as a stack (the last element in is the first one removed) or a queue (first in, first out). However, it uses extend farther because there are so many situations in which ordering must be preserved as an arbitrary number of elements are added.

SortedCollection

When elements are not added in the desired order, sorting is required. **SortedCollection** provides that extra capability. By default, elements are sorted in ascending order. You can override this default by specifying an alternative sort algorithm enclosed in a block. For example, the expression **SortedCollection sortBlock: [:x :y | x >= y]** creates a new collection whose elements will be sorted in descending order.

LinkedList

As its name suggests, a **LinkedList** is a collection in which each element points to the next element. An **OrderedCollection** can accomplish the same thing, but is less efficient in circumstances involving large numbers of additions and deletions. For example, the **ProcessorScheduler** class makes use of **LinkedList** to track the highly dynamic list of processes. **LinkedList** achieves its efficiency in a way that prohibits its elements from belonging to other collections at the same time.

Dictionary

The **Dictionary** class, instead of imposing an integer index on each element, permits any object to be the external key. The result, as in the familiar Webster’s dictionary, is a collection of key-value pairs. For example, an element might consist of the word ‘object’ with the associated definition ‘something solid that can be seen or touched’. Thus, each element in a Dictio-

nary is typically an instance of **Association**, which is a key-value pair. The `nil` object is specifically excluded as a valid element.

An **IdentityDictionary** is similar, except that it uses `==` for comparisons instead of `=`. That is, the values in an **IdentityDictionary** are expected to be literals or other unique objects that can be compared with the more efficient identity operator (`==`).

Table 6-1 Summary of Collection Classes

Collection class	Distinguishing features
Set	Discards duplicate elements
Bag	Tallies duplicates
Array	Integer index (and fastest access)
Interval	Integer elements in progression
OrderedCollection	Integer index; preserves the order in which elements are added
SortedCollection	Integer index; elements are sorted by user-defined algorithm (ascending order is default)
LinkedList	Each element points to the next element, for maximum efficiency of dynamic lists
Dictionary	noninteger index; each element consists of a key-value pair for dictionary-like lookups

Creating an Instance

The simplest creation message is `new`, as in `Set new`. This works for all collections except **Interval** and **LinkedList**—they have custom creation messages that require arguments.

To specify a starting size for the collection, use `new:` (not applicable to **Interval** or **LinkedList**).

To specify the first element in the collection, use `with:`, as in the expression `Set with: #colorNbr`. Up to four elements can be specified in this way, as in

OrderedCollection with: \$p with: \$d with: \$q with: \$!. (Not applicable to Interval.)

Table 6-2 Instance Creation

Array	To specify a starting size and fill all elements with a default, use <code>new: withAll:</code> , as in <code>Array new: 10 withAll: 99</code> , which creates an <code>Array</code> with 10 elements each containing the integer 99.
Interval	An <code>Interval</code> is normally created indirectly, by sending a <code>to:</code> message to a number, as in <code>1 to: 10</code> , which creates an interval containing the integers 1 through 10. An increment argument can also be specified, as in the expression <code>1 to: 10 by: 2</code> . (<code>Interval</code> provides a <code>from:to:by:</code> creation method, which is sent by the <code>Number</code> class when an <code>Interval</code> is created indirectly.)
SortedCollection	To specify a sorting algorithm in the form of a block of expressions, use <code>sortBlock:</code> , as in the expression <code>SortedCollection sortBlock: [:x :y x >= y]</code> (that is, create an instance of <code>SortedCollection</code> with elements sorted in descending order). The default sort order emplaced by the <code>new</code> creation message is ascending order.

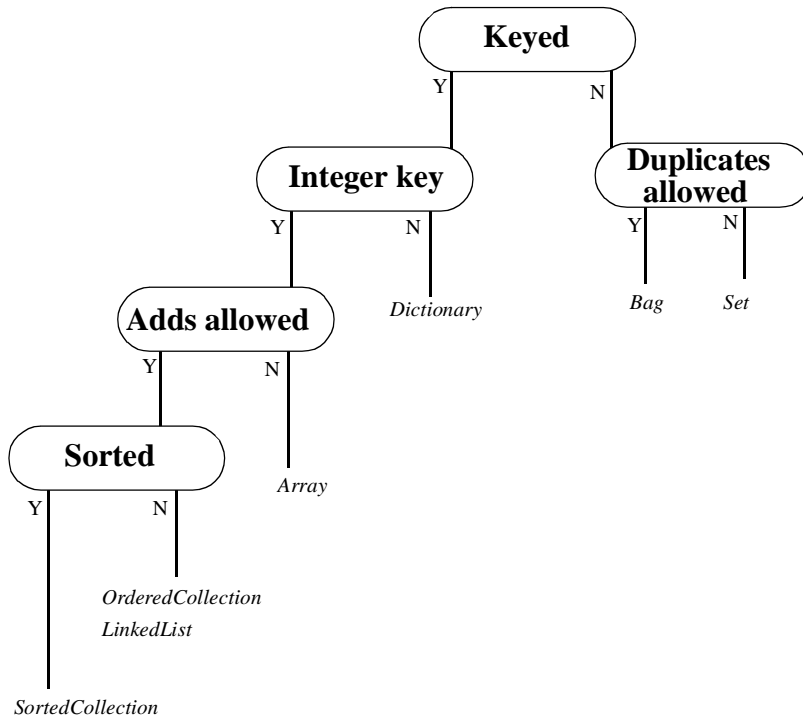


Figure 6-1 Collection Class Decision Tree

Adding, Removing and Replacing Elements

To append a new element to an existing collection, use `add:`, as in the expression `aSet add: anObject`. (Not applicable to `Array` or `Interval`.)

To append the contents of an entire collection, use `addAll:`, as in `aCollection addAll: anotherCollection`.

Use `remove:` to delete a single element. Use `removeAll:` to delete all of the elements in a subcollection, as in `aSet removeAll: subSet`. (Not applicable to `Array` or `Interval`.) By default, an error notification occurs when the element to be removed is not found; to specify an alternate action block, use `remove:ifAbsent:`.

If the collection has an index, use `at:put:` to replace the existing value with a new value. For example, `anArray at: 3 put: #Done` stores the symbol `#Done` as the third element in `anArray`.

Bag—Because **Bag** keeps a tally of occurrences for each element, it provides an `add:withOccurrences:` method that lets you specify multiple occurrences of the element being added.

Array—To replace all elements of an array with a specific object, use `atAllPut:`, as in the expression `boxColors atAllPut: #burntOrange`. This is not to be confused with `atAll:put:` (notice the two colons), which replaces only those elements with indices in the interval that is passed as the first argument. For example, `boxColors atAll: (11 to: 20) put: #darkBlue` only replaces the values of the elements in positions 11 through 20. **Array** also provides methods for:

- n Replacing all elements that have a specific value with a new value (`replaceAll:with:`)
- n Replacing all elements of a specific value with a new value, within a range of indices (`replaceAll:with:from:to:`).
- n Substituting values from one collection into another (`replaceFrom:to:with:`), optionally starting at a specific location in the replacement array (`replaceFrom:to:with:startingAt:`)
- n Replacing all occurrences of a subarray with another array (`copyReplaceAll:with:`)
- n Replacing the elements having a specified series of indices with an array (`copyReplaceFrom:to:with:`).

OrderedCollection—As a subclass of **SequenceableCollection**, from which **Array** receives the special behavior described above, **OrderedCollection** shares those behaviors.

Additional refinements to the basic adding and removing methods are available to instances of **OrderedCollection**. Their names describe their functions adequately, so a list of them will suffice:

- n `add:after:`
- n `add:before:`
- n `add:beforeIndex:`
- n `addFirst:`
- n `addLast:`
- n `addAllFirst:`
- n `addAllLast:`
- n `removeFirst`
- n `removeFirst:`

- n `removeLast`
- n `removeLast:`
- n `removeAtIndex:`
- n `removeAllSuchThat:`

SortedCollection—As a subclass of `OrderedCollection`, `SortedCollection` inherits all of the refinements that apply to `OrderedCollection`.

LinkedList—Use `addFirst:` to add a link to the beginning of the list, or `addLast:` to append it at the end. The complementary methods `removeFirst` and `removeLast` are also supported.

Dictionary—To append an element from another dictionary, use `declare:from:`, as in `aDict declare: #manager from: personnelDict`. This example copies the association having `#manager` as its key from `personnelDict` into `aDict`.

Comparing Collections

Use the equals sign (=) to test for equivalence. Two collections of unlike class (`Set` and `Array`, for example) will not compare equal, even when they have identical elements. The not-equal method (`~=`) can also be used.

As with other objects, the `==` and `~~` methods can be used to compare two collections for identity match and mismatch, respectively.

Counting and Finding Elements

To find out how many elements a collection has, use `size`.

To find out whether a collection has a particular value among its elements, use `includes:.` Use `occurrencesOf:` to count the number of times a value is repeated in a collection.

To find out whether a collection has zero elements, use `isEmpty`.

Several *enumeration* methods are available for repeating a block of expressions for each element in a collection (see “Looping” on page 48). Those methods can be used to find elements that meet specific criteria, or to create a transformed copy of a collection.

Bag—Two methods are provided for getting a sorted listing of the values along with a count of the occurrences of each value. One method sorts the

counts, in descending order (`sortedCounts`) and the other method sorts by value, in ascending order (`sortedElements`).

Array—To find the beginning element, use `first`. To find the final element, use `last`.

To get the index number corresponding to a value, use `indexOf`. To confine the search to a range of indices, use `nextIndexOf:from:to:` (searching forward) or `prevIndexOf:from:to:` (searching backward). A similar method finds the location of a subcollection (`indexOfSubCollection:startingAt:`).

OrderedCollection—The refinements noted for **Array** also apply to this class. In addition, `after:` returns the element that follows the argument and `before:` returns the preceding element. For example, an OC `after: #burntOrange` finds `#burntOrange` in an OC and returns the element after it.

SortedCollection—The behavior mentioned for **Array** and **OrderedCollection** also applies to this class.

Dictionary—Each element of a **Dictionary** is a key-value pair, so this class provides extensions to the `at:` method, which returns only the value. Use `keyAtValue:` to return the key, or `associationAt:` to return both the key and its associated value.

To get a collection of all the keys in a **Dictionary**, use `keys`. Use `values` to get a collection of the values. By using a **SortedCollection** in such a maneuver, you can compile a sorted listing of keys or values.

Copying a Collection

The `copy` method, inherited from **Object**, creates a new collection that shares the instance variables of the original collection.

Array, **OrderedCollection**, **SortedCollection**—Use `copyWith:` to create a copy and append the argument as a new element. The `copyWithout:` method creates a copy that omits all occurrences of the argument.

To copy a subset of a collection, use `copyFrom:to:`, as in the expression `oneThruThreeArray := (oneThruSixArray copyFrom: 1 to: 3)`.

Converting and Printing

Use `printString` to convert a collection into a printable string of the form `collectionName (element1 element2 ...)`. The `printOn:` method performs that conversion and then outputs the string onto the argument stream.

The `storeString` method also creates a descriptive string, but in a format that permits the collection to be reconstructed from the string. Use `storeOn`: to output a collection's `storeString` to a stream.

Any collection can be converted to one of the simpler types, using `asBag`, `asOrderedCollection`, or `asSortedCollection`.

`Array`, `OrderedCollection`, `SortedCollection`—Collections belonging to one of these classes also respond to `asArray`, `readStream` (which creates a read-only stream on the collection) and `writeStream`.

The Collection Hierarchy

The concrete classes discussed in this chapter all have `Collection` as a common superclass that provides a great deal of the behavior. For `Set` and `Bag`, it is the direct parent.

`Dictionary` is a subclass of `Set`.

The remaining concrete classes (`Array`, `Interval`, `OrderedCollection` and `SortedCollection`) all have an intermediate superclass called `SequenceableCollection`, which provides the machinery for dealing with a well-defined ordering of elements.

`Array` has yet another intermediate superclass, `ArrayedCollection`, which provides behavior associated with an integer index.

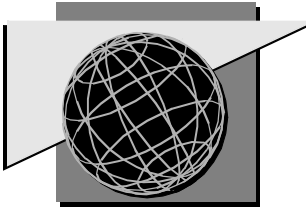
The following hierarchy illustrates the relationships of the commonly used classes described in this chapter. It omits classes such as `WeakArray` that exist in the hierarchy but are described elsewhere in the documentation. Abstract classes are underlined.

```
Object
  Collection
    Bag
    Set
      Dictionary
    SequenceableCollection
      Interval
      LinkedList
      OrderedCollection
      SortedCollection
```


ArrayedCollection Array

When accessed via the System Browser, all of the classes discussed in this chapter occur within one of four class categories:

- n Collections-Abstract (Collection, SequenceableCollection, ArrayedCollection)
- n Collections-Unordered (Set, Bag, Dictionary)
- n Collections-Sequenceable (Interval, OrderedCollection, SortedCollection, LinkedList)
- n Collections-Arrayed (Array)



Chapter 7

String Operations

As you might expect, characters and strings are primarily manipulated via two classes called **Character** and **String**. This chapter discusses operations at the character level first, followed by string operations.

The final section places **Character** and **String** in the context of their abstract superclasses and, in the case of **String**, its concrete subclasses.

As a collection of characters, a string responds to the messages described in Chapter 6, “Collection Operations.” The more pertinent behavior is reviewed in this chapter.

Creating a Character

When a character is created as a separate entity (rather than as part of a string of characters), the usual way to create it is by preceding the letter with a dollar sign. Thus, `firstLetter := $a` stores the first letter of the alphabet in variable `firstLetter`, while `shortString := 'a'` assigns a string containing that letter.

A character can also be created from its numeric equivalent with the `value:` method, so the expression `$M = (Character value: 77)` is true.

ParcPlace Smalltalk also supports an extended set of characters, which provides for international variations on the Roman alphabet (œ, ¥, ë, ÿ, etc.), among other things. This extended set of characters, with values from 128 to 65535, conforms to the Xerox Character Code Standard (consult Xerox Corporation publication XNSS 058710), with the exception that the codes for the dollar sign (\$) and the international currency symbol (₤) have been swapped to conform to ASCII. The `value:` method can also be used to create an instance of one of these characters, though correct displaying depends on that character being present in the selected font.

Characters with codes between 128 and 255 also coincide with the ISO 6937 standard (with the same exception for \$ and ₤). Of these, codes 193-207

represent nonspacing diacritical marks that are not normally used independently. Instead, codes in the range 16rF100 to 16rF1FF represent composite characters consisting of a base character plus a diacritical mark. The methods `basePart` and `diacriticalPart` provide access to the components of such a character.

Note: Any application that manipulates characters should be prepared to encounter any character value from 0 to 65535.

Character Operations

Because the extended character set contains so many subsets, `Character` provides a variety of tests to help you pigeonhole an instance:

Table 7-1 Character Tests

Method	Returns true if the character is...
<code>isLowercase</code>	a-z or a lowercase special character
<code>isUppercase</code>	A-Z or an uppercase special character
<code>isAlphabetic</code>	a-z, A-Z or a special character
<code>isVowel</code>	in the set: AEIOUaeiou (with or without diacritical marks)
<code>isDigit</code>	0-9
<code>isAlphaNumeric</code>	a-z, A-Z, 0-9 or a special character
<code>isSeparator</code>	space, cr, tab, line feed, form feed or null
<code>isDiacritical</code>	a diacritical mark (has a value in the range 16rC1 to 16rCF)
<code>isComposed</code>	composed of base and diacritical parts (has a value of 16rF100 or higher)
<code>isLetter</code>	English alphabet or extended character

To derive the integer equivalent of a character, use `asInteger`. To change the case, use `asUppercase` or `asLowercase`.

A `Character` can be compared to another character with the usual binary comparison methods (`=`, `>`, etc.).

To combine base and diacritical characters to form a composite, use `composeDiacritical:`, as in `aBaseChar composeDiacritical: aDiacritical`.

Creating a String

A string literal is any sequence of characters enclosed in single quotes (double quotes are for code comments), so the usual method of creating a string is to put single quotes around the desired words. A string can also be manufactured from an array of integers representing character codes (`fromIntegerArray:`), from a stream (`readFrom:`) or as a string of a specific number of null characters (`new:`).

Substring Manipulations

Use `size` to count the characters in a string. Having that information, you can use `at:` to retrieve the character at a specific location, or `at:put:` to replace a specific character. Use `first` or `last` to retrieve the beginning or final character.

To find out whether a specific character exists in a string, use `includes:`. To count the number of times a character occurs in a string, use `occurrencesOf:`. To find the index location at which a character first occurs, use `indexOf:`, as in `'Contract 88-36' indexOf: $-`. To confine the search to a range of indices, use `nextIndexOf:from:to:` (searching forward) or `prevIndexOf:from:to:` (searching backward).

To find a substring, use `findString:startingAt:`, as in the expression `'last line of codeEND' findString: 'END' startingAt: 1`. To start the search farther along in the string, use a larger number for the `startingAt:` argument.

To combine two strings, use a comma as in the expression:

```
salutation := 'Dear ', addresseeName.
```

To copy the beginning characters of a string, use `copyUpTo:`, and specify the number of characters to be copied as the argument. To copy a substring having a specific set of indices, use `copyFrom:to:`, as in `InputString copyFrom: 1 to: 5`.

To insert one string inside another, use `copyReplaceFrom:to:with:`, as in `'Steenon' copyReplaceFrom: 3 to: 2 with: 'ph'`. Note that these index locations are purposely arranged so as to replace *no* characters in the original string, inserting instead. To replace instead of inserting, use an index number for the `to:` argument that is equal to or larger than the `from:` argument.

To collapse a string to a specific size, use `chopTo:`. This method does not merely truncate to achieve its goal, it cuts out the middle, leaving near-equal

fragments from the beginning and end of the original string. In some situations, this may result in a more recognizable remainder than truncation would yield—a pathname, for example. To replace the deleted characters with an ellipsis (...), use `contractTo:`. To drop all vowels other than a leading vowel, use `dropFinalVowels`.

To convert all carriage returns to newlines, use `crsToNewlines`. The complementary method is called `newlinesToCRs`. Embedded backslash characters (`\`) in a string can be converted to carriage returns by using `withCRs`.

As a collection of characters, a string also provides looping methods such as `collect:` and `select:`, which can be used to repeat an algorithm for each character. See “Looping” on page 48 for more details.

Pattern Matching

To count the number of beginning characters that match in two strings, use `sameCharacters:`, as in `'INV90467'.sameCharacters:'INV90413'`, which returns 6.

To compare two strings while ignoring case differences, use `=`. Thus, `'exit' = 'Exit'` evaluates to `true`. To compare using case differences, use `trueCompare:`.

The `match:` method does the same thing, and also supports two wildcard symbols. A pound sign (`#`) can be used in place of any single character; an asterisk (`*`) can be used in place of zero or more characters. For example, `'Ms. #. *'` `match:'Ms. D. Gillen'` is `true`.

To control whether case is ignored, use a variant of the preceding method, `match:ignoreCase:`, specifying `true` or `false` as the second argument.

In contrast to the boolean comparisons discussed above, the `spellAgainst:` method provides a quantitative comparison. It returns a value between 0 and 100 indicating the similarity of the two strings (100 is an exact match). No case conversion is performed. For example, the expression `'graph'.spellAgainst:'grape'` returns a value of 80, because 80 percent of the characters are identical.

The String Hierarchy

The `Character` class is a subclass of `Magnitude`, the abstract superclass for numbers, dates, times and other objects that represent a magnitude. The

following hierarchy, with abstract classes underlined, illustrates `Character`'s place in the system:

```

Object
  Magnitude
    Character
  
```

The `String` hierarchy is much more complex. As mentioned before, a `String` is a collection of characters, so it is descended from `Collection`. It has three intermediate superclasses, `SequenceableCollection`, `ArrayedCollection`, and `CharacterArray`. The first two are discussed in “The Collection Hierarchy” on page 78—from them, a string inherits the structure and behavior that facilitates element-level operations.

The `CharacterArray` superclass provides behavior that is common to both strings and text. A `Text` object is a string that has font attributes, and is discussed in detail in the *VisualWorks Cookbook*.

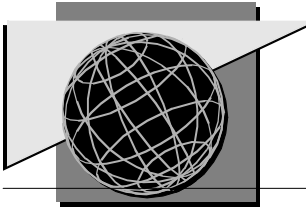
`String` is itself an abstract superclass, providing string-like behavior that is inherited by various string implementations: `ByteString`, `TwoByteString`, and `Symbol`, among others. `ByteString`, a string of characters with each character encoded as a byte, is the default string class. That is, the expression `String new` will return an instance of `ByteString` instead of `String`. However, when a character whose value exceeds 255 is stored into a `ByteString`, it will automatically be converted to a `TwoByteString` to accommodate it. Explicit conversion is achieved via the `fromString` method, as in `ByteString fromString: aMacString`.

The following hierarchy places `String` in the context of its superclasses and subclasses. Abstract classes are underlined.

```

Object
  Collection
    SequenceableCollection
      ArrayedCollection
        CharacterArray
          String
            ByteEncodedString
              ByteString
              ISO8859L1String
              MacString
              Symbol
          
```

ByteSymbol
woByteSymbol
TwoByteString



Chapter 8

Processes and Exception Handling

Besides control blocks, ParcPlace Smalltalk provides two other mechanisms for controlling the flow of execution. One facilitates the control of multiple independent processes, and the other provides a sophisticated apparatus for handling errors.

Creating a Process

A Smalltalk process is a light-weight process that is non-preemptive with respect to other processes of the same or lower priority. It represents a sequence of actions being performed by the computer. Frequently, two or more such processes need to be running simultaneously. For example, you might wish to assemble an index in the background at the same time as your application user is performing an unrelated activity such as entering data. In that case, the computer's attention must be divided between the two activities—in effect, we want to place a fork in the path so the processor will progress down both paths at the same time.

To split a new process to run alongside an existing one, send the message `fork` to a block of expressions, creating a new instance of `Process`. If the indexing operation mentioned above were capable of being launched from within the data-entry program, the expression for doing so would look something like `indexingBlock fork`, where `indexingBlock` is a block containing the launching instructions for the index program.

The `fork` message triggers execution of the block's contents just as a `value` message would. The difference is that the next instruction following the `fork` is executed immediately. The instruction that follows a `value` has to wait until the block has finished, which is undesirable in the case of a background process such as an indexing operation.

A block's response to `fork` is to create a new instance of `Process`, then notify the `Processor` to add the new process to its work load. This latter step is known as scheduling a process.

To create a new process without scheduling it, use `newProcess` instead of `fork`. In effect, the newly created process is immediately suspended, presumably so it can be restarted by another part of your program at the appropriate moment. In that way, the creation of the process can be separated from the scheduling.

To pass one or more arguments to a processing block, use `newProcessWith:`, supplying the argument objects in an `Array`, as in `aBlock newProcessWith: #(2 #NewHire)`. The number of elements in the `Array` must be equal to the number of block arguments.

Scheduling a Process

`Processor` is the lone, pre-fabricated instance of class `ProcessorScheduler`, in the same way that `Smalltalk` is the unique instance of class `SystemDictionary`. Both are global variables. `Processor` is responsible for deciding which instruction to execute next, choosing among the next actions in all of the current processes. It has to be made aware of a process first—the process has to be scheduled.

The `fork` message, described above, automatically schedules its newly created process. To schedule a suspended process (including a process created with a `newProcess` message), use `resume`, as in the expression `aProcess resume`.

To temporarily prevent execution of a process's instructions, use `suspend`. Thus, `resume` and `suspend` are complementary methods. A resumed process starts up where it left off when it was suspended.

To unschedule a process permanently, whether it is in `resume` or `suspend` mode, send it the message `terminate`.

Setting the Priority Level

The `Processor` has a great deal in common with a juggler who spins plates on the tops of those long, wobbly poles and then scurries from one to another, acutely attentive. Like the juggler, who services whichever plate is wobbling the most and spinning the least, `Processor` lets its processes set their own priority levels. Otherwise, it handles them in the order in which they were scheduled.

There are 100 possible priority levels. Eight of the levels are commonly used and can be accessed by name in code references. The lowest level is reserved,

so it does not have an access method. Table 8-1 describes the purpose of the remaining priority levels, starting with the most dominant.

Table 8-1 *Priority Levels*

Priority number	Method	Purpose
100	timingPriority	Processes that are dependent on real time
98	highIOPriority	Critical I/O processes, such as network input handling
90	lowIOPriority	Normal input/output activity, such as keyboard input
70	userInterruptPriority	High-priority user interaction; such a process pre-empts window management, so it should be of limited duration
50	userSchedulingPriority	Normal user interaction
30	userBackgroundPriority	Background user processes
10	systemBackgroundPriority	Background system processes

A newly created process inherits the priority level of the process that created it.

To assign a new priority to a process, use an expression of the form `aProcess priority: (Processor userInterruptPriority)`. Notice that the `priority: method` expects an integer argument, but the sender asks the `Processor` for the integer by name.

You can also specify the priority level at process creation time, using `forkAt:` with the requisite priority level integer.

The `Processor` gives control to the process having the highest priority. When the highest priority is held by multiple processes, the active process can be moved to the back of the line with the expression `Processor yield`—otherwise it will run until it is suspended or terminated before giving up the processor. A process that is yielded will regain control before a process of lower priority.

Coordinating Processes with a Semaphore

Sometimes one process has to wait for another process to mature before it can take a particular action. For example, a printer might be tied up for the next 20 minutes printing someone else's job. Does that mean your printing job should just tie up the Processor and refuse to yield until the printer is available?

The `Semaphore` class provides a simple mechanism for resolving such problems. In our example, an instance of `Semaphore` would be created to keep an eye on the printer: `printerSemaphore := Semaphore new`. The process that funnels output to the printer, which we'll call `printerProcess`, sends the message `printerSemaphore signal` each time it becomes available for more input. The waiting process, which has been suspended so it won't lock up the Processor, is then resumed.

How did the waiting process get suspended in the first place?

Instead of just dumping its contents and assuming they would be caught by `printerProcess`, the waiting process sent the message `printerSemaphore wait`. Because `printerSemaphore` had not yet received a `signal` message from `printerProcess`, the waiting process was suspended. If the `printerProcess` had already sent a `signal` message that was not consumed by another process, `printerSemaphore` would have done nothing, permitting the waiting process to dump its load.

If a `Semaphore` receives a `wait` from two or more processes, it resumes only one process for each `signal` it receives from the process it is monitoring. A `Semaphore` resumes the oldest process of the highest priority.

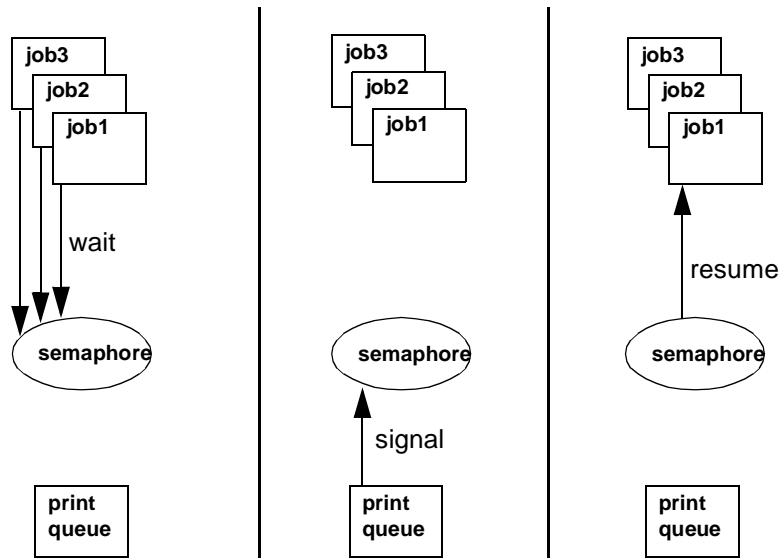


Figure 8-1 The three steps involved in using a semaphore

A Semaphore is like a guard who permits one person to approach the Queen at a time. Each time the Queen finishes an audience, she looks up at the guard and says **signal**. The guard then lets the next courtier in. (To add to the indignity, a courtier will not receive a place in line unless he or she gives the password to the guard: **wait**.)

Thus, a process can be in any of four different states: *suspended*, *waiting*, *runnable*, and *running*. The first two are very similar, with the distinction that explicit **suspend** and **resume** messages push a suspended process from or into runnability, while primitive semaphore methods accomplish the same for a waiting process. A runnable process is ready to go as soon as the **ProcessorScheduler** gives it permission. A running process is the one that the processor is working on.

Passing Data Between Processes

When an application needs to match the output of one process with the input for another process, care must be taken to make sure the transfer of data goes

as planned. The `SharedQueue` class provides a means of coordinating this transfer.

To create a `SharedQueue`, use `new` or `new:` with an integer argument specifying the number of desired slots.

To store an object in the `SharedQueue`, send it a `nextPut:` message with the data structure as argument. If another process has been waiting for an element to be added to the queue, which it indicated by sending `next` to the `SharedQueue`, that process will be resumed.

Using a Delay

The `Delay` class answers the common need for a means of postponing a process for a specific amount of time. To create a `Delay`, use `forSeconds:`, as in `Delay forSeconds: 30`. Or use `forMilliseconds:` if you require a finer quantification of time.

To create a `Delay` that continues until the system's millisecond counter reaches a particular value, use `untilMilliseconds:`. To find out the current value of the counter, use the expression `Delay millisecondClockValue`.

Merely creating a `Delay` has no impact on the current process. The process must send the `wait` message to the instance of `Delay`. Thus, the following expression in a method would suspend the current process for 30 seconds: `(Delay forSeconds: 30) wait`.

Using a Signal to Handle an Error

Error conditions generally result in creation of a notifier window with a predefined error message. For example, an attempt to divide an integer by zero results in a notifier that says, "Can't create a Fraction with a zero denominator." Sometimes it is desirable to provide a response that is more meaningful in the context of your application. If a divide-by-zero error occurred in a calculator application, for example, it would mean that a divisor had been entered incorrectly, and that's what you would tell the user.

Another reason to intervene is that the remote method, which performs the division and encounters the error, can only stop the program and proceed on command. It cannot go back to the data-entry part of the process, giving the user a chance to correct the error, because that is contained in the calling method. So it is in the calling method that we must provide an intelligent handler for the error.

The **Signal** class provides a mechanism for catching an error that occurs in some remote method and handling it locally. You can think of **Signal** as a hawk-like observer. When things go along smoothly, **Signal** just watches. But when an error surfaces, it swoops down and alters the flow of events as only a bird of prey can do. (Not just any error will do, though—hawks have specific appetites.)

Emplacing such an observer involves creating an instance of **Signal** and telling it what you plan to do and how to handle an error. This is accomplished with a **handle:do:** control structure. In pseudocode form, the resulting expression for our calculator's division method is:

```
aSignal
  handle: [error handling code]
  do: [the division operation].
```

The error that triggers the **handle:** block is an instance of **Exception**. Hence, dynamic error trapping in Smalltalk is usually called exception handling. An **Exception** is created by a **raise** message sent to a **Signal**. In our example, the method that performs the actual division would send a message such as:

```
aSignal raise
```

Thus, exception handling involves two steps: Placing a **Signal** handler to watch over a block of expressions, and raising an **Exception** when an error occurs.

Choosing or Creating a Signal

To create a new instance of **Signal**, use **Signal new**. The resulting instance has a parent of **Object errorSignal**—the significance of this ancestry is discussed below. To create a signal with a different parent, use **newSignal** and address it to the desired parent, as in the expression **divSignal := (Number errorSignal) newSignal**.

Many of the classes in the system already contain useful **Signals** as class variables, so it may be appropriate to choose an existing signal instead of creating a new one. These global signals are accessed via class methods. We've already mentioned the **ErrorSignal** in the **Object** class, which is accessed via the expression **Object errorSignal**. (The method name does not always match the signal name; for this reason, we refer to a global signal by the expression used to access it, rather than by its explicit name.)

Object `errorSignal` is the parent or grandparent of all other signals, with rare exceptions. This all-encompassing lineage permits it to catch any error. Naturally, that generality carries over to its response mechanism, reducing its usefulness in situations demanding a specialized error message or other response.

Under Object `errorSignal` is Number `errorSignal`, among others. Number `errorSignal` restricts its attention to numeric errors. It is the parent of several other signals, including ArithmeticValue `divisionByZeroSignal`, the specialized signal that suits our calculator's purposes. Our pseudocode example would then look like:

```
ArithmeticValue divisionByZeroSignal
  handle: [error handling code]
  do: [the division operation]
```

Before creating a new signal, consider whether an existing signal will serve the purpose. The following hierarchy contains the most commonly reused signals in the system:

```
Object errorSignal
  Object notFoundSignal
    Object indexNotFoundSignal
      Object subscriptOutOfBoundsSignal
      Object nonIntegerIndexSignal
    Dictionary keyNotFoundSignal
    Dictionary valueNotFoundSignal
  Object messageNotUnderstoodSignal
  Object subclassResponsibilitySignal
  ArithmeticValue errorSignal
    ArithmeticValue divisionByZeroSignal
    ArithmeticValue domainErrorSignal
  Stream positionOutOfBoundsSignal
  Controller badControllerSignal
  Object informationSignal
    Object notifySignal
    Stream endOfStreamSignal
  Object controlInterruptedSignal
    Object userInterruptSignal
```

For more obscure purposes, you can check the class methods of a relevant class to see whether it contains a useful signal. The following system classes, listed in alphabetical order, contain publicly accessible signals:

Table 8-2 *Publicly accessible signals*

ArithmeticValue	GraphicsContext
BinaryStorage	KeyboardEvent
ByteCodeStream	Metaclass
ByteEncodedString	Object
ClassBuilder	ObjectMemory
CodeStream	OSErrorHolder
ColorValue	Palette
CompiledCode	ParagraphEditor
Context	Process
Controller	Promise
ControlManager	Set
Dictionary	Signal
Exception	Stream
ExternalStream	UninterpretedBytes
FontPolicy	WeakArray

A `Signal` also has a *proceedability* attribute, which indicates whether the error is harmless enough to permit the process to proceed from that point onward. By default, a new signal inherits the proceedability setting of its parent signal. To establish a specific proceedability in a new signal, use `newSignalMayProceed:`, as in the following expression:

```
divSignal := (Number errorSignal) newSignalMayProceed: false
```

Creating an Exception

When an error such as zero division is perceived, an **Exception** object is created by sending a **raise** message to the appropriate signal. Thus, creating an exception is also called raising an exception. This object then travels back along the message stack looking for its matching signal (or an ancestor), triggering the intended **handle: block**. (In terms of our hawk-signal metaphor, the prey hunts for the predator.)

In the calculator example, the **Fraction** method that performs the division perceives that the denominator is zero. It sends a **raise** message to **DivisionByZeroSignal**, which creates an instance of **Exception**. This exception then traverses the chain of calling objects until it finds either **DivisionByZeroSignal** or a more general parent, such as **Number errorSignal**. (Remember that the error occurred in a **do: block** being executed by this signal, so the handler is located in the same place.)

The **raise** message effectively transfers control from the method in which the error was perceived to the **handle: block** in the calling method. A variant of **raise** permits control to proceed from the point of error (usually after the **handle: block** warns the user or corrects the cause, or both). To create a proceedable exception, use **raiseRequest** (the exception requests that control be returned to it). A proceedable exception can only be successfully addressed to a proceedable signal; a nonproceedable exception can be addressed to either type of signal. Thus, the exception largely determines its own proceedability.

Setting Parameters

An exception can carry an argument object back to the handler block, such as a value that can be used to diagnose the breakdown, an array of such values, or a block of remedial operations. The default is **nil**. To set that value, send a **parameter: message** to the exception, with the object as argument.

For situations in which the signal's notifier string needs to be replaced or augmented, send **errorString: to the exception**, with the replacement string as argument. If the first character of the argument string is a space, the argument is appended to the signal's notifier string. Otherwise, the argument string is used instead of the signal's string.

By default, an **Exception** begins its search for a handler in the context that sent the **raise** message. To substitute a different starting place, send a **search-From: message** to the **Exception**, with the starting-point context as argument.

Because more than one instance of the same **Signal** can exist, as implemented by different methods (with different handlers, possibly), an **Exception** can get fielded by the wrong handler unless it has a way to identify its originator. To do so, send `originator` to the **Exception**, with the object that originated the `raise` message as argument. To equip the handler with the originator, so it can spot the matching **Exception**, send a `handle:from:do:` message, supplying the originator as the argument to the `from:` keyword.

Passing Control From the Handler Block

A handler block can redirect the flow of control in one of four ways, listed in order of increasing assertiveness:

- n Refuse to handle the exception
- n Exit from the handler block and from the method in which it is located (i.e., a conventional return).
- n Proceed from the point at which the error occurred.
- n Restart the `do:` block and try it again.

To refuse control, use `reject`, as in `anException reject`. The exception will then continue its search for a receptive signal.

To exit from the handler block, use `return`. The `nil` object will be returned. To pass a value other than `nil`, use `returnWith:`.

To return control to the point at which the error occurred, use `proceed`. To pass an argument to be used as the value of the signal message, use `proceed-With:`. To proceed by raising a new exception—in effect, to substitute a different signal in place of the original error creator—use `proceedDoing:` and raise the new exception in the argument block.

To restart the `do:` block, use `restart`. To substitute another block of expressions for the original block, use `restartDo:`, as in the expression `theException restartDo: aBlock`.

If a handler does not choose one of the four options described here, it has the same effect as `theException returnWith:` the value of the block.

Raising a signal within its own handler does not restart the handler. However, raising a signal within a `proceedDoing:` or `restartDo:` block does invoke the signal's `handle` block again.

Returning to the calculator example, let's fill in the handler code:

```
ArithmeticValue divisionByZeroSignal
  handle: [:theException |
    Transcript cr; show: 'Enter a nonzero divisor'.
    theException restart]
  do: [the division operation]
```

Using Nested Signals

In some situations, it will be necessary to have more than one hawk watching the same process. For example, you might want to catch both numeric errors and dictionary errors, without using the full generality of a mutual parent such as `Object errorSignal`. To avoid nesting one `handle:do:` construct within another, create an instance of `SignalCollection`. A `SignalCollection` is created via `new` and an element is appended via `add:`, as with any `OrderedCollection`. Use `handle:do:` just as you would with an individual signal. When an exception is raised, it will try each signal in the collection until it comes to one that it recognizes.

A `SignalCollection` works fine when the same handler block is to be used no matter what kind of error crops up. But if each type of signal is the trigger for a different handler block, use a `HandlerList`. To create it, use `new`.

Each element of a `HandlerList` consists of a signal and an associated handler block. To add such an element, use `on:handle:`, as in `aHandlerList on: aSignal handle: aBlock`. To begin execution of the `do:` block, use `handleDo:`, as in `anHC handleDo: aBlock`.

A `HandlerList` can be built in advance and reused in various contexts, which is both more readable than the nesting approach and more efficient than building even a single handler on the spot. Bear in mind, however, that handlers in a `HandlerList` are not peers—they are effectively nested. A signal that is raised in a nested series will not be fielded by a handler that is lower in the hierarchy (or later in the collection). For example, the first set of expressions below is semantically equivalent to the second.

```
HandlerList new
  on: sg1 handle: [:ex | "response 1"];
  on: sg2 handle: [:ex | "response 2"];
  on: sg3 handle: [:ex | "response 3"];
  handleDo: ["Any arbitrary action"].
```

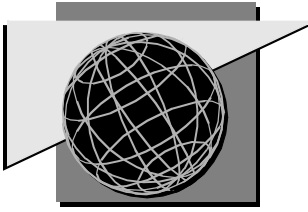
```
sg1 handle: [:ex | "response 1"]  
  do: [sg2 handle: [:ex | "response 2"]  
    do: [sg3 handle: [:ex | "response 3"]  
      do: ["Any arbitrary action"]]]].
```

Unwind Protection

When a block of expressions contains opportunities for a premature return, a means of cleaning up the mess may be required.

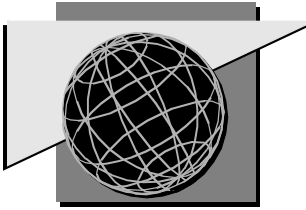
Providing such a mechanism is a kind of exception handling, though it is accomplished with a variant of the `value` message that initiates a block. Use `valueOnUnwindDo:`, with the cleanup expressions as the argument block. The cleanup block is used if the execution stack is cut back because of a signal, if a `return` is used to exit from the block, or if the process is terminated.

To execute the cleanup block after either a normal or an abnormal exit, use `valueNowOrOnUnwindDo:`. Remember that these messages are addressed to a block, not to a signal.



Part II

VisualWorks Tools



Chapter 9

Environment Tools

In the VisualWorks environment, tools are provided that enable you to easily control your working environment. These tools include:

- n VisualWorks main window
- n Settings
- n File List
- n Changes List
- n File Editor
- n Project

VisualWorks Main Window

The VisualWorks main window is a convenient device for opening tool windows. To open a tool, either click once on its icon (if the tool has an icon) or select it from the menu. This main window is the primary means of saving and quitting an image. The VisualWorks main window should not be closed.

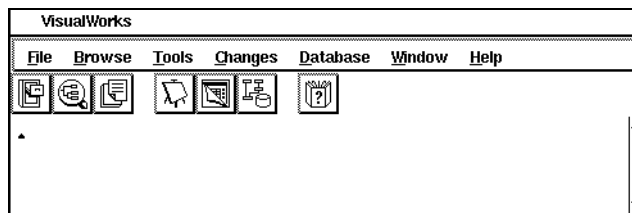


Figure 9-1 VisualWorks Main Window

The two most important items in the VisualWorks main window menu are for saving an image (**File?Save As...**) and quitting an image (**File?Exit VisualWorks...**). These are described fully on page 3.

Settings Tool

The Settings Tool controls a variety of global parameters such as the appearance of window decorations. Each customizable feature of the system is represented by a page in the Settings Tool. Use the tabs to navigate among the pages. Online help on each page of the Settings Tool will guide you in the proper setting of each parameter.

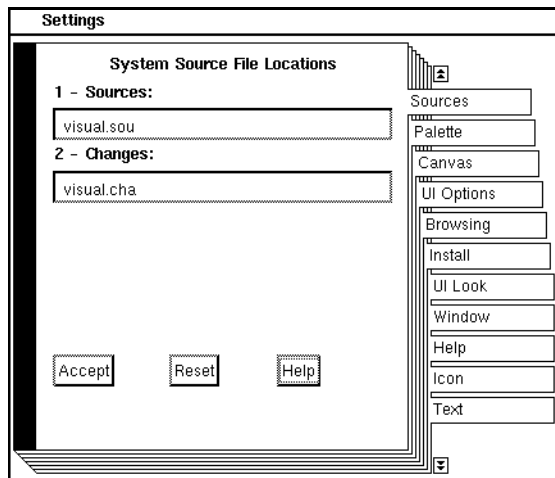


Figure 9-2 Settings Tool

User preferences include the following:

- n System source file locations
- n Settings for the VisualWorks palettes
- n Settings for painting tools
- n Settings that affect the user interface
- n Browsing
- n Canvas Installation
- n User interface look
- n Window placement
- n Options for message catalogs
- n Help options
- n Icon label length

- n Default Font, printing setup
- n Database Tools and user application defaults
- n Time zone

To open the Setting Tool, choose **File?Settings** in the VisualWorks main window.

File List

A File List is a special browser that interfaces with your operating system's file management facilities. With a File List, you can list the contents of any directory or file, edit a file, and create a new file.

VisualWorks provides two versions of the File List: the standard File List and an enhanced File List. The enhanced File List provides:

- n A menu bar for display options.
- n Support for globalization. See the *VisualWorks International User's Guide* for a description of these options.

To use the enhanced File List, turn on the **Use Enhanced Tools** switch on the **UI Options** page of the Settings Tool. Turn off the **Use Enhanced Tools** switch to use the standard File List.

To open a File List, choose **Tools?File List**, or click on the File List icon in the VisualWorks main window.

Figure 9-3 shows both the standard and enhanced File List.

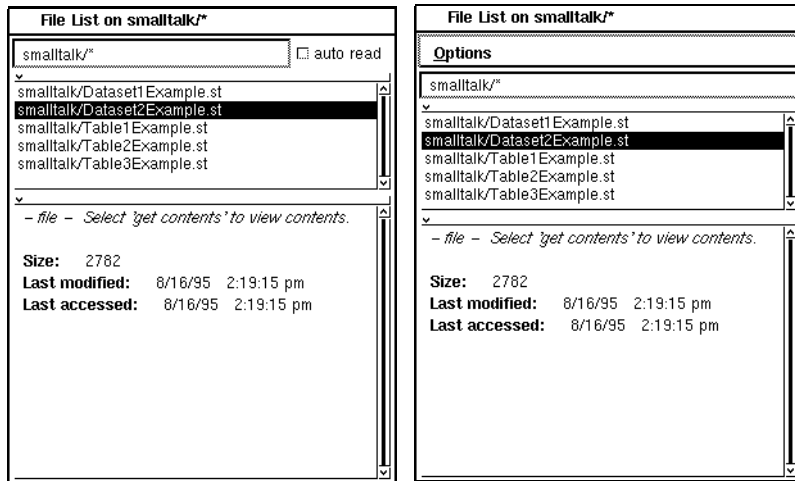


Figure 9-3 Standard and Enhanced File List

File List Views

The File List browser is divided into three views:

- n The pattern view (the top view) is for entering the pathname of a directory or file. Use an asterisk as a wildcard character. To create a new file, enter the full pathname and press <Return> or select **accept**.
- n The names view (the middle view) displays the directories and files that match the path view's search string.
- n The contents view (the bottom view) is a file editor with which you can modify the contents of the file and save the new version. The contents view also displays a list of files when a directory is selected in the file list view. When a file is first selected, the contents view displays the contents of the file, by default.

Display Options

To display the file characteristics as supplied by the operating system, select **get info** from the <Operate> menu in the names view. When no file or directory is selected in the names view, the number of entries in the list appears in the contents view.

You can control whether file contents or characteristics are displayed by default. To display the contents:

- n On the standard File List, turn the **auto read** switch on; to display file characteristics by default, turn it off.
- n On the enhanced File List, turn on the switch by choosing **Options?Auto Read**.

The `FileBrowser` class maintains the maximum size (50 KB, by default) of an auto-readable file; if a file exceeds that size, you will be given a choice between reading the contents (subject to a delay if the file is very large) or getting the file characteristics. To change the maximum size of an auto-readable file, modify the `initialize` method in the `FileBrowser` class.

To change the default pathname in the path input window, execute the following Smalltalk expression, substituting the desired *pathname*:

```
FileBrowser defaultPattern: 'pathname'.
```

File List Commands

Most of the menu commands within the File List have the same names and meanings as defined for the System Browser (**again**, **undo**, **copy**, **cut**, **paste**, etc.) The commands listed in Table 9-1, Table 9-2, and Table 9-3 only includes those that have not been defined previously. Use the <Operate> menu in each view to display the commands.

Table 9-1 lists the pattern view command.

Table 9-1 Pattern View Command

Command	Description
volumes...	Display a pop-up menu of disk volumes, so you can select one as the starting point for the pathname entry.

Table 9-2 lists the names view commands.

Table 9-2 Names View Commands

Command	Description
new pattern	Make the currently selected directory the entry in the pattern view, appending a trailing separator and asterisk, if the selection is a directory.
add directory...	Prompt for the name of a subdirectory to be created under the currently selected directory.
add file...	Prompt for the name of a file to be created in the currently selected directory.
get info	Display the file characteristics as supplied by the operating system.
get contents	Display the contents of the selected file in the contents view.
file in	Compile the contents of the selected file into the current image—the file is presumed to contain Smalltalk expressions that define classes and/or methods.
copy name	Copy the current selection into the VisualWorks paste buffer, so it can be pasted into the path input view (or elsewhere).
rename as...	Change the name of the directory or file.
copy file to...	Prompt for a pathname and save a copy of the selected file under the new name.
remove...	Prompt for confirmation, then delete the file or directory.
spawn	If a directory is selected, open a new File List with that directory as the default search string. If a file is selected, open a new file editor on that file.

Table 9-3 lists the contents view commands.

Table 9-3 Contents View Commands

Command	Description
file it in	Execute the selected text, which is assumed to be in the format created by the file out command, with exclamation points as delimiters.
save	Save the (edited) contents of the text view in the file that is selected in the file list view.
save as...	Prompt for the name of a new file and save the contents of the text view in a file with that name. The file cannot be saved to a different disk volume.
cancel	Replace the current contents of the text editing view with the contents of the disk file that is selected in the file list view.

Change List

VisualWorks keeps a running list of changes that are made to the image. The Change List enables you to view the changes made since the last time the image was saved, and reload the changes selectively.

The Change List is also useful for browsing a file-in containing Smalltalk code. See “Managing Projects and Versions,” for more details about using a

Change List and the **Changes** submenu of the VisualWorks main window in the list view.

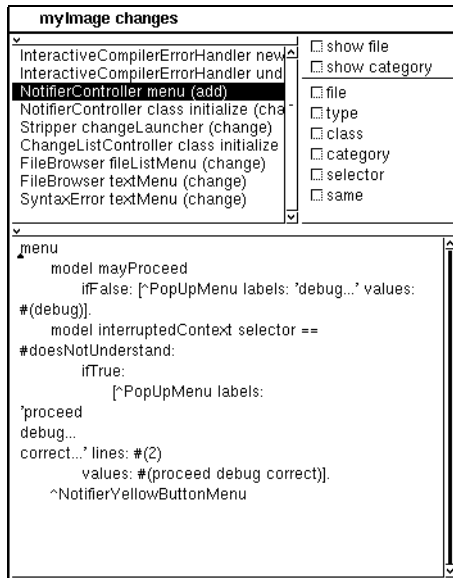


Figure 9-4 Change List

The Change List window has three views. The view at top left is for displaying a list of the changes. The top right-hand view provides on/off switches for filtering the contents of the change list.

The Change List's filter switches are described below, as well as the command menu for the list view. The commands for the code browser view are defined in Table 10-5, "Code View Commands," on page 123.

The first two switches (above the divider line) affect the format of each entry in the list. The remaining six switches control which entries are displayed. Any combination of filter switches can be selected.

Table 9-4 Change List Switches

Command	Description
Show file	For each entry in the list, precede it by the name of the file in which it is recorded.
Show category	For each entry in the list, precede it by the name of the class category (for a change to a class) or message category (for a change to a method) in which the change occurred.
file	Display only entries that are recorded in the same disk file as the selected entry.
type	Display only entries of the same type as the selected entry (for example, all do Its, which represent executed expressions).
class	Display only entries that affect the same class as the selected entry.
category	Display only entries that affect a class or message in the same category as the selected entry.
selector	Display only entries that involve the same message selector as the selected entry.
same	Display only entries that have the same type as the selected entry and affect the same class or method. The entries in the change list are identical, though the underlying code may be different for each entry.

Use the <Operate> menu to display the commands listed in Table 9-5.

Table 9-5 Change List Commands

Command	Description
file in/out?read file/directory	Prompt for a pathname. Add the Smalltalk expressions contained in the designated file to the list being displayed. If a directory pathname is supplied, add the contents of all files (presumed to be Smalltalk code) in the directory to the list.
file in/out?write file	Prompt for a filename. Store the code indicated by the displayed change entries in that file. The changes file thus created can be used to transfer system changes to another image.
file in/out?recover last changes	Display all changes made to the system since the last snapshot was made (i.e., the last save operation).
file in/out? display system changes	Add the contents of the Change Set to the displayed list of changes.
replay all	Execute every change that is displayed (and not marked for removal).
remove all	Mark all displayed entries for removal. Use forget to delete marked entries.
restore all	Unmark all displayed entries so they won't be affected by a forget operation.
spawn all...	Open a new Change List containing only the displayed entries (i.e., with all filter switches set to off but retaining the effect of the filters).
forget	Delete every entry that has been marked for removal by a remove operation.
replay selection	Execute the selected change entry.
remove selection	Mark the selected entry for deletion.
restore selection	Unmark the selected entry.

Table 9-5 Change List Commands

Command	Description
spawn selection...	If the selected entry involves a change to a method, open a method browser on the version of that method that is currently in use. Otherwise, do nothing.
conflicts?check conflicts, save as...	Prompt for an output filename. Find each change entry that affects the same class or method as another change entry, then print all versions of the affected code in the designated file. This is useful when you are integrating code from multiple files and you want to check for overlaps—places where one file changes the same method as another file.
conflicts?check with system, save as...	Prompt for an output filename. Find change entries in which the code differs from the system's current version and print both versions of the affected code in the designated file.

File Editor

The File Editor is a stand-alone version of the file-editing view in the File List. It provides a rapid, two-step means of opening an editor on a particular file, as follows:

1. Choose **Tools?File Editor...** in the VisualWorks main window.
A prompter asks for the name of the file to be edited. The name defaults to the contents of the **paste** buffer.
2. Enter the name of the file into the input field and click **OK**.

The <Operate> menu is identical to that of the File List's file editing view.

Project

You can create a separate Project to contain the views and change set associated with an aspect of your work. Such Projects can be nested, for a hierarchical organization. When you enter a Project, only the windows that you have opened in that Project are displayed (initially, only the VisualWorks main window).

To create a new Project, choose **Changes?Open Project** in the VisualWorks main window. A Project window will appear, consisting of an **enter**

button at the top and a text-editing view for recording a description of the project.

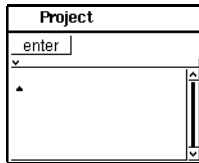
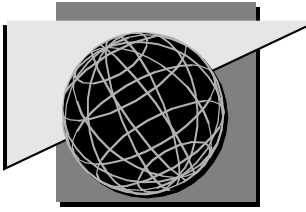


Figure 9-5 Project

To temporarily erase the current Project's windows and display the windows associated with a different Project, select the **enter** button in the desired Project's window. To exit from a project and redisplay the parent Project's windows, choose **Changes?Exit Project** in the VisualWorks main window.



Chapter 10

Smalltalk Programming Tools

This chapter discusses the major programming tools available in the VisualWorks environment:

- n System Browser
- n Workspace
- n System Transcript
- n Debugger
- n Inspector

System Browser

VisualWorks' principal programming tool is the System Browser. Its capabilities include not only "browsing" the code library, as its name suggests, but editing, compiling and printing any selected portion of it.

To open a new System Browser window (more than one can be open at a time), choose **Browse?All Classes** or click on the System Browser icon in the VisualWorks main window.

Structure

The System Browser has four upper views and one lower view, as shown in Figure 10-1. Each view provides a lower level of detail in the code library, ending in the *code subview* (5), which deals with a single method (the smallest unit of code in a Smalltalk program).

Class Categories

There are more than 700 classes of objects in the system, so they are placed in functional groups called class categories. In the illustration, the **Magnitude-General** category is selected.

Classes

The second view displays all of the classes in the currently selected category. The **Magnitude-General** category has five classes, as shown. The **Time** class is selected.

Protocols

A single class, such as **Time**, can respond to any number of messages. For the sake of convenience and conceptual clarity, they are placed in functional groups called protocols. In the third view, all of the protocols for the currently selected class are displayed. For **Time**, there are three categories of class messages. The **instance creation** category is selected.

Methods

The fourth view displays all of the method names in the currently selected protocol. The **instance creation** protocol contains three methods. The **now** method name is selected.

Code

The bottom view is not a list view like the other four. It is a special text editor that enables you to perform the full range of programming operations on the selected method, from editing to compiling. It is used to define classes as well as their methods, so it is called a *code* view.

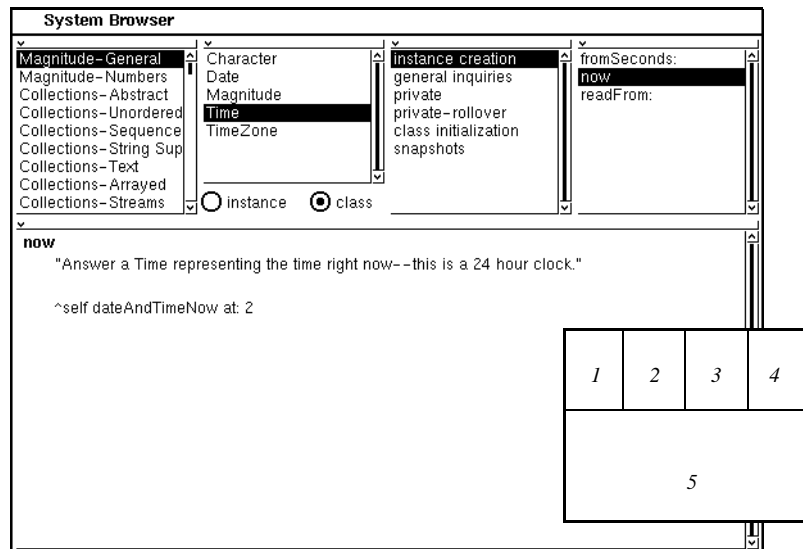


Figure 10-1 System Browser

Notice that only views 2, 4 and 5 deal with actual system objects (classes, methods, and code). Views 1 and 3 (class categories and protocols) are organizational constructs provided for your convenience.

The System Browser also has a toggle switch, represented by the two buttons at the bottom of the class view (**instance** and **class**). Each class has two kinds of methods: class methods and instance methods. To see the class protocols and method names in views 3 and 4, select the **class** button. To see the instance protocols and methods, select **instance**.

Each view has its own scroll bar at the right-hand edge. The <Operate> menu is different for each of the views, and changes depending on whether an item is selected in the view.

Each view in the System Browser has a unique menu, offering commands that are appropriate to its contents. The commands for each view are presented here, listed in the order in which they appear in the menu. In some views, the menu has fewer options when no item in the view has been selected.

Table 10-1 lists the class category view commands.

Table 10-1 Class Category View Commands

Command	Description
file out as	Prompt for a filename. Store a description of each class in the selected category, in a form that enables the class (including all of its methods) to be placed in another VisualWorks image with the file in command.
hardcopy	Print a hard-copy description of each class in the selected category.
spawn	Open a new browser on just the classes in the selected category (a Category Browser).
add	Prompt for a category name. Add the new category immediately above the currently selected category (if one is selected) or at the bottom of the list.
rename as	Prompt for a new name, then replace the currently selected category name, both in the list and in all class definitions under that category. An existing category name cannot be duplicated.
remove	Delete the currently selected category. If the category still contains one or more classes, confirm, then delete the classes also. References to the classes remain intact.
update	Bring the category listing up to date (after filing in a new category or adding one in another browser).
edit all	Display the entire category/class organization of the system in the code view (number 5). This permits you to rearrange the order of the categories, using the code view's editing facilities. Use the code view's accept command to place the changes in the system.
find class	Prompt for a class name, then select that class in the class view (and its category, in the category view). If a wildcard character is used, display a dialog of all classes with matching names. For example, the pattern C*View displays a list of all classes beginning with "C" and ending in "View."

Table 10-2 lists the class view commands.

Table 10-2 Class View Commands

Command	Description
file out as	Prompt for a filename. Store a description of the currently selected class, in a form that enables the class (including all of its methods) to be placed in another VisualWorks image with the file in command.
hardcopy	Print a hard-copy description of the class.
spawn	Open a new browser on the class.
spawn hierarchy	Open a new browser on the currently selected class and its superclasses and subclasses.
hierarchy	In the code view, display the names of the currently selected class, its superclasses and its subclasses, with indentations to indicate hierarchic precedence.
definition	In the code view, display the formal definition of the currently selected class. To change the definition, edit the text and use the accept command.
comment	In the code view, display the class comment. To change the comment, edit the text and use the accept command.
inst var refs	Display a dialog that lists all of the instance variables of the currently selected class and its superclasses. Select a variable name to open a browser on all methods that refer to that variable.
class var refs	Open a browser on methods that refer to a selected class variable.
class refs	Open a browser on all methods that refer to the currently selected class.

Table 10-2 Class View Commands

Command	Description
move to	Prompt for a category name (new or existing). Move the class to that category and update the System Browser.
rename as	Prompt for a new class name. Replace the name in the system dictionary and update the class view. (Use the class refs command to find methods that refer to the old class name—then substitute the new name manually.)
remove	Prompt for confirmation, then delete the selected class and its methods. (Note: It's much easier to find references to the class and its methods before you delete, using the class refs commands.)

Table 10-3 lists the protocol view commands.

Table 10-3 Protocol View Commands

Command	Description
file out as	Prompt for a filename. Store a description of the methods in the currently selected message category, in a form that enables the methods to be placed in another VisualWorks image with the file in command.
hardcopy	Print a hard-copy description of the methods in the currently selected message category.
spawn	Open a new browser (a message category browser) on the methods in the currently selected message category.
add	Prompt for the name of a new message category, then add that name in the message category view. Add the new category immediately above the currently selected category (if one is selected) or at the bottom of the list.
rename as	Prompt for a new name, then update the message category view.

Table 10-3 Protocol View Commands

Command	Description
remove	Prompt for confirmation, then delete the selected message category and its methods. (Note: It's much easier to find references to the methods before you delete, using the senders command listed below.)
edit all	In the code view, display a list of all message categories, and method names in each category. To change the order of the protocol names in the message category view, or to move methods from one category to another, edit the text and use the accept command.
find method	Display a dialog that lists all of the instance methods (if the instance switch is selected) or class methods of the currently selected class. Select a method name to display its code in the code view.

Table 10-4 lists the method view commands.

Table 10-4 Method View Commands

Command	Description
file out as	Prompt for a filename. Store a description of the currently selected method, in a form that enables the method to be placed in another VisualWorks image with the file in command.
hardcopy	Print a hard-copy description of the method.
spawn	Open a new browser (a Method Browser) on the currently selected method.
senders	Open a new browser (a Method Browser) on all methods that send the currently selected message.
implementors	Open a browser (a Method Browser) on all methods that implement the currently selected message (i.e., methods having the same name that exist in other classes as well as this one).

Table 10-4 Method View Commands

Command	Description
messages	Display a dialog of all method selectors that exist in the currently selected method. After one is selected, open a browser (a Method Browser) on all methods that implement that message.
move to	Prompt for the name of the message category to which the currently selected method is to be relocated. If the category doesn't exist, it will be created. If the destination is another class, include both the class name and the message category, as in "Customer>accessing". In the latter case, the method will be copied rather than relocated.
remove	Delete the currently selected method. (Note: It's much easier to find references to a method before you delete it, using the senders command listed above.)

Table 10-5 lists the code view commands.

Table 10-5 Code View Commands

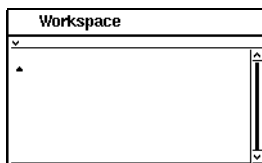
Command	Description
undo	Reverse the most recent cut or paste.
copy	Place a copy of the highlighted text in memory. If <Shift> is held down while copy is selected, the text is copied to the window manager's clipboard.
cut	Place a copy of the highlighted text in the paste buffer, then delete the original.
paste	Delete the highlighted text (if any), then place a copy of the most recently copied or cut selection in that location. If <Shift> is held down while paste is selected, a dialog presents the most recent five text segments that have been copied or cut, including the window manager's clipboard.
do it	Execute the highlighted text as a Smalltalk expression. The scope of execution is the selected class, so class variables can be used in the expressions, and self refers to the selected class.
print it	Same as do it , except a description of the resulting object is inserted in the text. The printed string becomes the current selection, so it can be deleted easily.

Table 10-5 Code View Commands

Command	Description
inspect	Same as do it , except an inspector is opened on the resulting object.
accept	Compile the code and, assuming no errors are found, store it.
cancel	Restore the entire text to its condition when it was last compiled (with accept).
format	Impose standard font characteristics and indentation conventions on the code.
spawn	Open a new browser (a Method Browser) on the method. This is useful when you want to preserve the original and make changes in the new copy. Whichever version is compiled last, via accept , takes effect for the entire image.
explain	Insert an explanation of the selected literal or variable. The explanation frequently contains a Smalltalk expression that you can execute to get more details.
hardcopy	Print a copy of the text or code on paper.

Workspace

A Workspace is like a free-floating code view, or a scratch pad with a pipeline to the compiler. It is a blank window in which you can test Smalltalk code before building it into the code library. To open a Workspace, choose **Tools?Workspace** or click on the Workspace icon in the VisualWorks main window.

*Figure 10-2 Workspace*

The Workspace's <Operate> menu gives it much the same functionality as the code view of the System Browser. Use the Workspace's <Operate> menu to edit text and execute expressions.

More than one Workspace can be open at a time.

System Transcript

The System Transcript, by default, displays in the VisualWorks main window. It shows a running list of informational messages generated by VisualWorks or your code. Error messages, on the other hand, are generally displayed in a pop-up window called a Notifier. To close a System Transcript, select **Tools** in the VisualWorks main window; in the submenu, select **System Transcript**.

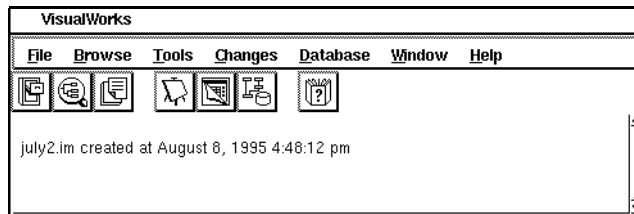


Figure 10-3 VisualWorks main window and associated System Transcript

Each time you save your image, the System Transcript records the date, time and name of the newly created image. When you file out a class category, the System Transcript records the name of each class as it is processed. You can also use the System Transcript to display messages, which is especially helpful during the debugging phase of a project. For example, the following expression could be inserted in a method to display the value of the variable named `account`, followed by a carriage return:

```
Transcript show: account printString.  
Transcript cr.
```

To avoid an update of the display with each part of a larger message, use `nextPutAll:` or `print:`, then use `endEntry` to output the message, as in:

```
Transcript  
    nextPutAll: 'The account is: '  
    print: account;  
    endEntry.
```

`Transcript` is a global variable, and refers to an instance of the class `TextCollector`, where you will find other useful `Transcript` behavior.

The System Transcript's <Operate> menu gives it much the same functionality as the code view of the System Browser. The menu contains commands for editing text and executing expressions.

Debugger

When a program error occurs, a notifier window appears. This notifier displays the last five message-sends in the context stack. The context stack lists message-sends that were waiting for a return when the breakdown occurred. Sometimes that listing of the context stack is sufficient for you to identify the problem and correct it. If so, choose **close** in the <Window> menu, or click the **Terminate** button to close the notifier.

When the error is not so serious as to prevent proceeding with the program (i.e., it is a warning), you can click **Proceed** in the notifier. The notifier will be closed and the program will continue.

When you need to examine the conditions that led to the failure more closely, click **Debug** in the notifier. The notifier will be replaced by a debugger, which enables you to trace the program flow leading to the error, proceed with execution step by step, and examine the operative method and the values of the variables at each stage of execution.

The debugging window consists of four component views, as shown in Figure 10-4: a stack view (1), a code view (2), and two inspectors (3 and 4). The inspectors each have two subviews, so technically the debugger has *six* views. It also has two buttons, **step** and **send**, located between views 1 and 2.

The stack view (1) is similar to the error notifier in that it lists the message-sends that were waiting for a return at the time of the error. The stack view's commands, described below, permit you to proceed with the program's execution at the desired pace, and to expand the depth of the listing.

The code view (2) is similar to the System Browser's code view (see page 122) and has the same <Operate> menu. When a message-send is highlighted in the stack view, the corresponding method is displayed in the code view. Within the method, the current point of execution is automatically highlighted by the debugger.

The instance-variable inspector (3) and the temporary-variable inspector (4) allow you to examine the values of the variables. The variables and their values are updated each time you choose a different position in the execution

stack with the stack view. For the operations and commands associated with an inspector, see page 128.

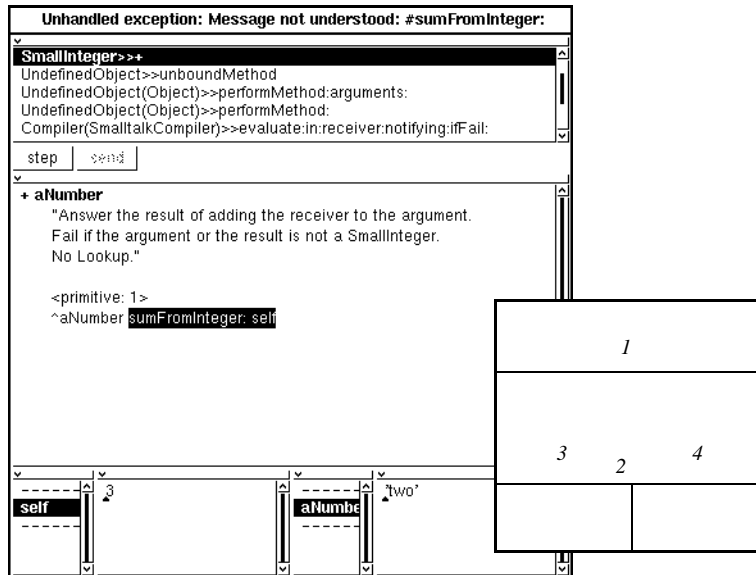


Figure 10-4 Debugger

For detailed instructions about using a debugger, see Chapter 14.

Table 10-6 Stack View Commands

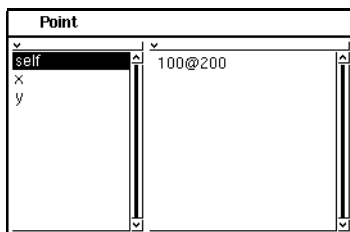
Command	Description
more stack	Double the number of message-sends displayed in the stack view, effectively reaching twice as far back into the history preceding the error. When the entire stack is listed already, this command disappears from the menu.
proceed	Close the debugger and continue program execution in the currently selected context (in the top context, if none is selected). Execution proceeds as if the interrupted message had completed.
copy stack	Copy the textual contents of the stack view to the paste buffer (so you can paste the stack text into a file).

Table 10-6 (Continued) Stack View Commands

Command	Description
restart	Close the debugger and restart execution from the beginning of the currently selected method.
senders	Open a browser on all methods that send the selected message.
implementors	Open a browser on all methods that implement the selected method.
messages	Display a dialog containing all message selectors in the selected method. Choose one to open a browser on all methods that implement that message.
skip to caret	Continue program execution to the location in the method marked by the cursor caret.
step	Execute the next message-send in the currently selected method (or in the top context in the stack, if none is selected). Halt after a value is returned.
send	Same as step, but halt in the method that receives the message-send.

Inspector

An inspector is a window that is used to examine the values of the variables of an object. In the simplest inspector, containing two views, the variables are listed in the left-hand view. When you select one such variable, its value appears in the right-hand view.

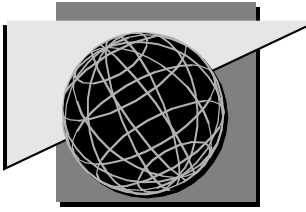
*Figure 10-5 Inspector*

If the value is a composite object, such as a collection, you can open a new inspector that exposes a component object by choosing **inspect** in the list view.

The right-hand view is a code view, in which you can type and execute Small-talk expressions. Instance and class variables are within the scope of the code view. In the example illustrated at left, if you type $x + y$ in the code view and then execute it, **300** is returned

Inspectors are built into the debugger window and other system tools. In a code view, you can open an inspector on a selected variable or literal by using **inspect** in the <Operate> menu.

Specialized inspectors provide extended inspecting capabilities for dictionary objects, collections, and Model-View-Controller triads. The type of inspector is automatically matched to the object type.



Chapter 11

Application Building Tools

VisualWorks provides a variety of tools for building applications within the Smalltalk programming environment. In this section, we briefly describe the following tools:

- n Resource Finder
- n Painting (Canvas and Palette tools)
- n Properties
- n Image Editor
- n Menu Editor

Resource Finder

The Resource Finder is for navigating among resources. Application classes are listed in the left-hand view. You can filter the list to show tool classes, example classes, and other categories. To do so, use the **View?All Classes** submenu. When you add or remove an application class by some means other

than the Resource Finder or the canvas's **install** command, use **View?Update** to register the change in all Finders.

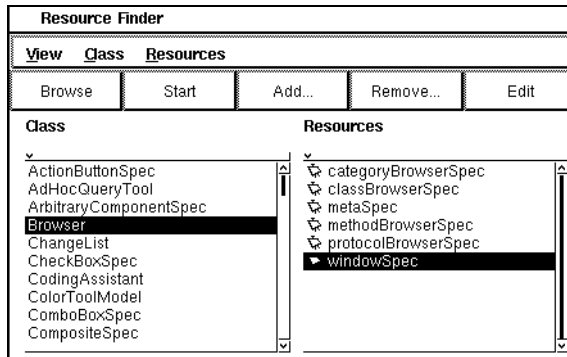


Figure 11-1 Resource Finder

An application class can support multiple interfaces, each of which may involve multiple canvases for the VisualWorks main window, secondary windows and dialogs. The right-hand view lists all of the resource methods for the selected class.

The buttons below the menu options of the Resource Finder enable you to **Browse** the code for a selected class, **Start** the main interface (by sending open to the class), **Add** a blank resource to a new or existing class, **Remove** a resource, and **Edit** a resource.

Note: Double-clicking on a class name brings up a Hierarchy Browser or Class Browser (depending on your Settings preferences) on that class.

Canvas Tool

The Canvas Tool enables you to control the following features of a canvas:

- n Whether the alignment grid is on, and its attributes.
- n Whether the fence is on, preventing objects from being accidentally dragged beyond the edges of the canvas.
- n Which look to use while editing the canvas.

Additionally, the Canvas Tool combines the most frequently used operations from the canvas's menu as well as providing easy access to the Properties Tool, which is used to customize each type of component.

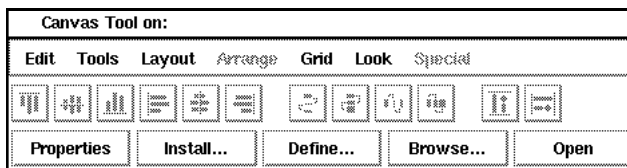


Figure 11-2 Canvas Tool

The left-most six buttons located directly under the menu control the horizontal and vertical alignment of the selected components. To the right are the Equalize and Distribute buttons, which operate along the vertical or horizontal dimension.

Palette

The Palette is the main companion tool for a canvas, in that it supplies the interface components for the canvas. The name “Palette” derives from the metaphor of *painting* canvases that describe your application windows.

By default, a Palette is opened automatically with each canvas, though you can arrange for manual Palette opening in the Settings tool. To open a Palette manually, select **tools?palette** in the canvas’ <Operate> menu.



Figure 11-3 VisualWorks Palette

The Palette has one button for each type of interface component. To add an input field to your canvas, for example, you simply click on the **Input Field** icon in the Palette and then click in the canvas to place the field there.

The **repeat** switch (at upper right) is used to place the Palette in repeat mode—the selected type of component remains selected even after you place one on the canvas. This is useful, for example, when you want to place multiple labels on a canvas without having to reselect the label button each time. A shortcut for putting the Palette in repeat mode is to hold down a <Shift> key while selecting the type of component.

When you open a Palette from the canvas's menu, the Palette closes and collapses with the canvas window. When you open a Palette via the VisualWorks main window, it remains open unless you explicitly close it. Any Palette can be used with any canvas—all Palettes are in sync.

Image Editor

The Image Editor is used to create and modify illustrations, with pixel-level control. The resulting graphic can be integrated into a variety of components.

To open an Image Editor, choose **Tools?Image Editor** from the VisualWorks main window.

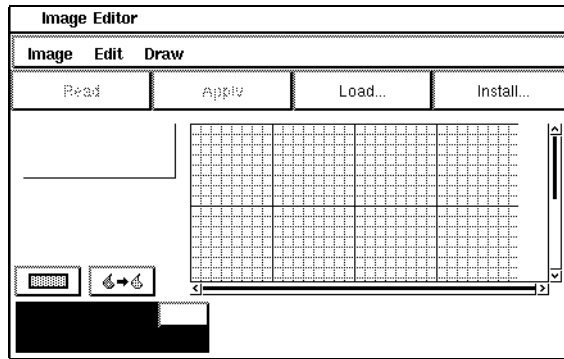


Figure 11-4 Image Editor

Menu Editor

The Menu Editor is used to create and edit menus, which can be integrated into a variety of components. You use the Menu Editor to create textual entries and specify attributes for desired menu items. The Menu Editor uses these entries to generate a specification for building an appropriate menu object. This code is then installed in a method in the application model.

You can use the Menu Editor to create a menu for any widget that provides a menu, such as a menu button.

VisualWorks provides two versions of the Menu Editor: the standard Menu Editor and an enhanced Menu Editor.

The standard Menu Editor allows you to build simple menus. To use the standard Menu Editor, turn off the **Use Enhanced Tools** switch on the **UI**

Options page of the Settings Tool. Then, choose **Tools?Menu Editor** from the VisualWorks main menu to open the standard Menu Editor.



Figure 11-5 Standard Menu Editor

Most of the buttons on the standard Menu Editor correspond to a menu command in the enhanced Menu Editor. See “Menu Commands” on page 137 for explanations of those buttons.

Enhanced Menu Editor

The enhanced Menu Editor provides all the capabilities of the standard Menu Editor, plus:

- n A notebook for displaying and editing Menu properties.
- n Support for globalization. See the *VisualWorks International User’s Guide* for a description of these options.

To use the enhanced Menu Editor, turn on the **Use Enhanced Tools** switch on the **UI Options** page of the Settings Tool. Then, choose **Tools?Menu**

Editor from the VisualWorks main window to open the enhanced Menu Editor.

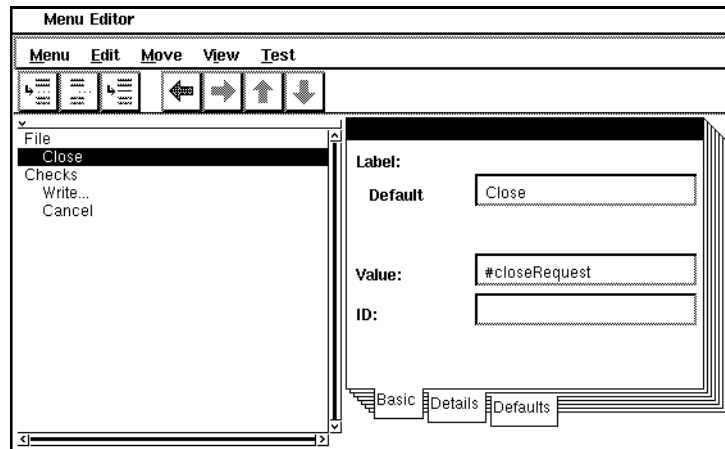


Figure 11-6 Enhanced Menu Editor

Menu Commands

Table 11-1 lists the **Menu** commands for the enhanced Menu Editor.

Table 11-1 Menu Editor Menu Commands

Command	Description
New	Clears the Menu Editor of the current menu and prepares for a new menu.
Load...	Loads a menu from a specified class and selector into the Menu Editor.
Install...	Prompts for the method selector and class name where menu specifications are stored.
Read	Edits a menu that has been applied to a selected widget.
Apply...	Applies the menu in the Menu Editor to an associated canvas.
Exit	Quits the Menu Editor.

Table 11-2 lists the **Edit** commands for the enhanced Menu Editor.

Table 11-2 Menu Editor Edit Commands

Command	Description
New Item	Creates a new top-level menu item. If a sub-level menu item is selected, the Menu Editor creates a new item at the selected level.
New Submenu Item	Creates a new submenu item.
Add Line	Inserts a divider line between two menu items.
Cut	Deletes the selected menu item and places it into the paste buffer.
Copy	Copies the selected menu item into the paste buffer.
Paste	Places a cut or copied menu item after the selected menu item.
Delete	Deletes the selected menu item.

Table 11-3 lists the **Move** commands for the enhanced Menu Editor.

Table 11-3 Menu Editor Move Commands

Command	Description
Up	Moves the selected menu item (and all its submenus) one level up in the hierarchy of the menu.
Down	Moves the selected menu item (and all its submenus) one level down in the hierarchy of the menu.
Right	Causes the selected menu item to become a submenu item of the item above it.
Left	Causes the selected menu item to move out one level in a hierarchical menu structure.

View?Sample Menu Bar opens a window that contains a test version of the menu bar that is currently entered into the Menu Editor.

The **Test** menu tests the appearance of the menu currently entered into the Menu Editor.

Properties

Table 11-4 lists the Basic properties for the enhanced Menu Editor.

Table 11-4 Basic Properties for Menu Editor

Property	Description
Label (Default)	Text of the menu item's label. Placing a & before a character creates a mnemonic key for that menu item. Mnemonic keys allow menu actions to be executed by keyboard keystrokes. To execute a mnemonic, press and hold the <Alt> key while pressing the mnemonic key.
Value	Name of the method that will perform the menu item's action.
ID	An identification for programmatic manipulation of the menu item. It specifies a Smalltalk symbol that you can use to reference the menu item programmatically

Table 11-5 lists the Details properties for the enhanced Menu Editor.

Table 11-5 Details Properties for Menu Editor

Property	Description
Shortcut character	The keyboard shortcut (or accelerator key) for the specified menu item. Press and hold the <Alt> key while pressing the shortcut character to execute the menu item command.
Label image	The graphic image used in place of, or in combination with a textual label for the menu item.

Table 11-6 lists the Defaults properties for the enhanced Menu Editor.

Table 11-6 Defaults Properties for Menu Editor

Property	Description
On/Off indicator	Prefixes a check box as a toggle indicator to the menu item if Initially on or Initially off is selected.
Initially enabled	Enables or disables the menu item.
Initially hidden	Visually removes the menu item when this switch is turned on.

Properties Tool

The Properties Tool is used to control various attributes of each component on a canvas. The set of attributes varies with each type of component—for example, an input field can have a menu while a check box cannot.

After you have painted a component, you can use the Properties Tool to modify its properties to suit your application's requirement. You can then use the **define** dialog to generate some and possibly all of the supporting code.

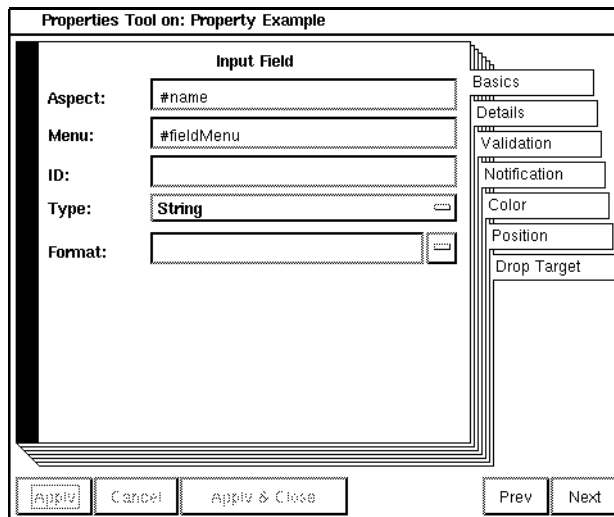


Figure 11-7 Properties Tool

The primary attribute for nearly all components is the **Aspect**, which is the name of the method that the component uses to fetch its value holder from the application model.

The properties can be edited either with a dialog (via **properties** in the canvas's <Operate> menu) or with a persistent tool (clicking on the **Properties** button on the Canvas Tool).

Use **Apply** or press <Return> to apply the properties to the component. The **Prev** and **Next** buttons (in the persistent tool) can be used to move the tool's focus to another component in the canvas, saving you the trouble of shifting the focus via the mouse.

Basics Properties

The properties listed in Table 11-7 are basic to most widgets.

Table 11-7 Basics Properties

Property	Description
Label	Specifies either the text that identifies the widget and forms a part of it, or the name of a method that supplies a graphic (provided that the Label Is Image property is also selected).
Label Is Image	Specifies that the widget is to be identified by a graphic used in place of a textual label. The method that supplies the graphic must be specified in the Label property.
ID	Specifies a Smalltalk symbol that you can use to reference the widget programmatically, e.g., by sending messages to the builder. Each running application has an object called a builder that assembles and opens the live user interface from the canvas specification. The builder does this by instantiating and keeping track of appropriate widget objects. You can use a widget's ID property to specify the symbolic name that the builder will use to reference the widget object while the application is running. You can then access the widget object programmatically by asking the application model for its builder, and then asking the builder for <code>componentAt: #idSymbol</code> .

Details Properties

Details properties listed in Table 11-8 are available on most widgets.

Table 11-8 Details Properties

Property	Description
Font	Specifies the font to be used for text in the widget's label or for any other text displayed by the widget. You can choose: <ul style="list-style-type: none">n System, which provides the font that matches the current platform's system font, if there is one.n Default, which provides the font that is currently selected on the Text page of the Settings tool (the default selection for the Settings tool is a medium-sized font).n Large, which provides a font that is slightly larger than the default.n Small, which provides a font that is slightly smaller than the default; Fixed, which provides a fixed-width font (useful for aligning text in columns).
Bordered	Specifies whether the widget is to be surrounded by a solid box.
Opaque	Specifies that the widget obscures any portion of any other widget it overlaps. This property is best used with passive widgets (such as regions, group boxes, dividers, etc.) for artistic effect. If you put an opaque widget on top of an active widget (such as a text editor, button, list), user interactions with the active widget may cause it to redraw itself on top of the opaque widget.
Can Tab	Specifies that the user can transfer focus to the widget by tabbing. When a widget has focus, any keyboard input is directed to it.
Initially Disabled	Specifies that the widget is disabled when the interface is opened, overriding the focus policy. You must program the application model to reenable the widget.
Initially Invisible	Specifies that the widget is invisible when the interface is opened. You must program the application model to make the widget visible.

Validation Properties

Validation properties are specified when you want a widget to ask its application model for permission to proceed with certain actions, namely, accepting focus, changing internal state, or giving up focus. Validation properties are useful for providing input flow control—for example, to prevent the user from entering invalid data into an input field, or to prevent the user from entering a field before filling in other prerequisite fields.

Each validation property specifies the symbolic name of a *validation callback*, which is the message you want the widget to send while preparing for the relevant action. For each validation callback specified, you must program the application model to contain a corresponding method that returns a boolean value. When the method returns true, the widget proceeds with its action; otherwise, the widget waits for new user input so it can send the callback again. You can implement the validation method to redirect input focus or to disable and enable input widgets.

If you want a validation method to inspect the widget's value, you specify the callback name with a colon (**selector:**). This tells the widget to pass its controller object as an argument to the method. The method can then ask the controller for the widget's value. For certain widgets (input fields, text editors, and combo boxes), you use statements such as the following to get and set values through the controller:

```
input := aController editValue
```

Table 11-9 Validation Properties

Property	Description
Entry	Specifies the symbolic name for the widget's entry validation callback. The widget sends this message to its application model when it prepares to take focus (for example, when the widget is tabbed into or selected with the mouse) If the method returns true, the widget takes focus; otherwise, focus is refused.

Table 11-9 Validation Properties

Property	Description
Change	Specifies the symbolic name for the widget's change validation callback. The widget sends this message to its application model after the user changes the widget's value and attempts to exit the widget (presses return, tabs, clicks on another widget) before the widget writes the input value to its value model. The corresponding method in the application model should determine whether the pending input value is acceptable. If the method returns true, the widget's controller writes the input value to the value model; otherwise, the value model remains unchanged.
Exit	Specifies the symbolic name for the widget's exit validation callback. The widget sends this message to its application model when it prepares to give up focus. The message is sent when the user attempts to exit the widget (presses return, tabs, clicks on another widget). The corresponding method in the application model should determine whether the widget can actually give up focus. If the method returns true, the widget gives up focus; otherwise, focus is retained.
D. Click	Specifies the symbolic name for the widget's double-click validation callback. The widget sends this message to its application model when preparing to respond to a double-click. This property appears with List and Table widgets only.

Notification Properties

You specify Notification properties when you want a widget to inform its application model that certain actions have taken place, namely, that the widget has taken focus, changed internal state, or given up focus. Notification properties are useful for facilitating complex flow of user input.

Each notification property specifies the symbolic name of a notification callback, which is the message you want the widget to immediately send after the relevant action. For each notification callback you specify, you must program the application model to contain a corresponding method. You implement this method to provide the desired response to the widget's action. You can implement the notification method to activate other widgets in the interface.

Table 11-10 *Notification properties*

Property	Description
Entry	Specifies the symbolic name for the widget's entry notification callback. The widget sends this message to its application model immediately after taking focus. You must implement a corresponding method in the application model that provides the desired response to this event.
Change	<p>Specifies the symbolic name for the widget's change notification callback. The widget sends this message to its application model immediately after the widget sends its input value to its value model. You must implement a corresponding method in the application model to provide the desired response to this event. Note that specifying a change notification callback is similar to registering an interest in a value model via <code>onChangeSend:to:</code>, in that both cause a message to be sent after the value in the value model has changed.</p> <p>However, the two techniques also differ in important ways: The change notification callback is sent only to the application model. The message specified by <code>onChangeSend:to:</code> is sent to the specified receiver, which may, but need not be the application model. If both techniques are used together, the message sent by <code>onChangeSend:to:</code> is sent first, and the change notification callback is sent second.</p>
Exit	Specifies the symbolic name for the widget's exit notification callback. The widget sends this message to its application model immediately after it gives up focus. You must implement a corresponding method in the application model to provide the desired response to this event.
D. Click	Specifies the symbolic name for the widget's double-click notification callback. This callback is the message that the widget sends to its application model in response to a double-click. You must implement a corresponding method in the application model to provide the desired response to this event. This property appears with the List and Table widgets only.

Color Properties

A widget can have up to four color zones:

- n Foreground, which is determined by the widget itself, plus the current look policy. Typically it is a salient characteristic of the widget, such as its label, if it has one.
- n Background, which is determined by the widget itself, plus the current look policy. Typically, it is a less salient characteristic, such as the widget's interior area "behind" any label.
- n Selection foreground, which is the color of the foreground when the widget is selected.
- n Selection background, which is the color of the background when the widget is selected.

You use the Color page of the Properties Tool to apply color to any of these zones.

Table 11-11 Color Properties

Property	Description
V	Allows you to fine-tune the value of the selected color. The value is the degree of lightness of a color, from light to dark
S	Allows you to fine-tune the saturation of the selected color. The saturation is the degree of vividness, from grayish to vivid. (Only appears when a color other than white or black is selected)
H	Allows you to fine-tune the hue of the selected color. The hue is the gradation of color from red through yellow, green, cyan, blue, magenta, back to red. (Only appears when a color other than white or black is selected)
Read	Displays the colors that are currently assigned to the four color zones for the selected widget on the canvas.
Foregrnd	Specifies whether to assign the selected color to the widget's foreground color zone. Clicking on the Foregrnd button toggles between assigning the selected color and assigning no color.

Table 11-11 Color Properties

Property	Description
Backgrnd	Specifies whether to assign the selected color to the widget's background color zone. Clicking on the Backgrnd button toggles between assigning the selected color and assigning no color.
Selection Foregrnd	Specifies whether to assign the selected color to the widget's selection foreground color zone. Clicking on the Selection Foregrnd button toggles between assigning the selected color and assigning no color.
Selection Backgrnd	Specifies whether to assign the selected color to the widget's selection background color zone. Clicking on the Selection Backgrnd button toggles between assigning the selected color and assigning no color.

Position Properties for Bounded Widgets

Available for bounded widgets: Action Button, Slider, Input Field, Menu Button, Combo Box, Text Editor, List, Table, DataSet, Notebook, Subcanvas, View Holder, Divider, Region, Group Box.

Table 11-12 Position Properties

Property	Description
L	Identifies the positioning choices (Proportion and Offset) for the left edge of the selected widget.
T	Identifies the positioning choices (Proportion and Offset) for the top edge of the selected widget.
R	Identifies the positioning choices (Proportion and Offset) for the right edge of the selected widget.
B	Identifies the positioning choices (Proportion and Offset) for the bottom edge of the selected widget.
Proportion	Indicates the fraction of the window's width or height that precedes the widget's edge. You can specify a value from 0 (the position of the left or upper window edge) to 1 (the position of the right or lower window edge). The proportionally-determined position can be further adjusted by an offset.

Table 11-12 Position Properties

Property	Description
Offset	Indicates how many pixels to adjust the widget's edge from the proportionally determined starting position. A positive offset adjusts the edge rightward or downward from the proportional setting. A negative offset adjusts leftward or upward from a nonzero proportion. A 0 offset causes the position of the widget edge to be determined entirely by the proportion.

Drop Source Properties

Drop Source properties are available for the List widget only.

Table 11-13 Drop Source Properties

Property	Description
Drag Ok	<p>Specifies the symbolic name for the message that queries whether to initiate a drag and drop from the widget. The widget sends this message to the application model when the user starts to drag the mouse pointer within the widget's bounds. The specified symbol must end with a colon (:)—for example, <code>customerWantToDrag:</code>. This allows the widget to pass its controller as part of the message.</p> <p>You must implement a corresponding method in the widget's application model. Typically, this method tests whether the widget's data exists and is appropriate for transfer. This method must return <code>true</code> if the operation is to proceed, and <code>false</code>, otherwise.</p>
Drag Start	<p>Specifies the symbolic name for the message that initiates the drag and drop. The widget sends this message to the application model only if the Drag OK method returns true. The specified symbol must end with a colon (:)—for example, <code>doCustomerDrag:</code>. This allows the widget to pass its controller as part of the message.</p> <p>You must implement a corresponding method in the widget's application model. This method must create initialized instances of <code>DragDropData</code>, <code>DropSource</code>, and <code>DragDropManager</code> and then send <code>doDragDrop</code> message to the <code>DragDropManager</code> instance. This method may use the effect symbol returned by the completed operation to trigger followup actions on the drag source.</p>
Select On Down	<p>Specifies the effect of pressing the mouse button down in a drop source list. When this property is on (the default), the user can perform drag and drop as a single gesture. That is, pressing the mouse button causes the list item under the pointer to be selected immediately, so that the drag can proceed from there.</p> <p>When this property is off, the user must click on a list item to select it, and then, as a separate gesture, press the mouse button down again to start the drag.</p>

Drop Target Properties

Drop Target properties are available for all widgets except linked and embedded dataforms.

Table 11-14 Drop Target Properties

Properties	Description
Entry	<p>Specifies the symbolic name for the message to be sent by the <code>DragDropManager</code> when the user drags the mouse pointer into the widget's bounds. This message is sent only if the widget's Drop property is also filled in. The specified symbol must end with a colon (:)—for example, <code>browseDragEnter:</code>. This allows the <code>DragDropManager</code> to pass information about the object being dragged (a <code>DragDropContext</code> instance) as part of the message.</p> <p>Specifying this property is optional; however, if you do specify it, you must implement a corresponding method in the application model. Typically, this method uses information in the provided <code>DragDropContext</code> to determine whether it is appropriate to drop the dragged data in this widget, and, if so, creates a corresponding visual effect in the widget. Because this method executes once, it should be used to turn on visual effects (such as highlighting) that don't track the pointer. By convention, this method should return a symbol representing the anticipated type of transfer (typically <code>#dropEffectMove</code>, <code>#dropEffectCopy</code>, or <code>#dropEffectNone</code>).</p>
Over	<p>Specifies the symbolic name for the message sent by the <code>DragDropManager</code> while the user moves the mouse pointer within the widget's bounds. This message is sent only if the widget's Drop property is also filled in. The specified symbol must end with a colon (:)—for example, <code>browseDragOver:</code>. This allows the <code>DragDropManager</code> to pass information about the object being dragged (a <code>DragDropContext</code> instance) as part of the message.</p> <p>Specifying this property is optional; however, if you do specify it, you must implement a corresponding method in the application model. Typically, this method uses information in the provided <code>DragDropContext</code> to determine whether it is appropriate to drop the dragged data in this widget, and, if so, changes the visual appearance of the widget. Because this method executes whenever the pointer moves, it can produce visual effects that track the pointer's movement. This method must return a symbol representing the anticipated type of transfer (typically <code>#dropEffectMove</code>, <code>#dropEffectCopy</code>, or <code>#dropEffectNone</code>). The <code>DragDropManager</code> responds to this symbol by changing the shape of the pointer while it is over the widget.</p>

Table 11-14 *Drop Target Properties*

Properties	Description
Exit	<p>Specifies the symbolic name for the message that is sent by the <code>DragDropManager</code> when the user moves the mouse pointer out of the widget's bounds. This message is sent only if the widget's Drop property is also filled in. The specified symbol must end with a colon (:)—for example, <code>browseDragOver:</code>. This allows the <code>DragDropManager</code> to pass information about the object being dragged (a <code>DragDropContext</code> instance) as part of the message.</p> <p>Specifying this property is optional; however, if you do specify it, you must implement a corresponding method in the application model. Typically, this method uses information in the provided <code>DragDropContext</code> to determine whether a drop was valid for this widget, and, if so, to reverse any changes that were made to the widget's visual appearance. Because this method executes once, it is appropriate for turning off visual effects that were turned on by the Entry method. By convention, this method should return a symbol representing the anticipated type of transfer (typically <code>#dropEffectMove</code>, <code>#dropEffectCopy</code>, or <code>#dropEffectNone</code>).</p>
Drop	<p>Specifies the symbolic name for the message that is sent by the <code>DragDropManager</code> when the user releases the mouse button with the pointer inside the widget's bounds. The specified symbol must end with a colon (:)—for example, <code>browseDragOver:</code>. This allows the <code>DragDropManager</code> to pass information about the object being dragged (a <code>DragDropContext</code> instance) as part of the message.</p> <p>You must implement a corresponding method in the application model. Typically, this method uses information in the provided <code>DragDropContext</code> to determine whether the dragged data can be dropped in this widget, and, if so, what action to take as a result. In addition to initiating the desired action, this method reverses any changes that were made to the widget's visual appearance by the Entry method, if appropriate. This method returns a symbol representing the type of transfer (typically, <code>#dropEffectMove</code>, <code>#dropEffectCopy</code>, or <code>#dropEffectNone</code>); this symbol is returned to the Drag Start method, where further processing may occur.</p>

Define Dialog

The define dialog generates supporting code in a limited fashion:

- n It creates an instance variable in the application model when appropriate. The variable is given the same name as the component's **Aspect** or **Client** property.
- n It creates an instance method that returns the object held by the instance variable, or an empty action method for a button. The method also has the same name as the **Aspect**, **Client** or **Action** property.
- n Optionally, it adds initialization code to the method that accesses the instance variable, to initialize it with an appropriate information holder. You can override the default initialization by creating an instance method named `initialize`, in which you assign to the variable a custom value in a value holder.

The define dialog can be opened by selecting **methods?define** in the canvas's menu or by clicking **Define...** from the Canvas Tool. You can limit the define dialog's scope by first selecting a single component, a group of components, or none (which is equivalent to selecting all components).

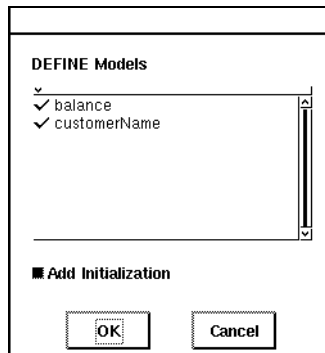
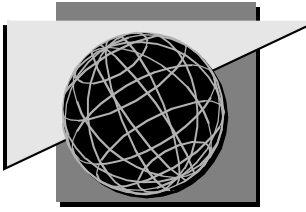


Figure 11-8 Define Dialog

The **define** dialog lists all of the information holders and actions referenced in the selected components. You can instruct the tool to ignore an information holder or action by deselecting its name in the list (click to remove the check mark).

By default, initialization code will be created in addition to the instance variable and accessing method. Turn off the **Add Initialization** feature if

you prefer to leave the variable uninitialized or you want to initialize it some other way.



Chapter 12

Database Application Building Tools

Using the VisualWorks Objectlens to synchronize data objects with relational data tables, the database tools allow relational data to be accessed and manipulated as objects.

A brief description of the database applications tools is provided below. For more detailed information, please refer to the VisualWorks Database documentation (online and hardcopy).

The Data Modeler

The Data Modeler is the central information point for the mapping between the database tables and Smalltalk classes.

Canvas Composer

The canvas composer creates a canvas, and stores it on a data form and then opens the VisualWorks painting tools to allow canvas customizing. In order to maximize reusability, a data form may have multiple canvases and it is recommended that canvases be constructed to provide application or organization standard components to use as building blocks.

VisualWorks Painting Tools

The VisualWorks painting tools (Canvas Tool and Palette) can be used to modify database specific canvases that are automatically created. New fields or columns can be manually added or deleted.

Embedded and Linked Data Forms

Building database applications within VisualWorks can be extended by using the Embedded and the Linked data forms. Selectable from the VisualWorks palette, using these specifications provide the following:

- n *Embedded data forms* - give the application developer the flexibility to use a part of a parent's application canvas for a subordinate application.
- n *Linked data forms* - give the application developer the ability to create additional canvases instead of using a part of the parent's canvas. The graphical representation of a linked data form in the parent's canvas is similar to a VisualWorks action button.

Mapping Tool

This graphical tool shows how the instance variables of a specific class are mapped to the columns of a table. It shows the type associated with each variable as well as the type and attributes of the mapped column.

The mapping tool provides the mechanism to map variables to different columns and to change the type or attributes of the columns.

It also provides actions for editing the type and columns. New variables can be added, they can be removed, their type changed, a table may automatically be created for a class, etc.

The Query Editor

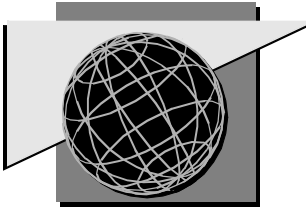
The Query Editor is a tool used to define a query that is to be associated with the application, the ownQuery, any Restricted Query defined, a menu query or and arbitrary query.

Menu Queries

The Query editor contains some utilities that make it easy to define a menu in terms of a query. To do so, the 'Select' field must be composed of a string and another arbitrary object. The strings will be the labels of the menu and the other objects will be the values.

Ad Hoc SQL Editor

The query editor allows you to make ad hoc queries to the target data and visually specify the data to be displayed with no required form building or Smalltalk programming.



Chapter 13

Application Delivery Tools

This chapter describes the tools available for extracting an application from VisualWorks and delivering it to its end users:

- n Parcel List
- n Parcel Browser
- n Image Maker

The first two tools enable you to create groups of class and method definitions, called *parcels*, that can be loaded into a running image without a compiler. The third tool enables you to create a minimal image for deployment to end users.

This chapter describes these tools. For more information about parcels and image-making, see Part IV, “Application Delivery.”

Parcel List

To open the Parcel List, choose **Tools?Parcel List** from the VisualWorks main window.

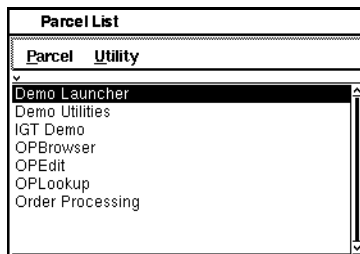


Figure 13-1 Parcel List

The Parcel List shows all of the parcels in the image. From the Parcel List you can perform operations on entire parcels, such as creating, loading, saving, and removing them.

The Parcel List has a menu bar with two menus: **Parcel** and **Utility**. The **Parcel** and **Utility** menu commands are described in tables 13-1 and 13-2.

The Parcel List also has a pop-up <Operate> menu. The <Operate> menu is identical to the **Parcel** menu.

Filing in `extras/tooldd.st` enables drag and drop in the Parcel List. When enabled, you can add classes and methods to a parcel by dragging them from the System Browser to a parcel in the Parcel List.

Parcel Menu Commands

Table 13-1 Parcel Menu Commands

Command	Description
New...	Prompts for a unique parcel name. Creates a new, empty parcel with that name. A parcel name may be any string. The Parcel List strips names of leading and trailing blank spaces and compresses multiple blank spaces into a single space.
Load...	Prompts for a filename. Loads the parcel and all the classes and methods it contains from the specified parcel file. Loading a parcel will display error messages if an attempt is made to overwrite an existing class definition or if prerequisite classes are not already loaded. See Chapter 25 for more information. <i>Note: Parcel files do not contain source code. Browsing classes and methods that have been loaded results in decompiled code.</i>
Save As...	Prompts for a filename. Writes the selected parcel and all the classes and methods it contains to a parcel file.
File Out As...	Prompts for a filename. Stores a description of each class and method in the selected parcel in a form that enables the class (including all of its methods) and any extension methods to be placed in another VisualWorks image with the file in command. Does not file out the parcel itself.
Remove	Removes the selected parcel from the system. The definitions (classes and methods) that the parcel contained remain in the system.

Table 13-1 Parcel Menu Commands

Rename As...	Prompts for a new name. Replaces the current parcel name, both in the Parcel Browser and in the Parcel List. Rename As has same restrictions as Name .
Browse...	Opens a Parcel Browser on the selected parcel.
Empty	Removes all of the definitions (classes and methods) from the selected parcel. The parcel itself remains in the system, as do the definitions for the classes and methods that were in it.

Utility Menu Commands

Table 13-2 Utility Menu Commands

Command	Description
File Into Parcel...	Prompts for a filename. Files in code from the specified source file and places all of the definitions (classes and methods) into the selected parcel.
Changes Into Parcel	Adds all of the definitions (classes and methods) from the current change set to the selected parcel.
Make Remove Script...	Prompts for a filename. Writes a script that will remove from the system all of the definitions in the selected parcel. The script will not remove the parcel itself.

Parcel Browser

The Parcel Browser displays the contents—categories, classes, protocols, and methods—of a single parcel and allows you to add and remove contents.

Open the Parcel Browser from the Parcel List by either:

- n Double-clicking on a parcel, *or*
- n Selecting a parcel and choosing **File?Browse**.

Structure

The Parcel Browser looks much like the System Browser (described in Chapter10, “Smalltalk Programming Tools”).

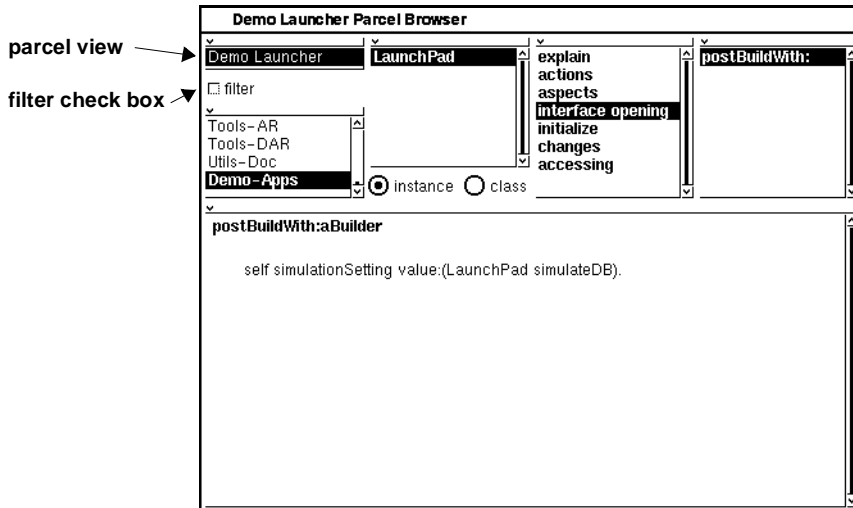


Figure 13-2 Parcel Browser

The Parcel Browser has five upper views and one lower view, as shown in Figure 13-2. Each view provides a lower level of detail in the code library, beginning with the parcel name and category list and ending with the code itself.

The Parcel Browser differs from the System Browser in three ways:

- n The Parcel Browser contains an additional view above the category view. The *parcel view* shows the name of the parcel and has its own <Operate> menu, which contains commands that affect the parcel as a whole.
- n The Parcel Browser displays the definitions that are in the current parcel. It can be made to show all of the definitions that are in the system by unselecting the **filter** check box.
 - q Items that are in the current parcel are in bold.
 - q Items that are in the system but not in the current parcel are in regular typeface.
 - q A class name that is in italics indicates that the class itself is not in the parcel but that one or more of its methods are.
- n The definitions displayed in the Parcel Browser cannot be edited. The Parcel Browser allows only operations that:
 - q Change which definitions are in the parcel
 - q Browse code for possible inclusion in the parcel

- q Write the parcel's definitions out of the image

Each view in the Parcel Browser has a unique menu, offering commands that are appropriate to its contents. The commands for each view are presented below, listed in the order in which they appear in the menu. In some views, the menu has fewer options when no item in the view has been selected.

Parcel View

Table 13-3 Protocol View Commands

Command	Description
file out as...	Prompts for a filename. Stores a description of each class and method in the selected parcel, in a form that enables the class (including all of its methods) and any extension methods to be placed in another VisualWorks image with the file in command. Does not file out the parcel itself.
save as...	Prompts for a filename. Writes the selected parcel and all of the classes and methods it contains out to a parcel file.
update	Brings the Parcel Browser listing up to date. Useful after creating or filing in new categories, classes, protocols, or methods.
rename as...	Prompts for a new name. Replaces the current parcel name, both in the Parcel Browser and in the Parcel List. Same name restrictions as in the Parcel List.
remove	Removes the current parcel from the system. The definitions (classes and methods) that the parcel contained remain in the system.
empty	Removes all of the definitions (classes and methods) from the selected parcel. The parcel itself remains in the system, as do the definitions for the classes and methods that were in it.
version	Shows the parcel's version in the code view. You can edit the version and accept the change. The version may be any string.
comment	Shows the parcel's comment in the code view. You can edit the comment and accept the change.
summary	Shows a summary, in the code view, of the classes and methods that are defined in the parcel.

Category View

Table 13-4 Category View Commands

Command	Description
add to parcel	Adds all of the class and method definitions that are in the selected category to the current parcel.
remove from parcel	Removes all of the class and method definitions that are in the selected category from the parcel. The category and its contents remain in the system.
find class...	Prompts for a class name. Selects that class in the class view (and its category, in the category view). If a wildcard character is used, displays a dialog of all classes with matching names. For example, the pattern C*View displays a list of all classes beginning with “C” and ending in “View.”

Class View

Table 13-5 Class View Commands

Command	Description
add to parcel	Adds the selected class and all of the methods that it contains to the current parcel.
remove from parcel	Removes the selected class and all of the methods that it contains from the current parcel. The class and its contents remain in the system.
hierarchy	In the code view, displays the names of the currently selected class, its superclasses, and its subclasses, with indentations to indicate hierarchic precedence.
definition	In the code view, displays the formal definition of the currently selected class.
comment	In the code view, displays the class comment.
inst var refs...	Displays a dialog that lists all of the instance variables of the currently selected class and its superclasses. Select a variable name to open a browser on all methods that refer to that variable.

Table 13-5 Class View Commands

class var refs...	Opens a browser on methods that refer to a selected class variable.
class refs	Opens a browser on all methods that refer to the currently selected class.

Protocol View

Table 13-6 Protocol View Commands

Command	Description
add to parcel	Adds all of the methods that are in the selected protocol to the current parcel.
remove from parcel	Removes all of the methods that are in the selected protocol from the current parcel. The protocol and its contents remain in the system.
find method...	Displays a dialog that lists all of the instance methods (if the instance switch is selected) or class methods of the currently selected class. Select a method name to display its code in the code view.

Method View

Table 13-7 Method View Commands

Command	Description
add to parcel	Adds the selected method to the current parcel.
remove from parcel	Removes the selected method from the current parcel. The method remains in the system.
senders	Opens a new browser on all methods that send the currently selected message.
implementors	Opens a browser on all methods that implement the currently selected message (i.e., methods having the same name that exist in other classes as well as this one).
messages...	Displays a dialog of all method selectors that exist in the currently selected method. Select one to open a browser on all methods that implement that message.

Code View

Table 13-8 Code View Commands

Command	Description
find...	Searches for the specified string.
replace...	Replaces one specified string with another.
undo	Reverses the most recent cut or paste.
copy	Places a copy of the highlighted text in memory. If <Shift> is held down while copy is selected, the text is copied to the window manager's clipboard.
cut	Places a copy of the highlighted text in the paste buffer and then deletes the original.
paste	Deletes the highlighted text (if any) and then places a copy of the most recently copied or cut selection in that location. If <Shift> is held down while paste is selected, a dialog presents the five most recent text segments that have been copied or cut, including the window manager's clipboard.
do it	Executes the highlighted text as a Smalltalk expression. The scope of execution is the selected class, so class variables can be used in the expressions, and self refers to the selected class.
print it	Same as do it , except a description of the resulting object is inserted in the text. The printed string becomes the current selection, so it can be deleted easily.
inspect	Same as do it , except an inspector is opened on the resulting object.
accept	Saves the contents of the code view. Only applicable for a parcel's version or comment. You may not change any class or method definitions using the Parcel Browser. You must use the System Browser or another programming tool to edit classes and methods.
cancel	Restores the entire text to its condition when it was last compiled (with accept).
hardcopy	Print a copy of the text or code on paper.

Image Maker

To create custom images, you use a tool called Image Maker. Image Maker enables you to remove development tools and other unwanted classes from an image. The resulting image is more appropriate for distribution to the end users of your application. It also occupies less disk space—perhaps significantly, depending on the extra classes that you specify for removal.

To use Image Maker:

1. File in **imagemkr.st** from the **utils** directory.
2. In a Workspace, execute the following:

ImageMaker open

Image Maker displays a window that allows you to choose what you want to remove from your development image before saving it as a deployment image.

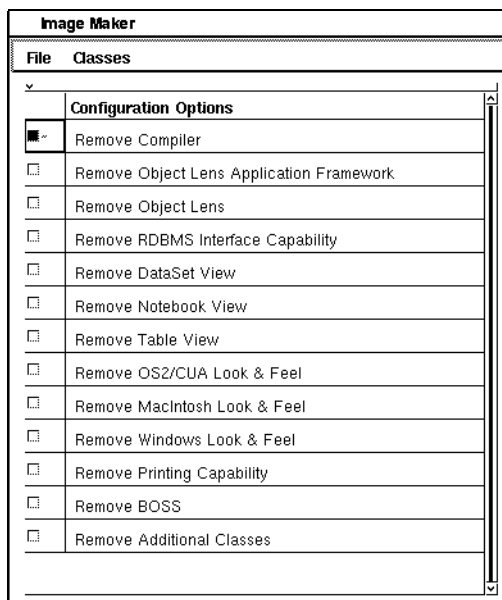
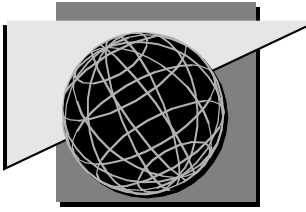


Figure 13-3 Image Maker

3. Choose the capabilities that you want removed from your development image and not included in the new deployment image file. (Note that the capabilities are not removed from your saved development image file.)
4. Choose **File?Make Deployment Image**.
5. Follow the instructions presented by Image Maker.

For more information about creating deployment images with Image Maker, see Part IV, “Application Delivery.”



Chapter 14

Debugging Techniques

The VisualWorks debugger enables you to look at the methods that are waiting for a return value when a program interrupt occurs, examine the values of variables in each context, dynamically change a value or a method, insert breakpoints, and restart execution at a chosen location with the new values and logic experimentally in place.

This chapter presents a task-oriented perspective, presenting a fuller discussion of how to use the Debugger and supporting tools. The chapter is organized by debugging techniques, roughly in order of increasing intensity.

Reading the Execution Stack

To diagnose a problem, sometimes it is sufficient to see the last few entries in the context stack. The Debugger's top view lists as much of the stack as you want to see, but you may not even have to launch the Debugger. The error notifier that results from a program interrupt lists the last five contexts. An error notifier showing the results of a programmatic error (3 + 'two') is shown in Figure 14-1.

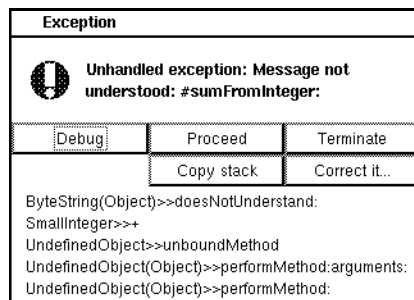


Figure 14-1 Error Notifier

The window label tells us that a `sumFromInteger:` message was sent to an object that does not implement a method by that name. (This summary is repeated in the top line of the window, for situations in which the window label is not wide enough to display all of the message.) Looking at the top line of the stack, we see that it was an object of type `ByteString`. (`ByteString` didn't understand the message, so it invoked the `doesNotUnderstand` method implemented by its parent class, `Object`). This is puzzling because we sent a `+` message to a `SmallInteger`, as recorded in the second line of the stack transcript. The last three lines of the transcript are not enlightening—they merely expose some of the execution machinery, which we have no reason to suspect in this case.

This example illustrates two features of the execution stack worth emphasizing. The first line of the execution stack is often only of marginal interest, because it usually represents the method that handles the error—it doesn't necessarily help you understand what caused the error. Also, the execution machinery is a frequent inhabitant of the execution stack—very quickly you learn to read around it.

Back to our example: Something odd happened in the `SmallInteger>>+` method. You can either use the System Browser to look at that method, or you can open a Debugger, as described in the next section.

Tracing the Flow of Messages

As described in the previous section, the error notifier displays the last five message-sends in the execution stack. When you need to look at one or more of those methods, the Debugger is the most convenient tool to use. To open a Debugger, select **debug** in the notifier's <Operate> menu. The notifier will disappear after the Debugger is opened.

The Debugger's window label is identical to that of the notifier from which it was created. The execution stack view, at the top, contains the most recent message-sends that occurred before the error. To see the associated method, select a message-send. In the illustration, `SmallInteger>>+` has been highlighted. The code view, in the center of the Debugger, displays the method. Within that method, the message-send that was being processed when the program failed is highlighted automatically.

A debugger displaying the results of a programmatic error (`3 + 'two'`) is shown in Figure 14-2.

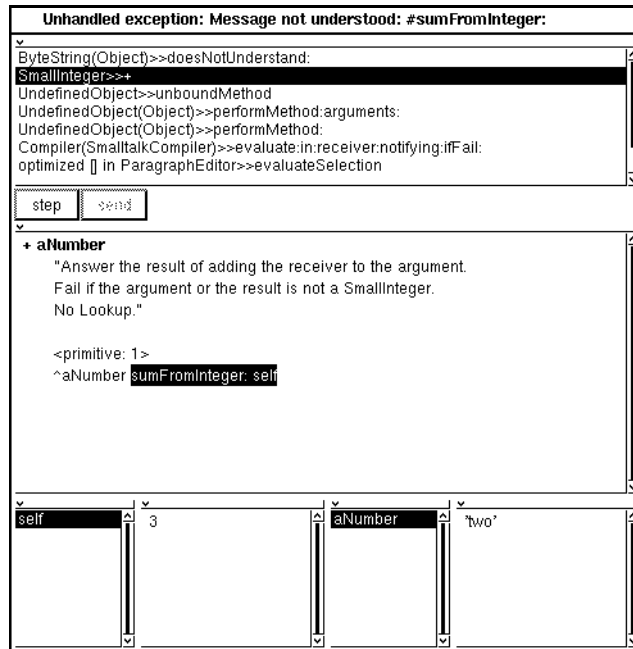


Figure 14-2 Debugger

Continuing our example from the previous section, in which the expression `3 + 'two'` was executed, we can see that the illegal expression could not be handled by the primitive method that normally adds two integers together. The alternative Smalltalk code was then executed.

Here we find the explanation for the mysterious `sumFromInteger:` message, which was sent to a `ByteString`. As you can see, the `+` method calls the `sumFromInteger:` method. But the *receiver* of the `+` message is the *argument* (`self`) of the `sumFromInteger:` message. The message receiver and argument have traded places. We know that the argument was the string `'two'`, so the `sumFromInteger:` message is being sent to an object of the wrong class, to a string instead of an integer. In the next section, we'll show how to verify this deduction.

Inspecting and Changing Variables

The bottom of the Debugger is devoted to two inspectors that allow you to see the values of variables as they exist at the chosen point in the execution stack. Each inspector consists of a pair of views, with a list of variables in the left view and the value of a selected variable in the right view. The inspector on the left is for instance variables, while the right-hand inspector displays temporary variables.

In the example that was introduced above, the expression `3 + 'two'` has caused the expression `'two' sumFromInteger: 3` to be executed. Now we know where `sumFromInteger:` came from. We can also see why it was “misunderstood” as indicated in the error notifier's window label—it was addressed to a string instead of the expected number. To verify this, select `aNumber` in the inspector view. Figure 14-3 shows the resulting inspector.

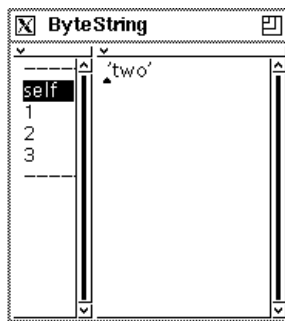


Figure 14-3 An inspector

The Debugger's inspectors let you change the value of a variable and then restart the program. Simply edit the value, changing `'two'` to a legal value such as the integer `2`. Then select **accept** in the <Operate> menu. A confirmer will then offer to begin at the top of the current method using the new value. Click on the **yes** button. You can then select **restart** in the stack view's menu, triggering execution.

In practice, the value `'two'` normally would be supplied by another method rather than a Workspace expression. Having traced the problem to this value, you can correct its parent method. To do so, edit and **accept** the revised method in any code view such as the one in the Debugger or the one in the System Browser.

Inserting Status Messages

In some situations, it is useful to have your program display status messages during the debugging phase. For example, you might want a record of the changing values of a particular variable as it passes through various states. In such a case, the System Transcript is used to display each message and thus accumulate the desired record.

To open a System Transcript, select **Utilities** in the VisualWorks main window, then select **transcript** in the submenu. (The system sometimes displays messages in the System Transcript, so it's a good idea to keep one open at all times.)

To send output to the System Transcript, insert expressions such as the following in your code, substituting a pertinent object name for the italicized word:

```
Transcript show: anObject printString.  
Transcript cr.  
Transcript tab.  
Transcript show: 'Checkpoint 1'; cr.
```

To avoid an update of the display with each part of a larger message, use `nextPutAll:` or `print:`, then use `endEntry` to output the message, as in:

```
Transcript nextPutAll: 'The account is: '; print: account; endEntry.
```

To clear the System Transcript for a new batch of messages, select **cancel** in its menu. Alternatively, execute the expression `Transcript clear`.

The System Transcript is a distinguished instance of the `TextCollector` class. For more transcript output messages, see the instance methods of that class.

Interrupting a Program

There are two ways to manually stop a Smalltalk program: by typing a *user interrupt* key sequence or by inserting a *halt* message in the program.

<Control>-c is the key sequence assigned to the user interrupt function. Enter this key sequence when you want to freeze a program that is looping endlessly, or to capture its state at a specific observable stage.

For more precise control, insert the expression **self halt** in a method at the location where you want execution to be interrupted, then **accept** the revised method. The next time that method is called, an error notifier will be displayed at the specified juncture.

Both user interrupts and **halt** messages generate the usual error notifier, which can be used to open a Debugger. The next section describes how to restart an interrupted program.

Restarting a Program

The Debugger provides five ways to restart an interrupted program, allowing you to control the starting and stopping place for continued execution. The first two commands described below control the starting point—since they continue execution as far as possible, the Debugger is closed at the outset. The last three control the stopping place—the Debugger is left open so you can inspect the conditions at the new position in the stack.

To close the Debugger and restart at the beginning of the currently selected method, select **restart** in the stack view's <Operate> menu. This is useful mainly after you have altered the method's code. To restart at a prior position in the program, select the context in which you want execution to begin, then select **restart** in the <Operate> menu.

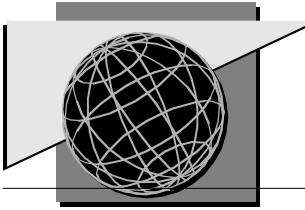
To close the Debugger and continue from the point of interruption in the currently selected method, select **proceed** in the <Operate> menu. This is useful when you have changed the values of one or more variables in the Debugger's inspector views, or when the current interrupt is of less interest than one that is still to come. Execution continues as follows:

- n If a user interrupt caused the break, execution proceeds from the point of interruption.
- n If anything other than a user interrupt caused the break, execution proceeds with an assumed value as the return of the interrupted message-send. That value is nil unless you execute a **do it** or **print it** command inside the Debugger, in which case the value returned from that action is used.

To continue to a specific place in a method, click on that location to put the insertion point (caret) there, then select **skip to caret** in the <Operate> menu.

To execute the next message-send, select the **step** button in the Debugger or select **step** in the <Operate> menu. Execution stops after the value is returned from the called method.

To send the next message, and “follow” it by displaying the called method, select the **send** button or select **send** in the <Operate> menu. This command provides the finest granularity of message-flow inspection.



Chapter 15

Managing Projects and Versions

VisualWorks provides tools to help you carve a large task into manageable projects, to share code with other developers, and to track code versions. These tools—Project, Change List and Change Set—are described in structural terms in the chapter “Environment Tools,” which describes how to use those tools.

We begin with Project, which helps you organize your display into groups of views and your coding into groups of changes. Then the various ways of viewing and manipulating those changes are discussed.

Entering and Exiting a Project

Creating a Project involves launching a new Project window, by selecting **Changes?Open Project** in the VisualWorks main window. The Project window contains a text view in which you can describe the project. Above that view is an **enter** button—use this button to activate the project, clearing the display of the parent project’s windows.

The Project window with the description **User Interface** is shown in Figure 15-1.

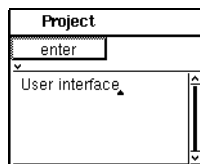


Figure 15-1 Project Window

A new VisualWorks main window will be provided with the new project. Use the VisualWorks main window to launch tools as usual. When you exit the project, those windows will be remembered by the system for the next time

you **enter** the project. To exit the current project, choose **Changes?Exit Project** in the VisualWorks main window.

Projects can be nested to create a hierarchy of working contexts. Besides allowing you to create separate groups of tools for different aspects of your work, Project also keeps a separate Change Set for each project. The Change List, however, when used to access the changes file, ignores project boundaries. Change Set and Change List are discussed further below.

To close a project permanently, select **close** in its <Operate> menu. If you have made changes to the system in that project, a confirmer will verify your intent to close the project. The image is not affected by this decision—it reflects changes made in any project. Similarly, the changes file is the same for all projects. Only the Change Set associated with a project, along with the window setup, is lost when a project is closed.

Summarizing Project Changes

The Change Set is a summary of changes that have been made within a project. Unlike the Change List, it does not track the evolution of those changes. Instead, its intent is to list the affected parts of the system so you can use the **file out as** command to store your work in a set of disk files. This strategy for transporting changes from one image to another is most convenient when the changes are confined to a few classes and categories. For more involved sets of changes, it may be easier to use the Change List to **write file** with the desired changes.

A ChangeSet inspector, summarizing the changes for the active project is shown in Figure 15-2.

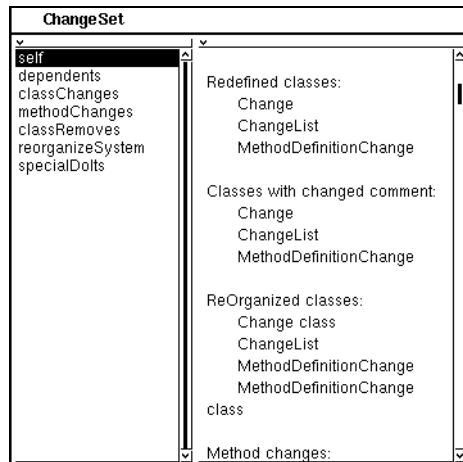


Figure 15-2 ChangeSet Inspector

To open a Change Set, select **Changes?Inspect ChangeSet** in the VisualWorks main window. An inspector will be opened on the current Change Set, with types of changes listed in the left view. Select a type to display the changes in the right view, or select **self** to see the entire list in formatted form.

The Change Set can contain any of the following types of change:

- n Added, deleted and changed classes
- n Added, deleted and changed methods
- n Changes in class categories (reported as “Reorganized System”) and message categories (reported as a “Reorganized class”)
- n “Special doIts,” rarely encountered, involving a system change such as renaming a global variable, that is effected via an executed expression (a doIt). Such a change is only captured in the Change Set when the expression is executed by passing it as an argument to Smalltalk evaluate-AndRemember:.

In the spirit of summarization, the Change Set does not separately report changes involving methods in a class that has been added. When you **file out** the new class, its methods will be included anyway. To file out the entire change set, select **file out as** in the left view’s <Operate> menu.

To empty the Change Set for the active project, in readiness for a new batch of work, select **empty** in the left view's menu. You can also perform this operation by choosing **Changes?Empty Changes...** in the VisualWorks main window.

To update an open Change Set window after making a change to the system, select a different type of change in the left view temporarily. Unlike the Change List, the Change Set is not affected when you save the image.

The Change Set lists methods that have been changed but it has no code view with which to browse them. To open a method browser on the changed methods in the Change Set, execute the expression **Smalltalk browseChangedMessages**.

Some programmers prefer to have the Change Set remove entries related to a class when that class is filed out via a browser, and similarly for all classes in a category when the category is filed out. To turn on this feature, execute the following expression:

```
Browser removeChangesOnFileOut: true
```

Reverting to a Prior Version

The system automatically maintains a list of all changes made to an image. This Change List is stored in a disk file having the same name as the image, with the **.cha** extension. To open a specialized browser for use with the Change List, select **Changes?Open Change List** in the VisualWorks main window.

During the course of development, a class or method may undergo several changes. The Change List makes it easy to see the evolution and to examine the details of the code at any stage in its development. This is particularly useful when you need to see a prior version so you can change the code back.

The Change List contains two views and a set of toggle switches. To display the changes that have occurred since the last snapshot was taken, select **recover last changes** in the <Operate> menu of the list view at the top. If you want to display changes that are in the Change Set, select **display system changes** instead. To display all changes, ignoring snapshot boundaries, select **read file** and supply the name of the changes file as described above.

A Change List browser, showing the use of the same switch to narrow the displayed list of changes is shown in Figure 15-3.

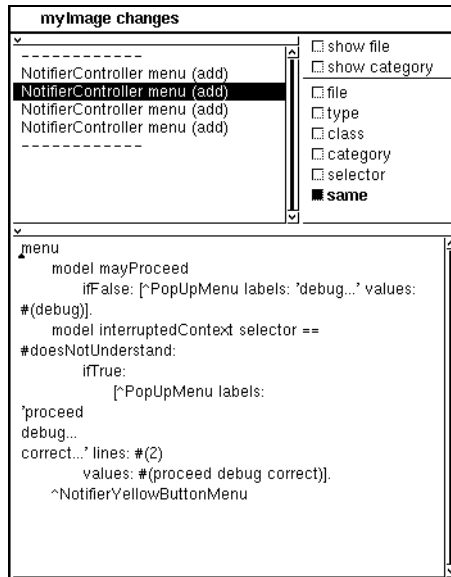


Figure 15-3 Change List Browser

Entries in the Change List generally identify the affected object and the nature of the change, as in “NotifierController menu (add).” When you select an entry, the affected class or method appears in the text view as it existed after the change.

Use the switches in the upper right corner of the Change List to filter the displayed entries. For example, to display only those changes that affect the same class as the one affected by the selected change entry, click on the **class** switch. To further restrict the listing to identical entries, such as “NotifierController menu,” click on the **same** switch.

With the **same** filter turned on, as shown in the illustration, it is easy to examine the evolution of a class or a method. To revert the code to a prior version, select the entry representing that version and then select **replay selection** in the <Operate> menu. You can also use **replay all**, when you want to incorporate all of the listed changes.

Sharing Code

When the code you want to transport to another image is confined to a few classes, methods or categories, use the System Browser's **file out as** capabilities to create a set of disk files containing the code. Use the Change Set, if necessary, to identify the affected classes and methods. When you want to save all of the changes in the Change Set, use that tool's **file out as** command.

When the code you want to share consists of fragments from many different classes and categories, it may be more convenient to use the Change List to **write file** with the desired code. Begin by loading all changes into a Change List, as described in the previous section.

Next, remove the irrelevant changes. For example, doIts are likely candidates for removal because they rarely affect the image in a lasting way. Also, remove duplicate entries, as when a method has undergone several changes—leave only the last entry in each case. Use **remove selection** and **remove all** to mark one or more changes for deletion, then use **forget** to erase them from the list. Use the filter switches to control the affected range of entries.

For example, to remove all doIts, begin by selecting any doIt. Then turn on the **type** switch so all of the doIts are listed. Select **remove all** in the <Operate> menu to mark them for deletion, then **forget** to erase them. Then turn off the **type** switch to see the remaining entries.

When the displayed list of changes is the desired set, select **write file** in the <Operate> menu and supply the name of a file in which to store the code. That file can then be loaded into another image via the **file in** command in a File Editor or File List.

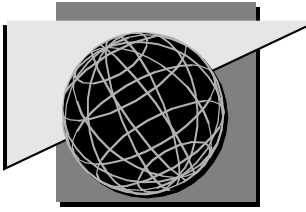
Only the displayed changes are included in a **write file** operation, so if it is possible to define the minimum set of changes by using the filter switches alone, it is not necessary to **remove** and **forget** the nondisplayed entries.

***Note:** When you write selections to a file, be sure to choose a filename that is different from any file that has been read into the change list. The change list maintains pointers to the code in the files that are read in, and these pointers become invalid when you overwrite a file.*

Condensing the Changes File

In a large development effort spanning months or years of programming, the changes file can become very large. To condense it so that it contains only the

most recent change for each method, execute the expression `SourceFileManager default condenseChanges`. Changes involving anything other than a method—such as a class addition or redefinition—will also be purged from the file permanently. It's a good idea to make a backup copy of the changes file before condensing it.



Chapter 16

Accessing Databases

To support the needs of information-intensive applications that rely on database managers, the External Database Interface provides access to relational databases from within a VisualWorks application.

The External Database Interface provides the framework for interacting with relational databases, in the form of a set of protocols supported by several superclasses, but does not provide direct support for any particular database. Database Connect products are available to provide connectivity to specific databases, such as ORACLE and SYBASE.

The examples in this chapter assume that you have installed and configured a VisualWorks database connection according to the instructions provided in the Database Connect's documentation. Using a VisualWorks database connection also requires that the necessary database vendor software be installed and correctly configured.

Overview

Interacting with a relational database involves the following activities:

- n Establishing a connection to the database server
- n Preparing and executing SQL queries
- n Obtaining the results of the queries
- n Disconnecting from the server

The External Database Interface consists of a set of classes that provide uniform access protocol for performing these activities, as well as the other activities necessary for building robust database applications. The classes that make up the External Database Interface are found in the class category **Database-Interface**. Each of these classes is listed below with a more detailed explanation to follow later in the chapter.

Table 16-1 Core External Database Interface Classes

Database Interface Class	Description
ExternalDatabaseConnection	Provides the protocol for establishing a connection to a relational database server, and for controlling the transaction state of the connection.
ExternalDatabaseSession	Provides the protocol for executing SQL queries, and for obtaining their results.
ExternalDatabaseAnswerStream	Provides the stream protocol for reading the data that might result from a query.

In addition to these three core classes, other classes provide useful information.

Table 16-2 Other External Database Interface Classes

Database Interface Class	Description
ExternalDatabaseColumnDescription	Holds the descriptions of the columns of data retrieved by queries
ExternalDatabaseError	Bundles the error information that may result if something goes awry.
ExternalDatabaseFramework ExternalDatabaseBuffer ExternalDatabaseTransaction	Provide behind-the-scenes support for the activities above, and are not accessed directly.
ExternalDatabaseInstallation	Provides the VisualWorks application used to install the database connections, which are available as separate products. Its use is described in the release notes that accompany each connection product.

Data Interchange

Before going further, it is important to understand how relational data is moved into and out of the Smalltalk environment. Data in the relational database environment is stored in tables, which consist of columns, each

having a distinguished datatype (INT, VARCHAR and so on). When a row of data from a relational table is fetched into Smalltalk, the relational data is transformed into an instance of a Smalltalk class, according to Table 16-3.

Table 16-3 Relational Type Conversion

Relational Type	Smalltalk Class
CHAR, VARCHAR, LONG	String
RAW, LONG RAW	ByteArray
INT	Integer
REAL	Double
NUMBER	FixedPoint
TIMESTAMP	Timestamp

NULL values for relational type become the Smalltalk value nil on input, and nil becomes NULL on output.

The row itself becomes either the Smalltalk class `Array` or an instance of some user-defined class. The choice is under your control, and is described later in the chapter.

If a particular DBMS supports additional datatypes, the mapping between those datatypes and Smalltalk classes is explained in the documentation for the corresponding VisualWorks database connection. For example, VisualWorks Sybase Connect supports a datatype called MONEY. The *VisualWorks Database Connect User's Guide for SYBASE* describes how that datatype is mapped to a Smalltalk class.

Establishing a Connection

To establish a connection to a database, you create an instance of `External-DatabaseConnection` (or one of its subclasses), supply it with your database user name, password, and environment (connect) string, then direct it to connect. In the following example we connect to (and then disconnect from) an ORACLE server.

```
| connection |
connection := OracleConnection new.
connection
```

```
username: 'scott';  
password: 'tiger';  
environment: '@T:dbserver:dbname'.  
connection connect.  
connection disconnect.
```

Securing Passwords

In the connection example above, references to the username, password, and environment string are stored in instance variables of the connection object, and will be stored in the image when it is saved. For security reasons, you may wish to avoid having a password stored in the image. A variant of the connect message allows you to specify a password without having the session retain a reference to it. The example below assumes that the class that contains the code fragment responds to the message `askUserForPassword`. The string it answers is used to make the connection.

```
connection  
  username: 'scott';  
  environment: '@T:dbserver:test'.  
connection connect: self askUserForPassword.
```

Getting the Details Right

Environment strings (also called connect strings by some vendors) can be tricky things to remember. As a convenience, `ExternalDatabaseConnection` keeps a class-side registry of environment strings, allowing them to be referenced by logical keys. This enables applications to provide users with a menu of logical environment names, instead of the less mnemonic environment strings.

`ExternalDatabaseConnection` supplies the following class-side messages for manipulating the registry:

```
addLogical: key environment: environment  
removeLogical: key  
mapLogical: key "Return the actual environment for the given key"  
environments "Return the Dictionary of environments"
```

Executing the following example establishes a logical environment named 'test'.

```
OracleConnection
  addLogical: 'test'
  environment: '@T:dbserver:test'.
```

Thereafter, applications that specify 'test' as their environment will actually get the longer ORACLE connect string. Actually, any string that an application provides as an environment is first checked against the logical environment registry. If no match is found, the application's string is used unchanged.

Setting a Default Environment

`ExternalDatabaseConnection` also remembers a default key, allowing applications to connect without specifying an environment. The default key is set by sending `ExternalDatabaseConnection` the message `defaultEnvironment:`, passing the default environment string as the argument. The message `defaultEnvironment` answers with the current default environment, which may be nil.

The following code sets 'test' to be the default logical environment, allowing applications to connect without specifying an environment.

```
ExternalDatabaseConnection
  defaultEnvironment: 'test'
```

Default Connections

In addition to hiding the details of the environment, `ExternalDatabaseConnection` has the notion of a default connection, allowing some applications to be coded without direct references to the type of database to which they will be connected. As an abstract class, `ExternalDatabaseConnection` does not create an instance of itself. Instead, it forwards the new message to the subclass whose name it has remembered as the default. For example, to register `OracleConnection` as the default class to use, execute:

```
ExternalDatabaseConnection defaultConnection:
#OracleConnection.
```

This feature, along with the environment registry explained above, allows the connection example to be rewritten as:

```
| connection |  
connection := ExternalDatabaseConnection new.  
connection  
    username: 'scott';  
    password: 'tiger'.  
connection connect.  
connection disconnect.
```

The default is set initially by the `ExternalDatabaseInstallation` application when the first database connection is installed.

On the Importance of Disconnecting

Establishing a connection to a database reserves resources on both the client, VisualWorks, and the host, database server, side. To ensure that resources are released in a timely fashion, it is important to disconnect connections as soon as they are no longer needed, as shown in the examples above. VisualWorks provides a finalization-based mechanism for cleaning up after a connection if it is “dropped” without first being disconnecting. Since finalization is triggered by garbage collection, the eventual cleanup could take place long after the connection has been dropped. If your application or application environment is resource-sensitive, we recommend proactively disconnecting the connections.

Using Sessions

Having established a connection to a database server, you can then ask the connection for a query session, which reserves the “right” to execute queries using the connection.

A session is a concrete subclass of `ExternalDatabaseSession`, and is obtained from a connected connection by sending the message `getSession`. The connection answers with a session. If the connection is to a Sybase server (i.e., is a `SybaseConnection`), the session will be a `SybaseSession`.

You can ask a session to prepare and execute SQL queries by sending the messages `prepare:`, `execute`, and `answer`, in that order. Depending on the DBMS, `prepare:` will either send the query to the server or defer the send

until the query is actually executed. This is important to note, because errors can be detected (and signals raised) at either `prepare:` or `execute` time.

To examine the results of the query execution, send an `answer` message to the session. This is important to do even when the query does not return an answer set (e.g., an `INSERT` or `UPDATE` query). If an error occurred during query execution, it is reported to the application at `answer` time. More on `answer`, and how it is used to retrieve data, below.

We can extend our connection example to execute a simple query. Note the use of two single quotes around the name. These are needed to embed a single-quote within a Smalltalk String.

```
| connection session |
(connection := ExternalDatabaseConnection new)
  username: 'jones';
  password: 'secret';
  connect.
(session := connection getSession)
  prepare: 'INSERT INTO phonenumber VALUES( "Smith", "x1234" )';
  execute;
  answer.
connection disconnect.
```

As a shortcut, the example above can be simplified somewhat by sending `prepare:` to the connection, which will answer with a prepared session.

```
| session |
session := connection
  prepare: 'INSERT INTO phonenumber VALUES( "Smith", "x1234" )'
;
session
  execute;
  answer;
disconnect.
```

We'll explore getting data back from a query later.

Variables in Queries

Repetitive inserts would be very inefficient if each insert required that a query be prepared and executed. This overhead can be side-stepped by preparing a

single query, with *query variables* as placeholders. This prepared query can then be repeatedly executed with new values supplied for the placeholders.

Query variables (also called parameters) are placeholders for values in a query. Some databases (e.g., ORACLE) produce an execution plan when a query is prepared. Preparing the plan can be expensive. Using variables and binding values to them before each execution can eliminate the overhead of preparing the query for subsequent executions, which can be a substantial performance improvement for some repetitive applications.

To execute a query containing one or more query variables, the session must first be given an input template object, which will be used to satisfy the variables in the query. The method by which values are obtained from the input template depends on the form of the query variable. If the input variable is a question mark, then the input template must either have indexed variables or instance variables. The first template variable will be used to satisfy the value for the first query variable, the second template variable will be used to satisfy the second query variable, and so on. Consider the example:

```
session prepare: 'INSERT INTO phonelist (name, phone) VALUES(?,
?)'.
#( ( 'Curly' 'x47' ) ( 'Moe' 'x29' ) ( 'Larry' 'x83' ) ) do:
  [ :phoneEntry |
    session
      bindInput: phoneEntry;
      execute;
    answer ].
```

Here the input template is an Array with two elements. The first element, the name, will be bound to the first query variable, and the second element, the phone number, will be bound to the second.

A closely related form for query variables is a colon followed immediately by a number. Again, the input template must contain indexed or instance variables, and the number refers to the position of the variable. The query above could be rewritten to use this form of query variable as follows:

```
session prepare: 'INSET INTO phonelist (name, phone) VALUES(:1,
:2)'.
```

Named Input Binding

The third form that a query variable can take is a colon followed by a name. Using this form of binding, the query above would be written as:

```
session prepare: 'INSERT INTO phonelist (name) VALUES(:name)'
```

The name in a query variable represents a message to send to the input template. The input template is expected to answer a value, which will then be bound for the variable. We could use this form of binding in the example above if `PhoneListEntry` included the *accessing* methods

name

```
"Answer the receiver's name"  
^name
```

phone

```
"Answer the receiver's phone number"  
^phone
```

This form of binding is very powerful, but should be used with great care. If the input template does not respond to the message selector formed from the bind variable name, a “Message Not Understood” notifier will result. Also, there are many messages that all objects respond to that would have unexpected effects if used as bind variables, such as `halt`.

Binding NULL

To bind a NULL value to a variable, use the “value” `nil`. This works in general, but causes problems in a particular scenario with ORACLE. The query

```
SELECT name, phone FROM phonelist WHERE name = ?
```

will not work as expected if the variable's value is `nil`. ORACLE requires that such queries be written as:

```
SELECT name, phone FROM phonelist WHERE name IS NULL
```

Getting Answers

Once a database server has executed a query, it can be queried to determine whether the query executed successfully. If all went well, the server is also ready with an answer set, which is accessed by way of an answer stream. Verifying that the query executed successfully and obtaining an answer stream are both accomplished by sending a session the message `answer`.

In responding to `answer`, the session first verifies that the query has finished executing. If the database server has not yet responded, the session will wait. If the server has completed execution and has reported errors, the session will raise an exception. See the “Error Handling” section below for information on the exceptions that might be raised, and details on how to handle them.

If no error occurred, `answer` will respond in one of three ways. If the query is not one that results in an answer set (that is, an `INSERT` or `UPDATE` query), `answer` will respond with the symbol `#noAnswerStream`. If the query resulted in an answer set (that is, a `SELECT` query), `answer` will return an instance of `ExternalDatabaseAnswerStream`, which is used to access the data in the answer set, and is explained below.

The third possible response to `answer` is the symbol `#noMoreAnswers`. When a database supports multiple SQL statements in one query, or stored procedures that can execute multiple queries, you can send `answer` repeatedly to get the results of each query. It will respond with either `#noAnswerStream` or an answer stream for each, and will eventually respond with the symbol `#noMoreAnswers` to signify that the set of answers has been exhausted.

Handling Multiple Answer Sets

If your application is intended to be portable and support ad hoc queries, we recommend that you send `answer` repeatedly until you receive `#noMoreAnswers`. For example, Sybase stored procedures can return multiple answer sets. The following code fragment shows how to retrieve the answer sets that might result from executing a Sybase stored procedure.

```
session
  prepare: 'exec get_all_phonenumbers';
  bindOutput: PhoneEntry new;
  execute.
connection class externalDatabaseErrorSignal
  handle:[ :ex | Dialog warn: ex parameter first dbmsErrorString ]
```

```

do:[ | answer |
    [ numbers := OrderedCollection new,
      (answer := session answer) == #noMoreAnswers]
    whileFalse: [ answer == #noAnswerStream
      ifFalse: [numbers := numbers , (answer upToEnd) ] ]
].

```

More information on managing Sybase stored procedures can be found in the *VisualWorks Database Connect User's Guide for SYBASE*.

What Happens when you Send an Answer Message

When you send `answer` to a session, a number of things happen in the background as the session prepares the resources needed to process an answer set. Most of these steps are out of the direct view of the application. However, an understanding of them may help when you are debugging database applications.

To answer a query, the session performs the following steps:

1. Waits for the server to complete execution.
2. Verifies that the query executed without error.
3. Determines whether an answer set is available.

If the query returns an answer set, then the session performs the following additional steps:

4. Obtains a description of the answer set.
5. Allocates buffers to hold rows from the answer set.
6. Prepares adaptors to help translate relational data to Smalltalk objects.

Waiting for the Server

Some database servers, such as Sybase, support asynchronous query execution, giving control back to the application after the server has begun executing the query. To determine whether the server has completed execution, a session sends itself the message `isReady`, which returns a Boolean indicating that the server is ready with an answer, until `isReady` returns true. If the target DBMS does not support asynchronous execution (for example, ORACLE), `isReady` will always return true.

Did the Query Succeed?

The session next verifies that the query executed without error. Errors that the server reports are bundled into instances of `ExternalDatabaseError` (or a Connection-specific subclass). A collection of these errors is then passed as a parameter to an exception. See “Error Handling” on page 208 for more details.

How Many Rows were Affected?

Some queries, such as `UPDATE` or `DELETE`, do not return answer sets. To determine how many rows the query affected, send the message `rowCount` to the session, which will respond with an integer representing the number of rows affected by the query. Because database engines consider a query to have executed successfully even if no rows were matched by a `WHERE` clause, testing the row count is an easy way to determine whether an `UPDATE` or `DELETE` query had the desired effect.

Database-specific restrictions on the availability of this information are documented in the release notes for your database-connect product.

Describing the Answer Set

If the query has executed without error, the session determines whether the query will return an answer set.

If the session returns an answer set, the session will obtain from the server a description of the columns in the set. Sending the message `columnDescriptions` to the session (after sending `answer`) will return an `Array` of instances of `ExternalDatabaseColumnDescription` (or a connection-specific subclass), which describes the columns in the answer set.

A column description includes: the name, length, type (expressed as a Smalltalk class), precision, scale, and nullability of a column. A column description will respond to the following *accessing* protocol messages:

name	"Answer the name of the column"
type	"Answer the Smalltalk type that will hold data from the column"
length	"Answer the length of the column"
scale	"Answer the scale of the column, if known"
precision	"Answer the precision of the column, if known"
nullable	"Answer the nullability of the column, if known"

Connection-specific subclasses may make additional information available.

Note that the names returned for calculated columns may be different depending on the target DBMS. For example, the query

```
SELECT COUNT(*) FROM phonelist
```

determines the number of rows in the phone list table. ORACLE names the resulting column "COUNT(*)", while Sybase does not provide a name.

Buffers and Adaptors

Finally, the session uses the column descriptions to allocate buffers to hold rows of data from the server, and adaptors to help create Smalltalk objects from the columns of relational data that will be fetched from the server into the buffers. This step is invisible to user applications, but can be the source of several errors. For example, if insufficient memory is available to allocate buffers, an `unableToBind` exception will be raised. An `invalidDescriptor-Count` exception will be raised if the output template (explained below) doesn't match the column descriptions.

Processing an Answer Stream

After the session has completed the steps above, and assuming that the query results in an answer set, the session creates an `ExternalDatabaseAnswerStream` and returns it to the application. `ExternalDatabaseAnswerStream` is a subclass of `Stream`, and is used to access the answer set. It responds to much of the standard streaming protocol described in the VisualWorks Cookbook. There are a few restrictions. Answer streams are not positionable, they cannot be flushed, and they cannot be written.

Answer streams are created by the session; your application should not attempt to create one for itself.

Answer streams respond to the messages `atEnd`, for testing whether all rows of data from an answer set have been fetched, and `next` for fetching the next row. Attempting to read past the end of the answer stream results in an `endOfStreamSignal`.

In our example, all rows of the phone list could be fetched as follows:

```
numbers := OrderedCollection new.  
answer := session answer.
```

```
[answer atEnd] whileFalse:  
  [ | row |  
    row := answer next.  
    numbers add: row ].
```

Sending `upToEnd` causes the answer stream to fetch the remaining rows of the answer set and return them in an `OrderedCollection`. Using `upToEnd`, the example above can be simplified as:

```
answer := session answer.  
numbers := answer upToEnd.
```

While this works well for small answer sets, it can exhaust available memory for large answer sets.

Unless the session has been told otherwise, data retrieved through the answer set comes packaged as instances of the class `Array`.

Using an Output Template

Having rows of a table (or columns from a more complex query) arrive packaged as instances of the class `Array` might suffice for some applications. For more complex applications, it is preferable to have the data appear as instances of some user-defined class. In our example, we would want rows of data fetched from the `phonelist` table to appear as instances of class `PhoneListEntry`.

To achieve this, `ExternalDatabaseSession` supports an output template mechanism. If an output template is supplied to the session, it will be used instead of the class `Array` when creating objects to represent rows of data in the answer set. In our example, this would look like:

```
session  
  prepare: 'SELECT name, phone FROM phonelist';  
  bindOutput: PhoneListEntry new;  
  execute.  
answer := session answer.
```

Rows of data from the table will now appear (by sending `answer next`) as instance of `PhoneListEntry`.

Columns of data from a row of the answer set are loaded into the output template's variables by position. Column 1 loads into the first variable, column 2 loads into the second variable, and so on. The output template can have either instance variables or indexed variables. When both are present, the indexed variables are used.

Skipping Slots in an Output Template

To skip a variable in the bind template, place an instance of the class `Object` in it. There must be exactly as many non-Object variables in the output template as there are columns in the answer set. For example, consider the scenario of having the additional instance variable `unused` in an instance of `PhoneListEntry`. If this instance variable is not fetched from the database, you could add the method

newForSelect

```
"Create a new instance of the receiver,  
and initialize it to be fetched from the database."
```

```
^super new initializeForSelect
```

to the *instance creation* protocol on the class side of `PhoneListEntry`, and

initializeForSelect

```
"Initialize an instance of the receiver to be fetched from the  
database."
```

```
unused := Object new.
```

to the *initialize-release* protocol on the instance side. This allows us to safely rework the example above by writing

```
bindOutput: PhoneListEntry newForSelect;
```

to specify the output template.

Using Column Names to Bind for Output

As with input binding, a name-based alternative is provided for output binding. Sending a session the message `bindOutputNamed:`, with the output template as an argument, causes the session to create a set of mutator

messages to send to the output template to store values fetched from the database. These mutator messages are formed by appending colons to the column names. Our phone list example could use named output binding if the class `PhoneListEntry` provided the following instance-side *accessing* methods:

name: aName

"Set the phone entry's name"

name := aName

phone: aPhoneNumber

"Set the phone entry's phone number"

phone := aPhone

The same caveats apply to named output binding as apply to named input binding. If the output template does not answer the message, a “Message Not Understood” notifier will result. Be sure that the needed method names do not override methods that are necessary for the functioning of the object.

Reusing the Output Template

By default, a new copy of the output template is used for each row of data fetched. If your application processes the answer set one row at a time, the overhead of creating a copy can be eliminated by arranging to reuse the original output template. Sending `allocateForEachRow: false` to the session tells it to reuse the template. Output template reuse is temporarily disabled when sending `upToEnd` to the answer stream.

Setting a Block Factor to Improve Performance

Some database managers allow client control over the number of rows that will be physically transferred from the server to the client in one logical fetch. Setting this blocking factor appropriately can greatly improve the performance of many applications by trading buffer space for time (network traffic).

If our phone list database resided on an ORACLE server, our example might be greatly improved by sending the message `blockFactor:` to the session, as follows:

```
session
```

```
  prepare: 'SELECT name, phone FROM phonenumber';  
  bindOutput: PhoneListEntry new;  
  blockFactor: 100;  
  execute.
```

Since the phone list entries are small, asking for 100 rows at a time is not unreasonable.

Note that the block factor does not affect the number of objects that will be returned when you send the message `next` to the answer stream. Objects are read from the stream one at a time.

If a database connection does not support user control over blocking factors (as with Sybase), the value passed to `blockFactor:` is ignored, and the value remains set at 1. Additional restrictions on the use of `blockFactor:`, if any, are listed in the release notes for your Database Connect product.

Canceling an Answer Set

If your application finishes with an answer stream before reaching the end of the stream (perhaps you only care about the first few rows of data), it is good practice to send the message `cancel` to the session. This tells the database server to release any resources that it has allocated for the answer set. The answer set will be automatically canceled the next time you prepare a query, or when the session is disconnected, but a proactive approach is often preferable.

Disconnecting the Session

Establishing a session reserves resources on the client side, and often on the server side. When you're done with a session, sending the message `disconnect` to the session disconnects it and releases any resources that it held. The connection is not affected. A disconnected session will be automatically reconnected the next time a query is prepared. If you expect your application to experience long delays between queries, you might consider disconnecting sessions where possible.

Sessions will automatically disconnect when their connection is disconnected. Sessions are also protected by a finalization executor, and will be disconnected, eventually, after all references to them are dropped.

Catalog Queries

To simplify access to a database's catalog, `ExternalDatabaseSession` provides a few methods that hide the details of the particular database vendor's catalog structure.

To obtain a list of available tables, send a session the message `listTables`. To get a subset of the available tables, send `listTablesLike:`, with a `String` argument containing an SQL wildcard, as in:

```
"Get a list of available tables."  
tables := session listTables.
```

```
"Get a list of all tables that begin with 'PHONE'  
tables := session listTablesLike: 'PHONE%'.
```

```
"Get a list of all tables owned by PUB2"  
tables := listTablesList: 'PUB2.%'.
```

Each element in the resulting collection is an instance of the class `String`.

***Note:** The availability of a table does not mean that the application has the necessary permissions to access the table.*

To obtain a description of the columns in a table, send a session the message `describeColumns:`, with the table name as an argument.

```
columns := session describeColumns: 'phonest';
```

Each element in the resulting collection is an instance of `ExternalDatabaseColumnDescription`.

The catalog query messages may cause a query to be prepared and executed using the session, and might also affect a session's input and output templates. If you reuse the session, you will have to establish new input and output templates, if desired.

Controlling Transactions

By default, every SQL statement that you prepare and execute is done within a separate database transaction. To execute several SQL statements within a single transaction, send **begin** to the connection before executing the statements, followed by **commit** after the statements have executed. To cancel a transaction, send **rollback** to the connection.

The connection keeps track of the transaction state. If an application bypasses the connecting by preparing and executing SQL statements like **COMMIT WORK** or **END TRANSACTION**, the connection will lose track of the transaction state. This might lead to later problems.

Coordinated Transactions

Several connections can participate in a single transaction by appointing one connection as the coordinator. Before the connections are connected (that is, sent **connect** or **connect:**), send the coordinating connection the message **transactionCoordinatorFor**: once for each participating connection, passing the connection as the argument.

After the coordination has been established, sending **begin** to the coordinator begins the coordinated transaction. Sending **commit** or **rollback** to the coordinator causes the message to be broadcast to all dependent connections.

If the database system supports two-phase commit, the coordination assures the atomic behavior of the distributed transaction. If the database does not support two-phase commit, a serial broadcast is used.

Participants in a coordinated transaction must be supported by a single-database connection. It is not possible, for example, to mix **ORACLE** and **Sybase** connections in a coordinated transaction.

Releasing Resources

If your application has relatively long delays between uses of the database, you may want to release external resources during those delays. To do so, send a **pause** message to any active connections. This causes the connections to disconnect their sessions, if any, and then disconnect themselves. Any pending transaction is rolled back. Both the connections and their sessions remain intact, and can be reconnected.

To revive a paused connection, send it **resume**. The connection will then attempt to re-establish its connection to the database.

Note: If the password was not stored in the connection, as discussed under “Securing Passwords” on page 190, the `proceedable exception required-PasswordSignal` will be raised.

Sessions belonging to resumed connections will reconnect themselves when they are prepared again.

Sending `pause` or `resume` to `ExternalDatabaseConnection` has the same effect as sending `pause` or `resume` to all active connections.

Tracing the Flow of Execution

A tracing facility is built into the VisualWorks database framework, and is used by database connections to log calls to the database vendors’ interfaces. Enabling this facility can be quite useful if your application’s use of the database malfunctions.

A trace entry consists of a time stamp, the name of method that requested the trace, and an optional information string. Database connections use this string to record the parameters passed to the database vendor’s interface routines, and the status or error codes that the interfaces return. This information can be invaluable when tracking down database problems.

Directing Trace Output

To direct tracing information to the System Transcript window, execute the following expression in a workspace (or as part of your application):

```
ExternalDatabaseConnection traceCollector: Transcript
```

To direct tracing into a file, execute the following:

```
ExternalDatabaseConnection traceCollector: 'trace.log' asFilename  
writeStream
```

Setting the Trace Level

The framework supports the following of levels of tracing. The default trace level is zero.

Table 16-4 Trace Levels

Trace Level	Description
0	Disables tracing.
1	Limits the trace to information about connection and query execution.
2	Adds additional information about parameter binding and buffer setup.
3	Traces every call to the database.

The trace level is set by executing:

```
ExternalDatabaseConnection traceLevel: anInteger
```

Disabling Tracing

Setting the trace level to 0 disables tracing.

Adding Your Own Trace Information

To intermix application trace information into the trace stream, place statements like

```
ExternalDatabaseConnection trace: aStringOrNil
```

in your application. An argument of `nil` is equivalent to an empty string; only a time stamp and the name of the sending method will be placed in the trace stream.

You can avoid hard-coding the literal name `ExternalDatabaseConnection` by asking a connection for its class, and sending the trace message to that object, as in:

```
connection class trace: ('Made it this far ', count printString , ' times').
```

See the *tracing* protocol on the class side of `ExternalDatabaseConnection` for additional information.

Error Handling

Error handling in the VisualWorks database framework is based on signals and exception handlers.

For practical purposes, the set of errors that a database application might encounter can be divided into two groups.

The first group is state errors, and these errors normally occur when an application omits a required step or tries to perform an operation out of order. For example, an application might attempt to answer a query before executing it. If the application is coded correctly, these kind of errors generally do not get generated.

The second group is execution errors, and they get generated when an application performs a step in the correct order, but for some reason the step fails.

When either type of error is encountered, an exception is signaled and any available error information is passed as a parameter of the signal. The application is responsible for providing exception handlers and recovery logic.

Signals and Error Information

The database framework provides a family of signals, most of which are based on the common parent `externalDatabaseErrorSignal`, which is defined in the *signal constants* protocol on the class side of `ExternalDatabaseFramework`. “The Database Signal Hierarchy” on page 209 describes signals in more detail.

If a signal is the result of a database error, the connection code that sends the signal to an exception handler first collects the available database error information into instances of `ExternalDatabaseError`, and then passes the information as a parameter of the signal. If the signal results from a state error, the signal is sent without additional information.

An instance of `ExternalDatabaseError`, or a connection-specific subclass, stores a database-specific error code, and, when available, includes the string that describes the error. The error code is retrieved by sending a database error the message `dbmsErrorCode`, and to get the string the message `dbmsErrorMessage` is sent. See the `ExternalDatabaseError` *accessing* protocol for additional information.

Exception Handling

The example below shows one way to provide an exception handler. The handler is for the general-purpose database exception `externalDatabaseErrorSignal`. If this exception, or one of its children, is signaled from the statements in the `do:` block, the `handle:` block is evaluated. In this example, the `handle:` block extracts the error string from the first database error in the collection that was passed as a parameter to the exception handler, and uses this string in a warning dialog.

```

connection class externalDatabaseErrorSignal
  handle: [ :ex | "If the query fails, display the error string in an OK
dialog"
  Dialog warn: ex parameter first dbmsErrorString ]
  do: [
  session
    prepare: 'SELECT name, phone FROM fonelist';
    execute.
    answer := session answer ].

```

In this example, the error is caused by the invalid table name in the query. If the connection in this example is to an ORACLE database, the database error in the collection passed to the handler (that is, the database error accessed by `ex parameter first`), will be an instance of `OracleError`, and will hold as its `dbmsErrorCode` the number 942, and as its `dbmsErrorString` the string `ORA_00942: table or view does not exist`.

The Database Signal Hierarchy

The hierarchy of signals that the database interface provides is found in the *signal constants* protocol on the class side of `ExternalDatabaseFramework` and its subclasses.

The two parent signals in the hierarchy are `externalDatabaseErrorSignal` and `externalDatabaseInformationSignal`. The error signals generally represent failures, which prevent continuation, while the information signals represent errors that a handler can recover from by sending `proceed` or `proceedWith:` to the exception.

A few of the signals are of special interest.

The signals `invalidConnectionState` and `invalidSessionState` are raised when a state violation is encountered in a connection or session, respectively.

These signals indicate that the application has performed an operation out of order.

The signal `externalDatabaseLibraryInaccessibleSignal`, which results in a “Database API libraries inaccessible” notifier, is a signal that is often encountered in the early stage of database application development. This signal is raised if a connection determines that the Smalltalk Object Engine cannot access the required database vendor’s libraries. On Windows and OS/2 platforms, this is typically caused by not having the required DLLs in the search path. On UNIX and Macintosh platforms, this is usually caused by running an Object Engine that was not linked with the required vendor libraries. If you experience this signal, double check the VisualWorks database connection documentation to verify that you are running the correct Object Engine, and that any software required by the database vendor is present and configured.

Choosing an Exception to Handle

With the wealth of exceptions that might be signaled, which ones should an application provide handlers for? The answer, as with many of life’s difficult questions, is “it depends.” For many applications, it only matters if a query “works.” In this case, providing a handler for `externalDatabaseErrorSignal` is usually sufficient. Other applications might be more sensitive to specific types of errors, and will want to provide more specific handlers.

Unfortunately, the use of exception-specific handlers is complicated by the fact that the errors that the low-level database interface reports may at first appear to be unrelated to the operation being performed. For example, the connection to a remote database server can be interrupted at any time, but the exception signaled will depend on the database activity that the application was performing at the time the problem was detected.

The recommended strategy is to provide a handler for as general a signal as you feel comfortable with (for example, `externalDatabaseErrorSignal`), and invest effort, if necessary, in examining and responding to the database-specific errors that will be passed to the handler. We recommend against providing a completely general handler (for example, for `Object errorSignal`), especially during development, as this will make nondatabase problems more difficult to isolate.

Image Save and Restart Considerations

When an image containing active database connections is exited, the connections are first **paused**, and any partially completed transactions are terminated via **rollback**.

To arrange for your application to perform some set of steps before the transaction is terminated, your application model must first register as a dependent of the class `ExternalDatabaseConnection`. For example:

```
ExternalDatabaseConnection addDependent: self.
```

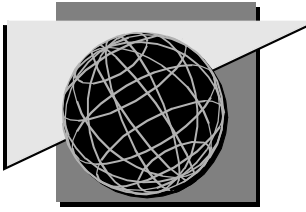
The application model then creates an `update:` (or `update:with:`) method, and tests for the `update:` argument `#aboutToQuit`. For example:

```
update: anAspectSymbol with: aValue  
    anAspectSymbol == #aboutToQuit  
    ifTrue:[ "perform desired action." ].
```

Reconnecting When an Image is Restarted

When an image is restarted, all references to external resources are initialized, as if a `pause` message had been sent to the class `ExternalDatabaseConnection`. To arrange for your application to take further action, take the steps described above, testing for the `update:` argument `#returnFromSnapshot`.

Your application can reconnect its connections by sending them `connect` (or `connect:` with a password). This re-establishes the connection to the database server (subject to the constraints discussed in “Releasing Resources” on page 205). Any sessions will need to be re-prepared by sending the sessions `prepare:` with the query to prepare, though your application might as easily drop the old sessions and get new ones.



Chapter 17

Troubleshooting

This chapter lists exceptional conditions you may encounter, and suggests remedies. Because Smalltalk lets you modify fundamental system classes such as `Object`, it is fairly easy to cause errors in system operation—though not if you exercise caution when changing a system class. For example, removing the `Object` class would not be a good idea.

Recovering from a System Failure

The best defense against the unforeseen is to use the VisualWorks main window's **File?Save As** command to make a snapshot of your image frequently. The Change List provides a second line of defense.

After a power outage or other system failure, open the image and the project in which you were working at the time of the crash. Open a Change List by selecting **Changes?Open Change List** in the VisualWorks main window.

Display the changes made since the last snapshot by selecting **recover last changes** in the list view's <Operate> menu. Edit the displayed list, if you want to cull unnecessary entries such as doIts and any changes that may have contributed to the system failure. For example, to delete all doIts, which are usually unnecessary for recovery purposes, select any doIt in the list view and select the **type** switch. Choose **remove all** in the <Operate> menu to mark all the doIts for deletion, then select **forget** to erase them. Then deselect the **type** switch to display the remaining changes.

When the displayed list contains the desired changes, select **replay all** in the <Operate> menu.

If you made changes in more than one project since the last snapshot was saved, you may want to perform this recovery operation separately for each project's changes. That way, the Change Set associated with each project will be updated correctly. For more information about the Change List, see "Reverting to a Prior Version" on page 182.

Start-up Errors

If the command line that is used to start VisualWorks is incorrect, one of the errors listed in Table 17-1 will result. Fix the problem as described and then try the start-up again. The errors are listed in alphabetical order.

Table 17-1 Start-up Errors

Error Message	Description
Can't open file 'filename.'	The image file named on the command line doesn't exist.
Can't load image.	Your image file may not be as long as its header claims it is (that is, either the header was damaged or the image file was truncated).
Your image file is not compatible with the virtual machine.	You are probably trying to run an obsolete image or a file that is not an image.
Insufficient memory to allocate heap.	Your machine lacks the necessary swap space or physical memory to load the image. If other processes are tying up memory, try removing some of them.
No image filename supplied.	The command line must include the name of an image file.
Option 'x' (option-name) value should be between low and high.	An illegal value was supplied for a command-line option.
OS initialization error, sorry.	Some kind of operating system resource is unavailable.
Unable to read the image file.	No read permission for the image file.
usage: virtualMachineFilename options] imageFilename	Any command-line syntax error.

Source Code Unavailable in Browser

If the sources file (named `st80.sources` by default) is moved to a nonstandard directory, you must make a new snapshot that recognizes its actual location.

To do so, open a Settings Tool to the **Sources** page and insert the pathname for the sources file in the appropriate field, and **Accept** the new setting. Then make a new snapshot by choosing **File?Save As...** in the VisualWorks main window.

Low Space

When a low-space notifier warns that the system is running out of memory space, close any unneeded windows to free up memory resources. Then select **File?Collect Garbage** in the VisualWorks main window. For more information about the system's memory management facilities, see the chapter "Memory Management."

No VisualWorks Main Window

If you close the VisualWorks main window, you no longer have access to the usual means of opening new tools, saving your image and quitting from VisualWorks.

You can close the VisualWorks main window from a Workspace, a System Browser or a System Transcript. To do this, execute the following expression to start a new VisualWorks main window:

```
LauncherView openLauncher
```

You can then continue working.

Can't Exit from VisualWorks

If you find that you are unable to exit out of VisualWorks (the **File?Exit VisualWorks** command does not work), use the facilities provided by your operating system and window manager. These facilities are explained in the following sections.

UNIX

If you started VisualWorks from a shell, type `<Control>-c` in that shell.

If you started by selecting a menu item or clicking an icon in your windowing system, you will have to kill the process as follows:

1. In a shell, enter the following to list the active processes:

```
% ps ax
```

2. Substitute the PID (process identification number) associated with the VisualWorks process in the following command:

```
% kill -term PID
```

3. If that doesn't work, use the stronger but less graceful version of the command:

```
% kill -kill PID
```

Macintosh

Select **Quit** in the Macintosh menu bar at the top of the screen.

Windows

While holding down the <Shift> and <Control> keys, select the Close option on the system menu of the VisualWorks main window.

Emergency Exit (all platforms)

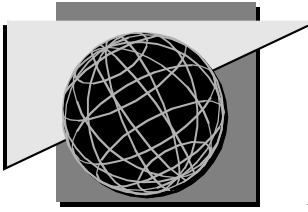
If Smalltalk stops responding to inputs such as mouse movements, try typing the program interrupt, <Control>-c. If that doesn't work, the system provides an Emergency Evaluator, which can be used to execute **Smalltalk quit** even when much of the system is in an unusable state. To open an Emergency Evaluator, type <Shift>-<Control>-c. That is, hold down both the <Shift> and <Control> keys while you press the c key.

An Emergency Evaluator window will appear, with instructions to type a Smalltalk expression terminated by <Escape>. Enter **Smalltalk quit** in the window, then press <Escape>. The system will shut down, after which you can restart it.

When You Need Assistance

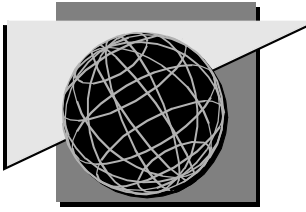
ParcPlace-Digitalk provides technical support to customers who have purchased the ObjectSupport package. VisualWorks distributors often provide similar services. When you need to contact a technical support representative, please be prepared to provide the following information:

- n The *version id*, which indicates which version of the product you are using. Choose **Help?About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**
- n Any modifications (*patch files*) distributed by ParcPlace-Digitalk the you have imported into the standard image. Choose **Help?About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**
- n The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.



Part III

Application Components



Chapter 18

Application Framework

A set of objects that collaborate to solve a problem can be called an *application*, as in “an application of computer technology to the problem.” Most applications, regardless of what problems they solve, have certain functions in common. For example, most applications accept input from the user and respond by performing an action, such as modifying data.

It would be wasteful to duplicate the shared mechanisms in each new application. VisualWorks provides a set of classes from which your application can inherit these foundation mechanisms. This set of classes is called an *application framework*, because it is much like a framework for a home to which you fasten your unique choice of siding, wall board, roofing, flooring, doors, windows, lighting and paints.

As with any object-oriented construct, the application framework consists of objects that provide services to collaborating objects. This chapter describes the application framework, discussing each kind of object, each major service it provides, the clients for that service, and the ways your applications can make use of the resulting mechanism.

This chapter concentrates on the operational mechanisms of the application framework. For practical instructions in applying these mechanisms, see the *VisualWorks Cookbook*.

First, a brief overview of the framework classes, as background for the mechanism descriptions.

Overview

Domain Model Is Separate From User Interface

An application typically begins with one or more *domain models*, which define the structure and processing of data in the domain of the application.

For example, in a sketching application, the domain model is responsible for storing the lines that make up the sketch, and for adding and removing lines upon request.

Instances of a domain model are the objects that the application is primarily concerned with creating, modifying, storing and destroying. The *user interface (UI)* is the part of the application that enables a user to control this activity by using mouse and keyboard actions. The UI consists of a window containing *widgets*—user controls such as buttons, input fields and lists.

The first and most fundamental aim of the application framework is this: Keep the domain model separate from the user interface.

This separation of domain model from UI makes the application easier to maintain, and also promotes reusability of the application components. If the domain model provides generic services rather than services that rely on special knowledge about a particular UI, it is easier to substitute a different interface later as UI technology and user needs evolve.

This separation also makes it easier to provide multiple UIs that employ the same domain model. For example, a novice user and an expert user can employ entirely different UIs to interact with the same domain model.

Similarly, keeping the UI components free of special knowledge about a particular domain model makes it possible to reuse those components in different applications.

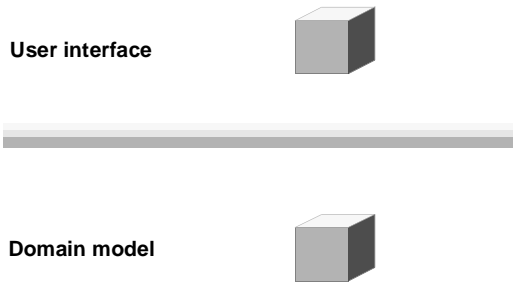


Figure 18-1 Separation of UI from domain model

ApplicationModel Acts as Mediator

Obviously, a user interface has to mirror the domain model to a large degree. So how can windows and widgets that know nothing of a particular domain

model, and a domain model that knows nothing of them, collaborate successfully to form a unified application?

The answer is a mediating object, which translates the UI's generic requests for data and operations into specific messages to the domain model.

For example, an input field asks this mediating object for its data value when it is first displayed, and the mediator is responsible for knowing which data-accessing message to send to the domain model.

Similarly, a menu in the user interface notifies the mediator when the user has selected, say, the third menu item. The mediator is responsible for knowing which operation to request from the domain model, or from some other component of the application.

This mediating object is called an *application model*, because it defines relations between parts of an application much as a domain model defines relations between items of information.

Besides creating domain models, the primary activity in creating a VisualWorks application is defining a custom subclass of `ApplicationModel` to act as mediator. This subclass is typically generated during the **Install** stage of user-interface creation. The UI components are typically reused without modification, though their properties, such as size and color, can be set graphically.

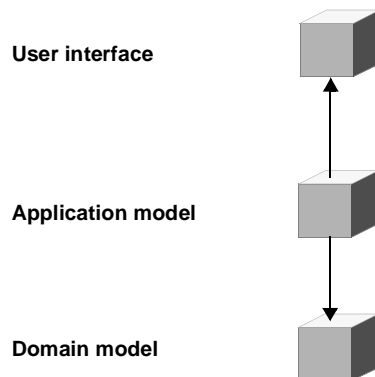


Figure 18-2 Application model as mediator between UI and domain model

Value Model Links Widget to Attribute

An application model links a domain model to a user interface by linking individual components of those larger objects.

Each component of a UI is a widget. Each component of a domain model is an attribute or an operation. Each widget modifies an attribute or starts an operation. The application model defines what each widget does.

For attribute-setting widgets, the application model employs an adaptor to translate generic value-getting and value-setting messages into specific messages to the domain model. This adaptor is called a value model, because it defines the relation between an attribute's value and widgets that depend on that value.

There are different kinds of value models for different kinds of attribute values. For example, a `ValueHolder` is used when the attribute value is a simple data value such as a string of characters, while an `AspectAdaptor` is used when the simple data value is embedded in a composite attribute, such as a `BankAccount`.

A value model is typically generated during the **Define** stage of user-interface creation.

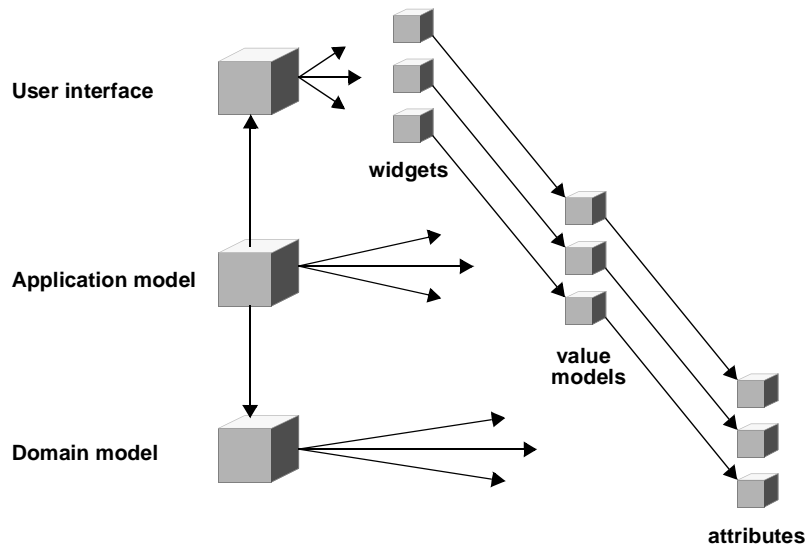


Figure 18-3 A value model links a widget to an attribute. Classically, the attributes are components of the domain model, as shown here, but they can also be components of the application model. Supplying value models is how an application model mediates between UI components and domain model components.

Builder Assembles User Interface

The process of defining a user interface involves painting widgets on a window canvas. Then, various properties for each widget are defined, including its size, location, label, and value name or operation name. These properties are captured in a widget specification object, or *spec* for short.

When the canvas is installed, the window and widget specs are stored in a class method in the application model, called a spec method. When the application is started, the specifications are used to assemble actual widget objects in a running `ApplicationWindow`.

The application model delegates this specification-capturing and interface-building activity to an instance of `UIBuilder`. This *builder* object is a valuable source of information about the interface. For example, you can programmatically access a specific widget by asking for it by name from the builder.

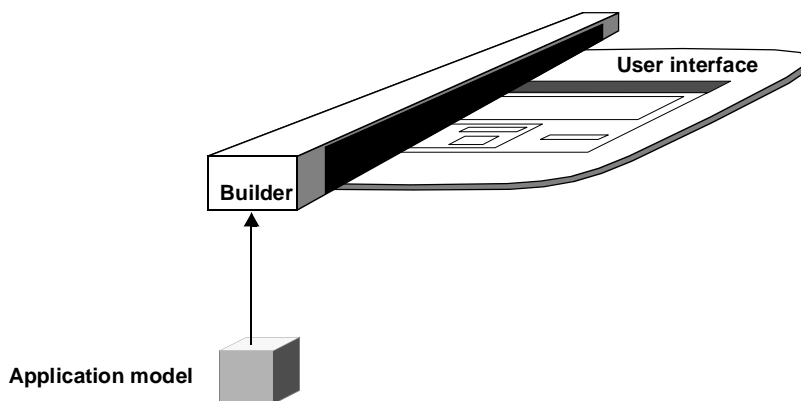


Figure 18-4 The application model delegates the task of constructing a window and its widgets to a `UIBuilder`. The builder works from a set of specifications provided by the application model class, in the form of a `spec` method.

Widget Has Visual Component and Optional Controller

Each widget in a user interface is either passive or active. A passive widget merely displays something, while an active widget both displays something and responds to mouse or keyboard activity. For example, a label widget is passive while an input field is active.

The responsibility for displaying something is performed by a *visual component*, also called a *view*, while the responsibility for responding to user input is performed by a *controller*. The motivation for separating these responsibilities in different objects is reusability.

For example, a radio button and an action button have quite different appearances but both respond to a mouse click by triggering a response from the application model. Their visual components must be different, but they can use the same kind of controller.

Creating a custom widget involves defining a custom visual component for it, and then choosing an existing controller or defining a custom controller. A special containing widget called a view holder enables you to integrate this custom view-controller pair into a user interface as peers of the standard widgets on the VisualWorks Palette.

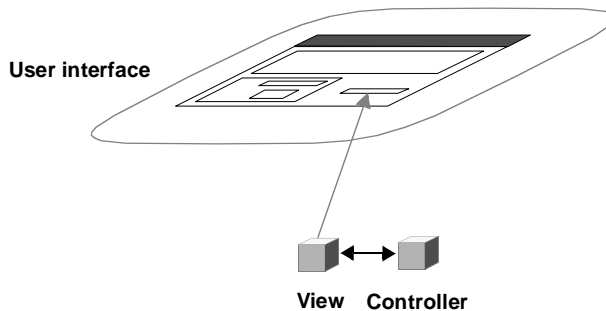


Figure 18-5 Each widget consists of a view for displaying an aspect of the model, and an optional controller for responding to mouse and keyboard activity within that view's boundaries.

About the Example Application

To demonstrate the principles described in this chapter, an example application is supplied with VisualWorks. The application was primarily created to demonstrate the creation and integration of a custom view and custom controller, so it is called `CustomView1Example`.

A second version of the application, called `CustomView2Example`, appears and behaves identically but employs a different kind of controller (event-driven). During the discussion of controllers, the distinctions between `CustomView1Example` and `CustomView2Example` will be explored.

The two versions of the application rely on a set of classes that reside in files (not in the standard VisualWorks image) and must be loaded into your image.

Loading the example classes

1. Open an Online Documentation window by clicking on the **Help** icon in the icon bar of the main VisualWorks window.
2. In the Online Documentation window, click on the **File** menu and select **Browse Example Class**. A dialog will list the available example applications. (If none are listed, make sure the **Help** page of the Settings Tool shows the correct pathname of the `visual/online` directory or folder.)
3. In the dialog, select `CustomView1Example` in the list and then click on **OK**. A dialog will confirm your intention to file in the example classes.

- Repeat steps 2 and 3 for the CustomView2Example class.

Components

The example application is a rudimentary sketching utility that enables you to start a new sketch, draw lines in the sketch, erase lines, and switch among the sketches you have created. The example application does not provide for storing the sketches after the application is closed.

The example classes include a domain model (*Sketch*), an application model (*CustomView1Example*), a custom view (*SketchView1*) and a custom controller (*SketchController1*). The event-driven classes—*CustomView2Example*, *SketchView2* and *SketchController2*—serve parallel functions.

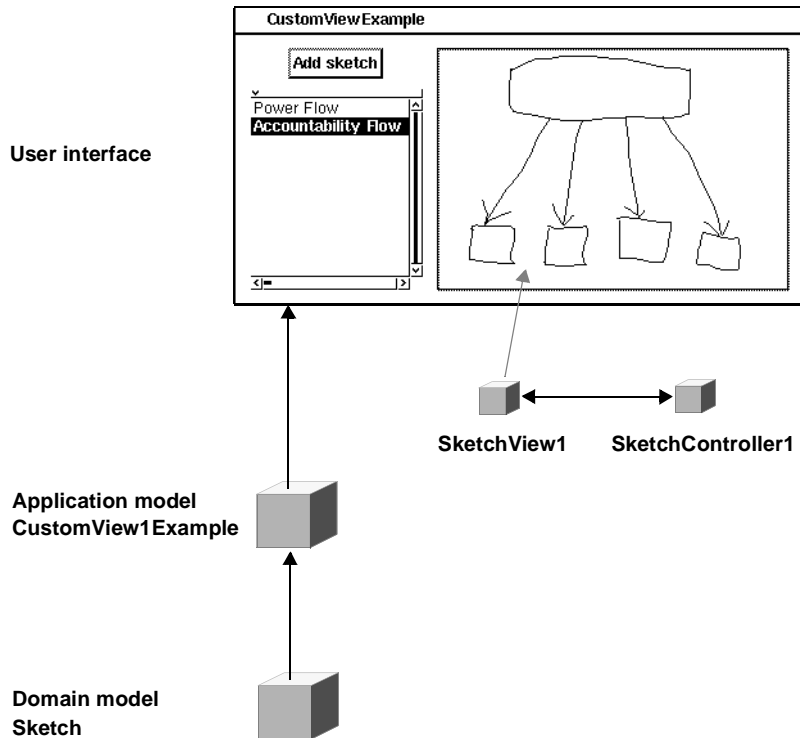


Figure 18-6 How the example classes fit into the framework.

Domain Model

Overview

A domain model is typically the first class you create when developing a new application, because the application model and user interface rely heavily on the domain model. A System Browser is used to create a domain model.

A domain model has two essential responsibilities: storing data and providing data-processing operations. Because it does not have to supply any complicated mechanisms, there is no application-framework support for a domain model. It is typically a subclass of `Object`, or of an existing domain model class.

Multiple Domain Models

In all but the simplest of applications, the information is subdivided among a set of related domain models. For example, in a banking application, a customer model stores customer information, an account model stores bank-account information, a transaction model stores banking-transaction information, and so on.

Deciding how to subdivide information among a set of domain models is the subject of the analysis phase of a development project.

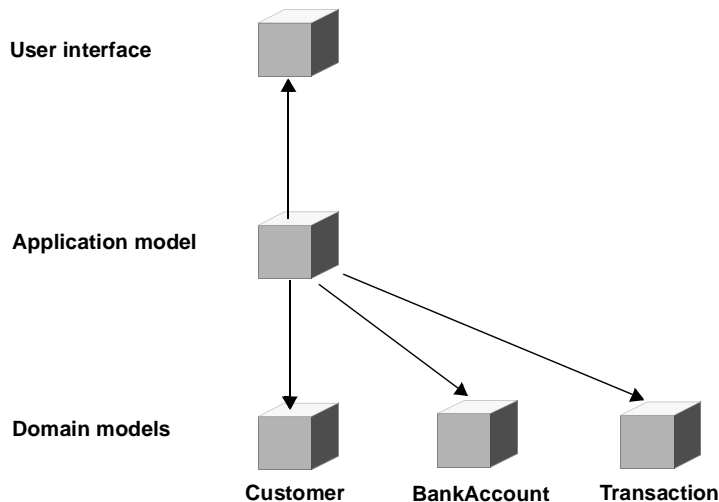


Figure 18-7 Domain information is often divided among multiple classes.

Data Storage

One responsibility of a domain model is to hold the information with which an application is concerned. In the example application, a **Sketch** holds a collection of sketched lines as well as a name for the sketch.

How Data Is Stored

Typically, each attribute of the object is stored in a separate instance variable in the domain model. For example, a **Sketch** has an instance variable called **name** for storing the name of the sketch, and an instance variable called **strokes** for storing a collection of sketched lines.

Similarly, each relation to another domain model is also stored as an instance variable. For example, in a banking application, the **Customer** class would typically have an instance variable named **account**, for storing an instance of **BankAccount**. If a **BankAccount** had a reason to know its customer—that is, if the relation between **Customer** and **BankAccount** were bidirectional—the **BankAccount** class would have an instance variable named **customer**, for storing an instance of **Customer**.

A System Browser is used to add instance variables to a domain model class.

How Data Is Accessed

Each attribute and relation variable typically needs to have one method for getting its value, called an *accessor* method, and one method for setting its value, called a *mutator* method. (In some usages, “accessor” is an umbrella term referring to both accessor and mutator methods.) For example, a **Sketch** has a `name` method for getting the sketch’s name, and a `name:` method for setting the name.

A System Browser is used to define accessor and mutator methods for instance variables, in a protocol named *accessing*. Most programmers find it wise to use these methods rigorously throughout an application to get and set variable values, even though other methods in the domain model can get and set a variable’s value directly.

Data Processing

In addition to storing and accessing data, a domain model is responsible for enabling client objects to modify the data. Each such data-processing operation takes the form of an action method.

For example, a **Sketch** provides action methods for starting a new polyline (`beginStroke`), adding a point to a polyline (`add:`), erasing a polyline (`eraseLine`), and erasing the entire sketch (`eraseAll`).

Application Model

Overview

An application model is the core of an application in that it links the components of the domain model to the components of the user interface. In this mediating capacity, the application model has intimate knowledge of the domain model and UI.

Class Hierarchy

Each application requires its own subclass of `ApplicationModel`, the framework class that supports the mediating responsibilities.

In general, each such subclass drives a single application window, so a multi-window application may involve multiple subclasses of `ApplicationModel`. For example, the Online Documentation window in VisualWorks uses one

application model (**HelpBrowser**) while its example-browsing subwindow uses another model (**ExamplesBrowser**).

You can, of course, use an existing subclass of **ApplicationModel** as the parent of your new class, typically an abstract class of your own devising. Doing so would be useful, for example, when you want to build certain generic facilities into the abstract class so that all of your applications inherit and reuse those facilities.

Frequently reused subclasses of **ApplicationModel** include:

- n **SimpleDialog**, for dialog windows
- n **LensDataManager**, for data-form windows
- n **LensMainApplication**, for database applications

Creation

An application model class is typically generated the first time you install an application canvas.

An instance of such a class is typically created by selecting the name of the model in a Resource Finder and then clicking on the **Start** button. Programmatically, an application model is typically created as the first step in opening the application interface, by sending a variant of **open** to the class.

Components

The **ApplicationModel** class keeps two dictionaries for the convenience of all applications. One dictionary, called **DefaultLabels**, is used to register frequently used label strings, to avoid duplicating those strings throughout a set of applications. The second dictionary, **DefaultVisuals**, provides a similar service for graphic images that are used as labels.

An application model holds a **UIBuilder** in its builder variable, which is used to build the main user interface.

Responsibilities

The **ApplicationModel** class, its subclasses and its instances are responsible for:

- n Storage of reusable labels and images (**ApplicationModel** class)
- n Storage of interface specs (subclass)
- n Storage of value models (instance)

- n Dependent notification (instance)
- n Application startup (instance)
- n Application cleanup (instance)

Storage of Reusable Labels and Images

The `ApplicationModel` class provides a central registry for frequently used label strings, such as a company name, and a similar registry for graphic images, such as a company logo. The registries takes the form of two dictionaries held in class variables named `DefaultLabels` and `DefaultVisuals`.

***Note:** For a more structured approach to label storage, especially in a multi-cultural context, see the *International User's Guide* discussion of message catalogs.*

The class protocol named *resource accessing* contains methods for adding an entry to a registry (`labelAt:put:` or `visualAt:put:`), which is typically all that you need to do. Each label string or graphic image is associated with a lookup key that you provide, corresponding to the `Symbol` with which you identify the `Label` property of the widget. For a label string, the pound sign must be included in the `Label` property. For an image, the `Label Is Image` property must also be turned on for the widget.

This causes VisualWorks to search for an application method with the same name as the lookup key. In this case, there is no such method, so VisualWorks next searches the appropriate registry.

Each image in the `DefaultVisuals` registry occupies a significant amount of memory, depending on its size and color depth, so sparing usage is recommended.

Storage of Interface Specs

Each application model is responsible for storing the specifications for the application's user interfaces. Each application window requires a separate set of specifications. Each window's specs are stored in a class method, in an *interface specs* protocol. Thus, each concrete subclass of `ApplicationModel` usually has at least one such method for the primary window. The default name for this primary spec method is `windowSpec`.

A spec method is generated or regenerated by VisualWorks each time you **Install** a canvas. The method contains a literal array identifying window and widget specifications. When the canvas is opened for running or repainting,

the contents of the array are used to create a hierarchy of spec objects—roughly speaking, a `WindowSpec` containing a set of `WidgetSpecs`.

While you can edit a spec method as you can any other Smalltalk method, be aware that your edits will be overwritten if the canvas is later installed again. Also, the format of the literal array is considered private, and may change without warning in later releases of VisualWorks.

Storage of Value Models

Each application model is responsible for storing a value model for each value-displaying widget in each of the application's interfaces. The value model extracts the desired item of data from the domain model when the widget needs it, and updates the domain model when the widget indicates a new value has been accepted by the user.

Each such value model is typically held by an instance variable in your subclass of `ApplicationModel`. An accessor method, typically in an instance protocol named *accessing*, is also needed so the interface builder can obtain the value model and hand it to the widget at startup time.

Both the instance variable and the accessor method can be generated by VisualWorks, by using the **Define** button in the Canvas Tool. The value model is initialized to a zero or empty value in the accessor method, though you can override that initialization in the application model's `initialize` method.

There are several types of value models, which are described in the “Adaptors” chapter of the VisualWorks Cookbook.

For example, in a banking application, the application model would typically have an instance variable named `balance`. The variable would be initialized to hold an `AspectAdaptor` that is capable of accessing the `balance` variable in the domain model (a `BankAccount`). At startup time, the application's `UIBuilder` would supply the value model to a read-only input field for displaying the current account balance.

Dependent Notification

When Object B is affected by a change in Object A, Object B is said to be a *dependent* of Object A. Dependencies of this nature occur commonly in applications, and the application model collaborates with value models to notify dependents of relevant changes. These indirect messages enable each value model to communicate with its dependent widget without having to hold onto that widget directly.

For example, in the sketching application, selecting a sketch in the list widget causes the set of lines for that sketch to be displayed in a sketching widget. The sketching widget is a dependent because it needs to know when the selection is changed in the list of sketches.

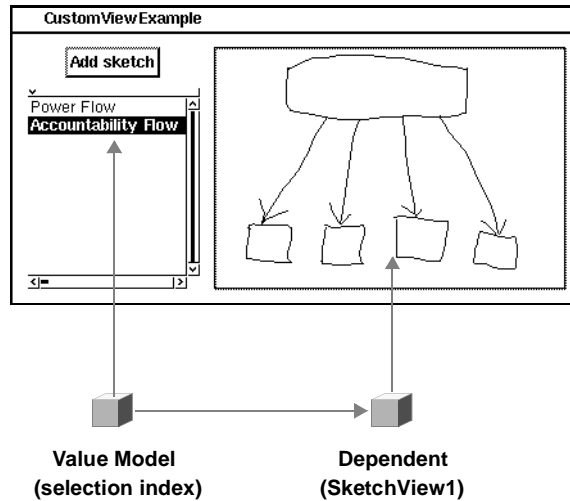


Figure 18-8 A value model can use dependency notifications to cause a secondary widget to update itself whenever the value model receives a new value. Here, the value model receives a new selection index for the list widget, and notifies the sketching widget to update its display.

It is important to note that the sketching widget is not a dependent of the list widget. Rather, it is a dependent of the value model that holds the list of sketches. The list widget is the primary dependent of the value model, and receives notifications much as its sibling widget does.

VisualWorks provides three layers of support for dependent notification:

- n Notifications from a value model to an application model. Many applications rely on this partially automated layer exclusively because it is the easiest to implement and handles the common cases.
- n Notifications from any object to any object. This is the foundation layer upon which the first layer is built, and which provides broader functionality for situations involving arbitrary types of objects.

- n Event-based notifications for objects of any type. This is actually an alternative to the second-layer architecture, provided for compatibility with Digitalk Smalltalk.

Notifications From Value Model to Application Model

An application model provides a value model to keep a widget in sync with its data value in the domain model. When a secondary widget also needs to be kept in sync with that data value, the application model employs a `DependencyTransformer`.

A `DependencyTransformer` is like a single-minded robot that is told, in effect: “Keep your eye on this value model—whenever its value is changed, notify me.”

This robot is told what message to send to the application model. By convention, the message begins or ends with the word “Changed,” as in `valueChanged` or `changedSelection`.

The **Notification** page of the Property Tool enables you to specify this message, in effect setting up a `DependencyTransformer` to monitor the primary widget’s value model.

The application model is expected to implement the corresponding instance method, in a `change messages` protocol. That method updates the value model for the secondary widget, which in turn causes the secondary widget to update its display, completing the cycle of dependency.

Example

Using the sketching application, here is how the sequence of events occurs:

1. The user clicks on the name of a sketch in the list widget, causing the `selectionIndexHolder` value model to change its value.
2. A `DependencyTransformer` notices the change and notifies the application model by sending a `changedSketch` message to it.
3. The application model, in its `changedSketch` method, gets the newly selected sketch and installs it in the sketch widget’s value model.
4. The sketch widget displays the sketch.

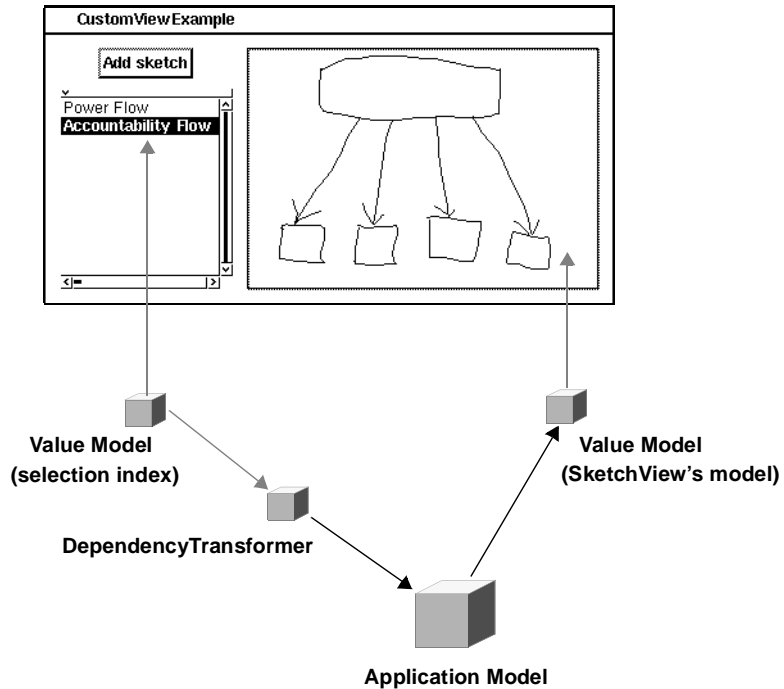


Figure 18-9 A DependencyTransformer is installed as a dependent of the value model that holds the selection index. The transformer is notified of a selection change, and in turn sends a changedSketch message to the application model, which gets the sketch from the first value model and installs it in the sketch view's value model.

Notifications From Any Object to Any Object

While the **Notification** page of a widget's property sheet enables you to arrange for a notification to an application model, you can use a **DependencyTransformer** to arrange for a notification from any object to any object. Going even further into the dependency mechanism, you can arrange for a direct notification without the use of a robotic third party.

DependencyTransformer

When a value model changes its value, it sends a `changed: #value` message to itself. The `changed:` method is inherited from **Object**, and sends an `update: #value` message to all dependents of the value model.

A `DependencyTransformer`, when it receives an `update: #value` message, sends a specified message to a specified receiver. In the usual situation, as discussed above, it sends a specified message to an application model. But as a general technique, it can be used to send any message to any receiver.

In addition, when the robot is monitoring an object other than a value model, it can be made to react to a `changed: #selection` message, for example, or any other *aspect symbol* indicating the nature of the change. The aspect symbol is used by contract between the object being monitored and the transformer.

For example, a `BankAccount` might send `changed: #balance` to itself, and the `DependencyTransformer` might be configured to pay attention to the corresponding `update: #balance` message, while ignoring other `update:` messages.

Setting up a notification in this way involves creating a `DependencyTransformer` with the appropriate aspect symbol, message selector and message receiver, and then adding that transformer as a dependent of the target object (using `addDependent:`). If the target object is not a subclass of `ValueModel`, you must also arrange for it to send `changed: <aspectSymbol>` to itself in the method that effects that change. Subclasses of `ValueModel` take care of that detail, because they are the most common targets.

Subclasses of `ValueModel` are capable of setting up a transformer for you. Just send `onChangeSend: <selector> to: <receiver>` to the value model.

Any object can set up a transformer in response to `expressInterestIn: <aspectSymbol> for: <receiver> sendBack: <selector>`.

Direct Dependency

You can dispense with the transformer by implementing an `update: method` for the dependent object. Then add that object as a dependent of the target object (using `addDependent:`). As a result, when the target object sends `changed: <aspectSymbol>` to itself, the dependent object will receive `update: <aspectSymbol>`.

Again, the aspect symbol must be agreed upon.

Variants of the `changed/update`: messages are available for situations requiring a parameter in addition to the aspect symbol (`update:with:`) and the target object (`update:with:from:`).

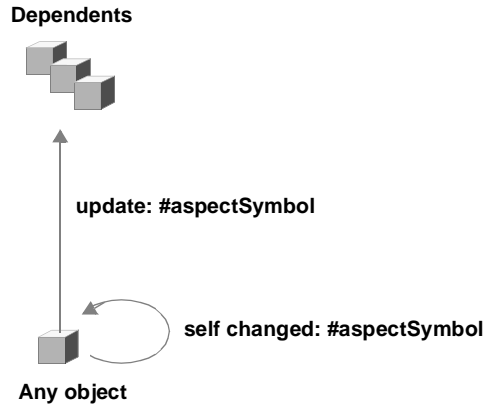


Figure 18-10 Any object can register dependents and then notify them by sending itself a `changed`: message with a symbol indicating the nature of the change. As a result, each dependent receives an `update`: message.

Removing Dependents

The `Object` class provides a central dictionary for keeping track of any object's dependents. An application that adds a dependent is also responsible for removing it (using `removeDependent:`), to avoid having the dictionary hold onto obsolete dependents and waste increasing amounts of memory.

The `Model` class provides an instance variable for storing dependents locally, avoiding the use of the central dictionary. Thus, instances of subclasses of `Model` (including the value model hierarchy) automatically release their dependents when they expire. Because value models are the targets of the vast majority of dependencies, this takes care of most situations.

Circular Dependencies

Because dependencies involve indirect communications, the hazard of circular message-passing becomes more likely. The most common situation in which circularity arises involves two mutually dependent widgets.

For example, in the Online Documentation window, the page number field is mutually dependent with the list widget. That is, changing the page number

updates the selection in the list, and changing the selection in the list updates the page number.

You can temporarily remove a transformer in such a situation, by sending `retractInterestIn: <aspect> for: <dependent>` to the target object just before you change its value. After changing the value, you must reestablish the transformer (using `onChangeSend:to:`).

You can temporarily remove a direct dependent by sending `removeDependent: <dependent>` to the target object, and then adding it (using `addDependent:`) after changing the value.

Event-Based Notifications

A similar mechanism for indirect communication with dependent objects was introduced in Digitalk's dialect of Smalltalk. For the convenience of users who are migrating applications from Digitalk Smalltalk/V or Visual Smalltalk to VisualWorks, this event-based mechanism is also available.

The code that supports event-based notifications is supplied in a file named **sysdeps.st**, in the **extras** subdirectory under the VisualWorks installation directory. The file can be loaded by filing it in, using a File List. After you file in the code, the class will have additional protocols whose names begin with the word "event." In addition, supporting classes will be added to the system, in a class category named "System-Dependency Events."

With this mechanism, each object can define certain events that it promises to announce when appropriate. Announcing an event to potential dependents is called *triggering* an event. A dependent object can register a handler for an event in which it is interested.

Defining Events

Each class is responsible for defining the events that it will trigger. The inherited class method `eventsTriggered` must be redefined for each class that wishes to define a set of valid events. The `eventsTriggered` method typically creates an `IdentitySet` of event names (`Symbols`), then returns the set. It can, of course, invoke `super eventsTriggered` to fetch the parent class's events, then add to that set before returning it.

Event names, like message selectors, can be unary or keyword names. A unary event has no parameter, while a keyword event has as many parameters as it has colons. For example a button class might define a `#clicked` event, because the dependent object needs no further information. A list class might define a `#changed:` event, because the dependent needs to know which item

in the list was selected—the selection can be passed as an argument to the `changed:` message.

Triggering Events

An object can trigger any event in its class's `eventsTriggered` set. It does so by sending a variant of `triggerEvent:` to itself. The argument is the event name. See the *event triggering* protocol in `Object` for the variants. An event is triggered in the method or methods that effect the described event.

Registering an Event Handler

A dependent object can arrange for a specific action to be taken each time the triggering object triggers a specific event. This is known as registering an event handler.

The dependent sends a variant of `when:send:to:` to the triggering object. The first argument is the event name, the second argument is a message name, and the third argument is the message receiver. See the *event configuring* protocol in `Object` for the variants.

The dependent object need not do the registering itself, of course. For example, an application model could use `when:send:to:` to arrange for a domain model to send a message to a dependent widget.

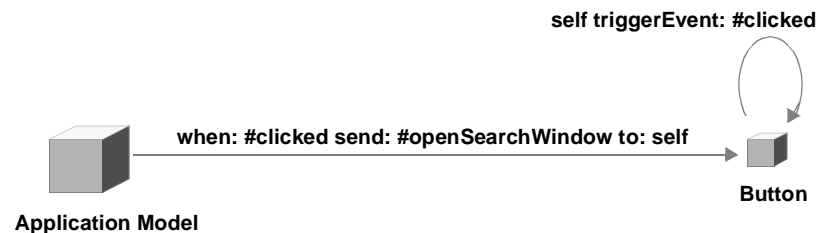


Figure 18-11 When using event-based notifications, a button class could define an event named `#clicked`. It would send `triggerEvent: clicked` to itself in the appropriate method. A dependent such as an application model could then arrange to be sent a message such as `openSearchWindow` whenever the button was clicked.

By default, an error occurs when an event handler is registered for an event that has not been defined by the triggering class. The registering object can verify that a particular event is triggered by sending `canTriggerEvent:`, either to the triggering object or to its class.

Removing Dependencies

When an event handler is registered, it is stored in a class variable named `EventHandlers`, which is inherited from `Object`. The application is responsible for removing each handler from this event table when the handler is no longer needed.

The triggering object can remove all handlers that have been registered with it by sending `release` to itself.

A more specific message, `releaseEventTable`, can be used when other forms of dependency are not ready to be released.

Application Startup

The first step in starting an application involves deciding which interface to open. The process of assembling and opening the chosen interface proceeds by stages. After each stage, your application model can intervene in the process to configure the raw interface as needed. The stages are:

- n Create an instance of `UIBuilder`
- n Pass the UI specs to the builder and ask it to construct the UI objects
- n Open the fully assembled interface window

By default, when an application model class is sent an `open` or `openInterface:` message, all three stages are performed. You can send `allButOpenInterface:` to an instance to perform stages one and two, then separately send `finallyOpen` to perform stage three.

Selecting an Interface

An application is typically started by sending an `open` message to the appropriate subclass of `ApplicationModel`. This assumes that the primary canvas was saved with the default name, `windowSpec`.

If the primary canvas has a different name, or if you want to open a different canvas, you can send `openWithSpec:` to the class, with the spec name as the argument.

The application model class creates a new instance of itself to run the interface. If you want to use an existing application model instance, you can send `open` or `openInterface:` to that instance. This is useful when you want to reuse an instance rather than create a new one, or when you want to initialize the application specially.

Prebuild Intervention

After an instance of `UIBuilder` has been created, but before it has been given a set of specs with which to construct a UI, the application model is sent a `preBuildWith:` message. The argument is the newly created `UIBuilder`.

Most applications do not need to intervene at this stage. Those that do, typically take the opportunity to load the builder with custom bindings that can only be derived at runtime. For more discussion of bindings, see “Storage of UI Bindings” on page 247.

Postbuild Intervention

The application model creates a hierarchy of spec objects from the spec method, and hands the root spec to the builder. The builder then creates a window and populates it with the appropriate widgets. The builder does not yet open the window, however.

At this stage, the application model receives a `postBuildWith:` message, with the builder as argument. The application model can use the builder to access the window and any named widgets within the window—that is, widgets that were given an `ID` property.

Applications commonly use `postBuildWith:` to hide or disable widgets as needed by the runtime conditions.

Postopen Intervention

The builder opens the fully-assembled interface. At this stage, the application model is sent a `postOpenWith:` message, again with the builder as argument. As with `postBuildWith:`, the application can use the builder to access the window and its widgets. This time, however, those objects have been mapped to the screen, which makes a difference for some kinds of configuration.

For example, the `FileBrowser` model that drives the File List interface uses `postOpenWith:` to insert the default path in the window’s title bar—something it could not do until after the window had been opened.

Application Cleanup

An application model often needs to take certain actions when the application is closed. For example, a word-processing application might need to ask the user whether edits that have been made to the currently displayed text should be saved or discarded.

Another common cleanup action is to break circular dependencies that would otherwise prevent the application from being garbage collected. For example, if application A holds application B, and vice versa, for the purpose of inter-application communications, neither would be removed from memory even after both of their windows were closed.

If the application user exits from the application by using a menu or other widget in the interface, the application model performs the exit procedure and can insert any required safeguards. But if the user exits by closing the main window, a special mechanism is needed to notify the application model.

The application model is held by the application window. When the window is about to be closed, its controller asks for permission from the application model, by sending a `requestForWindowClose`. The application model can redefine this method to perform any cleanup actions and then return `true` to grant permission or `false` to prevent the window from closing.

Builder

Overview

An application builder is a component of an application model that is responsible for generating a running user interface from a set of window and widget specifications. It performs this service either when an application is being started or when a canvas is being painted.

Creation

An application typically has a single builder, held by its `builder` variable. This primary builder is created by inherited *interface opening* methods.

Some applications also create one or more secondary builders as needed to construct subwindows and custom dialogs. Frequently, a secondary builder is discarded as soon as it has finished opening its window. When the application model needs to access widgets in the subwindow—for example, to disable a widget when conditions in the main window change—the secondary builder is held in a newly created instance variable.

Components

A builder holds the application model in its source variable, because the application model is the `SOURCE` from which the builder obtains value models, menus and other resources.

Each such resource is stored in a dictionary held by the builder. The lookup key in each case is the selector that was used to obtain the resource from the application model. For a value model, for example, this selector is the **Aspect** property of the widget. The dictionary is held in a variable named **bindings**, because it holds the resources to which the interface components are bound.

When a builder is given a set of specs, it begins traversing the hierarchy of specification objects, constructing a corresponding widget or support object for the eventual UI. When the window is created, it is stored in a variable named **window**. If a widget has an **ID** property, it is stored in a **namedComponents** dictionary in the builder.

The builder delegates the task of choosing a widget for each spec to a subclass of **UILookPolicy**. Each platform look-and-feel is enforced by a separate policy class, such as **Win3LookPolicy** for Microsoft Windows. This policy object is held in a variable named **policy**.

Responsibilities

In terms of its public contract with an application model, a builder is responsible for:

- n Storage of UI Bindings
- n Interface Assembly
- n Interface Opening
- n Window Access
- n Named Component Access

Storage of UI Bindings

For each **WidgetSpec** that identifies a value model or other resource, the builder obtains that resource from the application model. First, however, the builder checks its **bindings** dictionary to see whether that resource has already been fetched for a previous widget.

Your application can make use of this two-stage lookup process to insert resources into the **bindings** dictionary proactively. This is useful mainly when the application is building an interface directly, rather than employing a prebuilt canvas. A dialog whose set of widgets must be customized to suit the circumstances is the usual situation requiring direct UI building.

To supply a resource proactively, an application model sends a message such as **aspectAt: aKey put: aValueModel** to the builder. The first argument in

this case is the **Aspect** property of the widget. The second argument is the value model that would normally be returned by the application model in response to the aspect message.

The *binding* protocol in `UIBuilder` has variants of `aspectAt:put:` for other kinds of resources.

By loading the bindings in this way, your application avoids the builder's second-stage resource lookup, in which it sends the aspect message to the application model. The application model need not define a separate method for supplying the resource, nor an instance variable for storing what may be a very temporarily or infrequently needed resource.

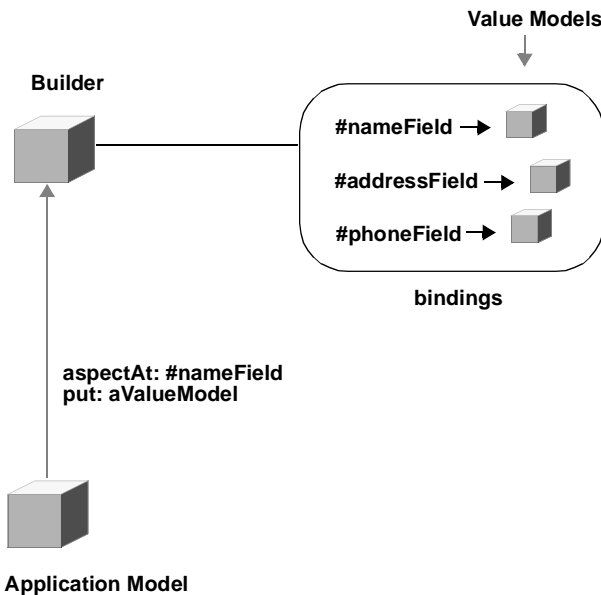


Figure 18-12 A builder caches value models and other resources in a bindings dictionary. An application model can preload this cache to avoid being asked for a resource via message-send.

Interface Assembly

A builder is responsible for converting a set of `UISpecification` objects into a window, widgets and supporting objects, using its look policy to select widgets that have a particular look and feel.

For example, when a builder's look policy is an instance of `Win3LookPolicy`, it will add a `Win3RadioButtonView` to the interface as the specific implementation of a `RadioButtonSpec`. When a `MacLookPolicy` is in effect, the builder will add a `MacRadioButtonView` to implement that same `RadioButtonSpec`.

The builder's default policy is established by using the Settings Tool to set a system-wide default. An application can install a different policy in the builder, usually in a `preBuildWith:` method, by sending a `policy:` message.

When an application is using a builder to construct a custom dialog or other UI directly, it does so by sending `add: aSpec` to the builder. The argument is typically a root-level spec object containing the hierarchy of window, widget and supporting specs. It can also be an individual spec object, though this is less often attempted. A root-level spec object is typically obtained by sending `interfaceSpecFor:` to the application model class, with the name of a spec method as the argument.

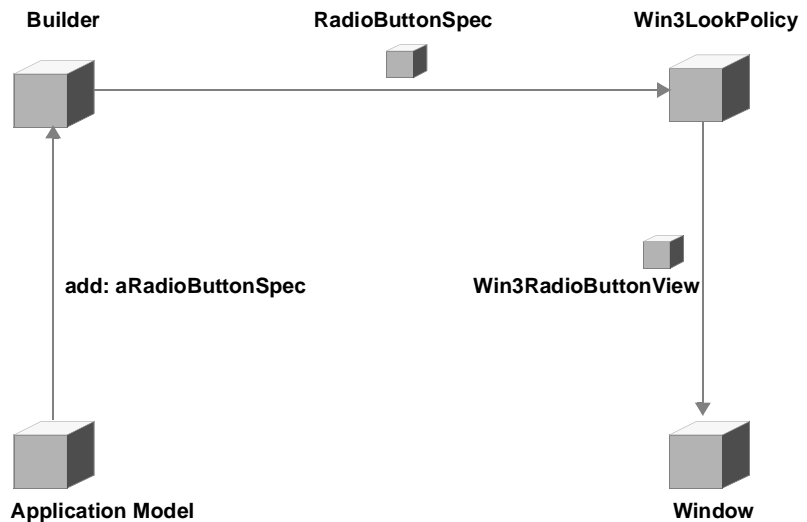


Figure 18-13 How an abstract specification is turned into a platform-specific widget. The application model asks its builder to add a spec to the interface. The builder delegates the task to its look policy. The look policy selects a widget that suits the platform look, and installs it in the interface window.

Interface Opening

A builder is responsible for opening the application window after populating it with widgets. An application can exert control over certain aspects of this phase by sending an *interface opening* message to itself. However, for finer control over the window location, size and type, the application can send a *scheduling* message directly to the builder.

The application can also use a *scheduling* message to control the timing of the window opening. For example, an application might cache a prebuilt interface and then open, close and reopen it as needed.

Window Access

An application model holds a builder, and the builder holds the application window. When the application model needs to access the window, perhaps to iconify it or change its title, it sends a **window** message to the builder.

Named Component Access

A builder is responsible for providing its application model with access to any widget that has an **ID** property, called a named component. The builder stores such widgets in its **namedComponents** dictionary.

The application can obtain a named widget by sending **componentAt: aWidgetID** to the builder. The usual motivation for doing so is to hide, disable or restore the widget to suit changing circumstances.

Each widget is contained by a **WidgetWrapper**, which uses a **WidgetState** to apply bordering, visibility and other appearance characteristics to the widget. It is the wrapper that is stored in **namedComponents** and returned from a **componentAt: query**. The usual operations that an application performs, such as hiding or disabling a widget, are actually addressed to the

wrapper. When the application needs to address the contained widget rather than the wrapper, it can ask the wrapper for its widget.

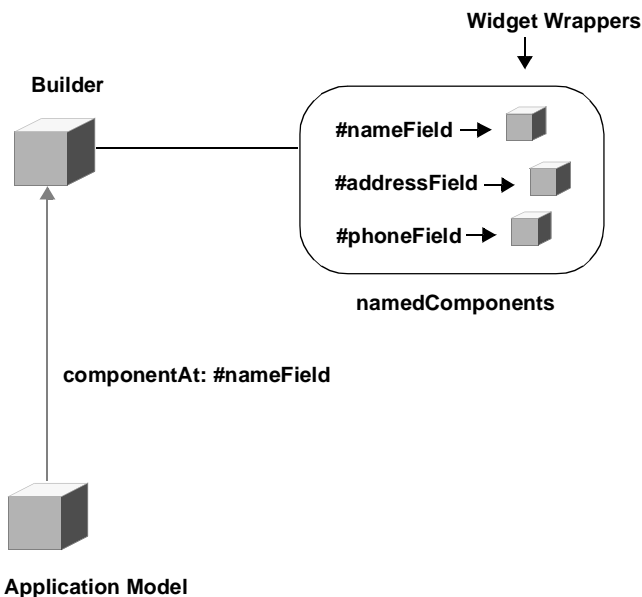


Figure 18-14 When a widget is given an ID property, the builder stores it, inside its wrapper, in a `namedComponents` dictionary. The application model can ask the builder for any named component while the application is running.

Window

Overview

A window is a display surface on which a set of widgets display their contents. While some window managers use the term *window* to mean both top-level windows and the subwindows in which individual widgets are displayed, VisualWorks uses the term exclusively for top-level windows.

Creation

A window is typically created by a `UIBuilder`, being the top-level interface object as specified by a `WindowSpec`.

Windows come in three types:

- n Normal windows, having full decorations
- n Dialog windows, having a border but (depending on the window manager) typically fewer border widgets and no title bar.
- n Pop-up windows, having no decorations, such as a menu

Variants of the *scheduling* protocol in `ApplicationWindow` and its parent, `ScheduledWindow`, enable the application to control which type of window is used.

Class Hierarchy

A window as used in an application is typically an instance of `ApplicationWindow`. The parent class, `ScheduledWindow`, is used in older applications that predate the VisualWorks canvas-painting tools. `ScheduledWindow` provides much of the state and behavior upon which `ApplicationWindow` relies.

Still farther back in the ancestor chain is `Window`, which is now mostly treated as an abstract class. Lacking a controller to enable a user to control the window, it is too passive for the vast majority of uses. However, it does provide important foundation for `ApplicationWindow`.

Components

An application window has a **controller**, usually an `ApplicationStandardSystemController`, that provides a menu of window-controlling actions such as **refresh** and **close**.

A window also has a **component**, which can be a single visual component such as a `ComposedText` but is almost always a `CompositePart` that holds a hierarchy of widgets.

A window has a **sensor** for providing information to widget controllers about mouse activity, and a **keyboardProcessor** for providing information about keyboard activity.

A window has a **label**, which is the string that appears in its title bar, and an **icon**, which appears when the window is collapsed.

When a window is supposed to collapse, expand or close whenever another window performs one of those actions, it is said to be a *slave* of that *master window*. A window has a `masterWindow` variable for holding its master, if any.

A window has several policy objects that are inherited by the window's widgets unless they have been given a policy explicitly. These policies control color, bordering and similar characteristics.

Responsibilities

Most of the responsibilities of a window involve straightforward communications, as documented in the “Windows” chapter of the *VisualWorks Cookbook*. Damage repair is a mechanism worthy of discussion here.

Damage Repair

A window is responsible for redisplaying portions of its surface that become damaged in either of two ways:

- n Window damage—when an overlapping window is moved, causing a formerly obscured region to need redisplaying, or when the window is refreshed.
- n Information updates—when the state of the application changes in a way that invalidates something that a widget is displaying

Window damage is communicated by the window manager. A damage rectangle resulting from an information update is communicated by the widget involved. The visual component of the widget sends a variant of `invalidate` to itself, which causes the window to be sent `invalidateRectangle:repairNow:forComponent:`.

In either case, the window is told that a rectangular portion of its surface has become damaged. This *damage rectangle* is accumulated by the window's sensor along with other damage rectangles.

When the application is not busy with a higher-priority process such as file accessing, the window redisplay all of the damaged portions. For each such rectangle, it restores its own background color and then asks each widget that intersects the rectangle to redisplay its contents.

The window communicates these requests to its component, the `CompositePart` that handles layout issues for the individual widgets. The `CompositePart` determines which widgets are affected and passes on to them the request to redisplay.

Damage repair is automatic, and most applications have no need to intervene in the process. Occasionally an application desires to repair damage proactively. For example, suppose an application requests information from the user by displaying a dialog and then, when the dialog has been dismissed,

begins a process that takes several minutes to complete. When the dialog is dismissed, the application window has to repair the damaged area that the dialog overlapped. But that repair is delayed until the application process is finished, unless the application asks for repair immediately.

The application can force immediate repair by sending `invalidate-Rectangle:repairNow:` to a widget, with `true` as the `repairNow` argument.

Visual Component

Overview

A window is a container for a visual component. A visual component, or view, is responsible for displaying some aspect of a model. For example, an input field might display an account balance, or a table might display a set of transactions.

When combined with a controller, which is responsible for handling user input, a visual component takes on the essential characteristics of a widget—an interface component that enables a user to view and modify some aspect of a domain model.

A value model typically mediates between the view and the domain model. So each component of an interface consists of a view and a controller interacting with a value model. This is known as a model-value-controller architecture, or MVC, which has been refined through generations of Smalltalk.

A visual component can take many forms. This section describes the types of visual components according to their fundamental characteristics:

- n Passive or active
- n Autonomous or dependent
- n Singular or composite

The “Custom Views” chapter of the *VisualWorks Cookbook* provides instructions for creating a custom visual component and integrating it into an interface.

Passive vs. Active Components

An application window has a visual component, so we say that the window is the *container* of that component. The visual component that is put into that container can be either passive or active. A passive visual component receives requests from the container but does not send requests to it. An active visual

component receives and also sends requests, so it needs to keep track of its container.

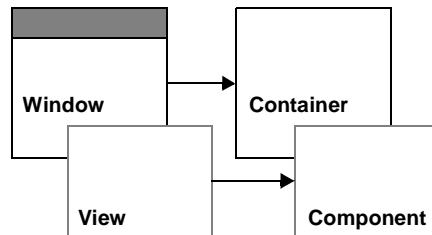


Figure 18-15 *Window and view as container and component.*

The abstract class `VisualPart` provides this ability to communicate with the container, so it is the parent of all active visual components, including `View` and its subclasses.

Passive visual components include instances of `Image`, `Icon`, `ComposedText` and `TextList`. The geometric objects, such as `Circle` and `Spline`, are not `VisualComponents`—they must be placed in a `GeometricWrapper` to achieve component status. See the chapters on graphic objects and text objects in the *VisualWorks Cookbook* for more information about creating such objects.

Active visual components (`VisualParts`) include instances of:

- n `View` and its subclasses
- n `CompositePart`, which holds a collection of other components
- n `Wrapper`, which adds generic functionality such as bordering to a component

Among the motivations for a `VisualPart` to talk to its container are:

- n *Invalidation*, because a `VisualPart` coordinates its redisplaying activities with the damage-repair mechanism of the window.
- n *Bounds accessing*, because an active component adjusts itself to suit the size of its window
- n *Getting a graphics context*, because an active component redisplayes all or part of itself depending on the state of the model

- n *Getting a sensor and keyboard processor*, because the typical controller gets mouse and keyboard input via the window's sensor and keyboard processor

Autonomous vs. Dependent Components

As you might expect, active components come in many shapes. The three major kinds of `VisualPart` are composites, wrappers and views. Composites and wrappers both support layout and coordination of multiple views in a single window—they are described in a later section. Composites and wrappers are autonomous, because they typically operate independent of a model.

That leaves views. The distinguishing trait of a `View` is that it has a controller. As a subclass of `VisualPart`, a view also has a container—so far, then, a view knows its container and its controller. How does it know its model?

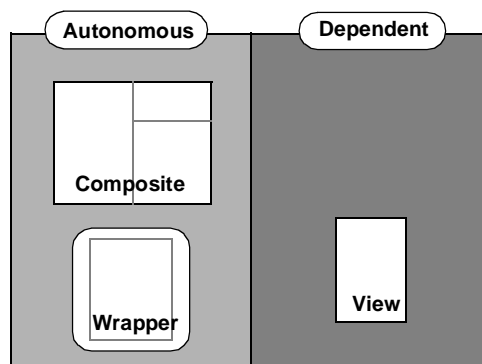


Figure 18-16 Autonomous vs. dependent visual components

Interposed between `VisualPart` and `View` in the class hierarchy is the class `DependentPart`. A `DependentPart` has a model. The only direct subclass of `DependentPart` that comes with the class library is `View`, so it may seem that the two of them could have been combined. However, you may well encounter a situation requiring a visual component that knows its model but has no controller.

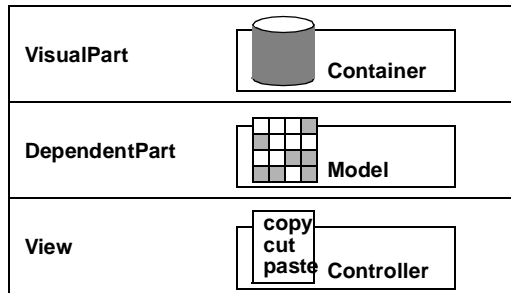


Figure 18-17 The connections provided by VisualPart, DependentPart and View as their primary contributions to the view hierarchy.

For example, a file-locating utility might reserve a portion of the window to display the directory in which a file was found. No user input would be accepted in that portion of the window, so no controller is required, but it changes in step with the model. Thus, anytime you create a new view class that is not intended to have a controller, it really belongs as a subclass of DependentPart.

Controller Linking

By default, a view creates its own controller the first time it is asked for one. It does so by sending `defaultControllerClass` to itself and creating a new instance of the returned class. For example, the View class returns Controller when asked for its default controller class.

In a custom view class, you can redefine the `defaultControllerClass` method. You can also supply a controller instance at runtime by sending a `controller:` message to a view.

Model Linking

A view registers itself as a dependent of its model. Then, whenever the model sends `self changed`, the view receives an `update:` message. Thus, linking a view to a model has two parts:

- n Registering the view as a dependent of the model. Subclasses of **DependentPart** handle this automatically when a **model:** message is sent to the view to set its model.
- n Implementing an **update:** method to display some aspect of the model's information

Redisplaying

A view is responsible for redisplaying its contents whenever it is affected by a damage rectangle being repaired by the window. It does so in a **displayOn:** method, which is sent by the window, with the window's graphics context as the argument.

A view must also register a damage rectangle with the window, representing all or part of its area, whenever it receives an **update:** message from the value model. Registering a damage rectangle is done by declaring the view's area invalid, by sending **invalidate** to itself. Subclasses of **DependentPart** do this automatically.

Thus, by implementing a **displayOn:** method, and sending **self invalidate** in its **update:** method, a view is assured of responding to any redisplaying situation in a unified way.

Selective Redisplaying

By sending **self invalidate**, the view is adding its entire display box to the window's list of damage rectangles. In many situations, the view only needs to redisplay a portion of its area that contains some dynamic element. An **invalidateRectangle:** message can be used to invalidate a portion of the view.

In a list view, for example, only one or two lines in the list need to be redisplayed: the line that was selected, if any, and the line that was deselected, if any.

Immediate vs. Lazy Damage Repair

By default, the window accumulates damage rectangles until it receives a **checkForEvents** message from its controller. The standard controller sends that message each time it is polled for activity. That is frequent enough in most situations. However, if a competing process is hogging the processor, there can be a significant delay between the time the model changes and the

view is updated. In such situations, another variant of `invalidate` lets you indicate that the window should repair its damage immediately:

self `invalidateRectangle`: aRectangle `repairNow`: true

A value of `false` for the `repairNow`: argument is the default, and is sometimes called *lazy damage repair*.

Composite Visual Component

A window is only prepared to communicate with a single visual component. To place two or more components in the same window, some intermediary object is needed to hold the *leaf components*. The situation is analogous to a cardboard box filled with wine bottles—without a grid of cardboard separators, the bottles would bump into one another, with unfortunate results for the wine lover.

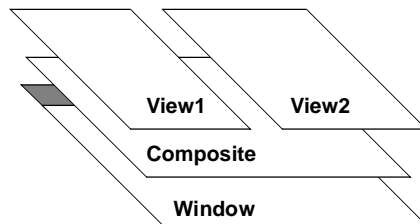


Figure 18-18 A CompositePart holds a collection of other visual components.

A `CompositePart` provides this separation framework. To the window, it is just another visual component, one that happens to hold a collection of other visual components. To each of its subcomponents, it is a container just like a window. A subcomponent can be another composite object.

When a graphics context is passed down from the window, the `CompositePart` passes it on to those of its subcomponents that say they intersect the affected area.

`CompositePart` has two main subclasses:

- n `DependentComposite`, for situations in which the composite object itself needs to be a dependent of the model. This is the composite equivalent of `DependentPart`.

- n **CompositeView**, a subclass of **DependentComposite**, which has a controller that spans all of its subcomponents. This is the composite equivalent of **View**.

Lest the analogy of the wine carton's grid lead you astray, a **CompositePart** does not perform translation or clipping for its subcomponents, which can overlap one another. The composite does not bother to keep track of where its subcomponents are located or how big they are. For complicated composites, such an arrangement would require that the composite consult a large collection of coordinates each time it needed to determine which subcomponent to update.

Instead, the composite leaves it to the subcomponents to keep track of such details. To spare you from having to equip every kind of visual component with this ability, it has been placed in a special kind of **VisualComponent** called a *wrapper*. Thus, a window that has two views would contain a composite that contains two wrappers, each of which contains one of the views.

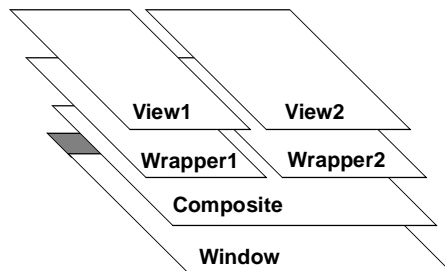


Figure 18-19 Each leaf component is contained by a wrapper.

Wrapper

Like a **CompositePart**, a **Wrapper** is both a container and a component. It contains a single visual component, providing a service to that component such as translating the origin or clipping the display box.

In other respects it tries to be transparent, forwarding the most common messages from its container down to its component, and from its component up to its container.

Wrappers can be nested inside one another, too, so you can build up a complex set of behaviors by using several simple, highly reusable wrappers.

Controller

A controller is what makes a view seem to respond directly to mouse movements and keyboard activity. By shaping the nature of that response, a controller defines the *feel* of an interface, as apart from its look.

Each window and each active component within a window has an associated controller. The window's controller typically provides the familiar <Window> menu (**move, resize, close**, etc.). A visual component's controller has a broader range of typical duties:

- n Providing an <Operate> menu
- n Notifying the model about selections made with the mouse
- n Forwarding keyboard activity to the model
- n Changing the cursor

Thus, in an application window such as the System Browser, you are communicating with two different controllers depending on which mouse button you press: The <Window> button is fielded by the window's controller, while the <Select> and <Operate> buttons are fielded by the view's controller.

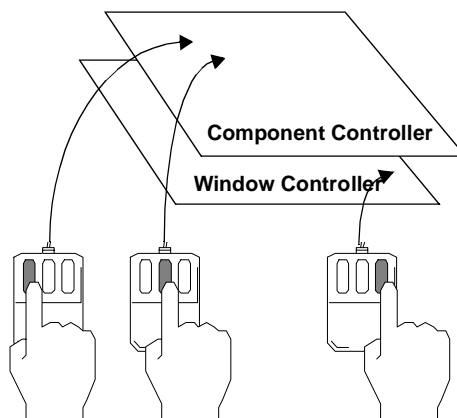


Figure 18-20 *The controller with which each mouse button interacts.*

Just as models and views are specialized for particular purposes, each controller class has its unique set of abilities. A button's controller may only need to pay attention to clicks of a particular mouse button, for example,

while a text-editing controller has to pay attention to both mouse and keyboard actions.

Polling vs. Event-Driven Controllers

VisualWorks supports one type of input controller that uses a polling architecture and another type that uses an event-driven architecture. Within a given user interface, all views must employ the same input architecture. The “Custom Controllers” chapter of the *VisualWorks Cookbook* provides instructions for creating either type of controller and integrating it with a view and a value model.

One type of input controller uses a loop to repeatedly check for input events, for as long as the controller retains control. (Typically, a controller retains control while the mouse cursor is within the boundaries of the associated view.) Each time the controller asks for events that have occurred since the previous iteration, it is said to be *polling* for events, hence the name *polling controller*.

The second type of input controller does not use a loop to poll for input events. Instead, it relies on the **ControlManager** to notify it whenever an input event occurs. It then decides whether the event is of interest—for example, a button widget’s controller cares about mouse-button events but ignores most keyboard events. Because this type of controller is inactive except when there is a relevant input event for it to process, it is said to be driven by events, or *event-driven*. While the flow of control is dispatched to a polling controller, individual events are dispatched to an event-driven controller.

A given controller class can be equipped for both input architectures. After you file in the input events code as described in the *VisualWorks Cookbook*, standard VisualWorks controllers are capable of servicing either a polling or an event-driven canvas. A custom controller class also inherits from **Controller** most of the machinery for fitting into either architecture, though you will typically need to add custom protocol for each architecture.

Thus, for example, you can continue to use a custom controller within the older polling style while you layer on the event-handling methods. Then you can switch any canvases that use the controller to the event-driven architecture.

Flow of Control (Polling Controller)

Ownership of the user input is commonly referred to as *control*. The host operating system hands control to a control manager when a VisualWorks window is activated. The control manager passes control on to the window that contains the cursor.

Control Manager

Each `ApplicationStandardSystemController`, which is associated with a window rather than a view, is entered in a collection held by the `ScheduledControllers` object. `ScheduledControllers` is a global variable that holds an instance of `ControlManager`. This control manager is responsible for passing control to the active window's controller, and is the reason that a `ScheduledWindow` has "scheduled" in its name.

The window's controller asks the window which of its subcomponents wants control, if any (`subviewWantingControl`). In response to this message, the window requests `objectWantingControl` from its component. The component, if it is a composite, forwards that message to each of its subcomponents, and so on. Each leaf-level component, upon receiving this message, asks its controller `isControlWanted`. The first controller to respond `true` is sent a `startUp` message by the control manager, beginning its *basic control sequence*.

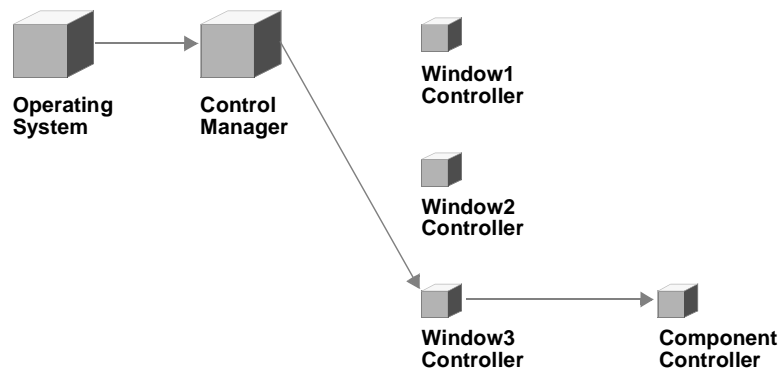


Figure 18-21 How the flow of control proceeds from the operating system to a specific component controller in a specific window.

Controllers rely on sensors to help them make their control-accepting decisions. Each window maintains a sensor for gathering input, and controllers bombard this object with questions. Has a mouse button been pressed? Which one? Has it been released yet? Where was the cursor located at the time? Based on the responses to questions such as these, the controller either accepts or refuses control.

Basic Control Sequence

The basic control sequence consists of three steps:

- n Initialize control
- n Loop while the conditions for holding control are met
- n Terminate control

Initialization is performed in a `controlInitialize` method. By default, a visual component's controller does nothing in response to this message. You can reimplement the method in a controller class to perform some special action when the controller starts up. In the sketching application, the cursor is changed from its normal shape to cross-hairs.

This implies that you change the cursor back to its normal shape when the controller yields control. That would be done in the control-terminating method, `controlTerminate`.

In practice, it is difficult to guarantee that initializing and terminating methods will be called in matched pairs. For example, if the user were to interrupt the drawing program, a new controller would take control and the cursor would never be changed back.

In its `controlLoop` method, a controller first verifies that the conditions for maintaining control (`isControlActive`) are still true. If so, it sends `controlActivity` to itself, after which it repeats the `isControlActive` test.

The `controlActivity` method is the real meat of the controller. Here, the controller typically queries its sensor to find out whether a particular type of input event has occurred.

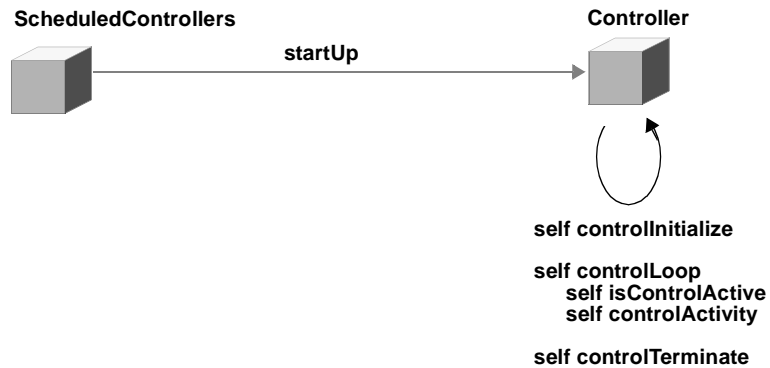


Figure 18-22 On receiving control, a controller initializes itself, repeats an activity loop, and then performs finalization actions

When the `isControlActive` test fails, control reverts to `ScheduledControllers`, which begins polling its scheduled controllers to find a new control receiver.

Input Sensing

In the polling architecture, `InputState` reads events from the Object Engine. For each event, `InputState` caches mouse button and keyboard states as needed, then dispatches the event to the `WindowSensor` of the appropriate window.

The sensor queues a keyboard event or a meta event in the appropriate `SharedQueue`. In the case of a button event, the `WindowSensor` executes a `Delay`, so the `UIProcess` has time to check the button states.

A widget controller, in its activity loop, polls the sensor for outstanding mouse and keyboard events, and reacts to each event appropriately

Flow of Events (Event-Driven Controller)

Event Queuing

The event-driven architecture is designed to co-exist with the polling architecture. After the `events.st` file in the `extras` directory has been filed in, each window canvas can be toggled between the polling and event-driven

architectures, using the Property Tool. The default is polling until `events.st` is filed in.

An event-driven window's sensor is an instance of `EventSensor`. `EventSensor` receives events from the `InputState` and queues button, keyboard and window events into one `EventQueue`.

By default, `EventSensor` collapses `MouseMovedEvents`. Collapsing events means that before queuing an event, the `EventSensor` checks whether the last item in the `EventQueue` is of the same type. If it is, it replaces the last event with the current one.

Event Dispatching

An `ApplicationStandardSystemController` for an event-driven window runs an `eventLoop` instead of `controlLoop`. In the `eventLoop`, the controller waits on its window sensor's `EventQueue` for the next event, then processes it, until its window is no longer active. When the controller dequeues an event, it passes the event to its `EventDispatcher`.

An `EventDispatcher` generally sends window events to the window, mouse events to a widget's controller and keyboard events to the `KeyboardProcessor` of the window.

While a polling controller is asked whether it wants control (`isControlWanted`), an event-driven controller is asked whether it handles a particular mouse event (`handlerForMouseEvent:`). While the polling controller typically uses a `viewHasCursor` test to accept or reject control, an event-driven controller typically uses `viewHasCursorWithEvent:` for the same purpose.

Instead of using a control loop that checks for user input continuously, an event-driven controller must be prepared to respond to each type of input event individually. To do so, it must be equipped with a set of event methods, such as `enterEvent:`, `exitEvent:` and `keyPressedEvent:`. The `Controller` class provides default methods (see the *events* protocol), which generally do nothing, so you only need to define methods for events that your custom controller cares about.

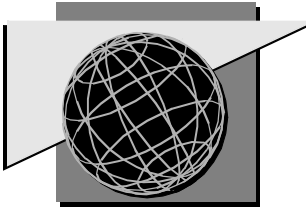
Selection Tracking

Selection tracking is a very common behavior among controllers. Most controllers need to do some form of selection tracking between the time they receive a button-pressed event and the time they receive a button-released event. For example, between the time the user presses and releases the <Select> button in a text editor, a `TextEditorController` grabs all `Mouse-`

MovedEvents and performs textual selection tracking for each mouse movement.

A polling controller starts a loop (in **selectDownAt:** or **startDragging**) in which it grabs and processes all events until the mouse button is released.

In the event-driven architecture, a hierarchy of classes has been created for performing selection tracking on behalf of different types of controllers. A **SelectionTracker** is created by a controller in response to a **redButtonPressedEvent:** message. A **RedButtonPressedEvent** is then dispatched to the **SelectionTracker**. At this point, the **SelectionTracker** generally grabs all mouse events until the <Select> button is released.



Chapter 19

Graphic Operations

The Smalltalk Portable Imaging Model (SPIM) supports the display of portable, two-dimensional, color graphics. The SPIM classes place a variety of static and animated graphic effects at your command.

This chapter describes the structure and use of the graphics classes, in four sections:

- n The fundamentals of SPIM graphics
- n The kinds of display surfaces on which a graphic can be drawn
- n The graphic objects that can be drawn on a display surface
- n How to integrate a graphic object into your application

Three other sections of the documentation describe closely related topics. The “Color” chapter of this manual shows how to apply patterns and colors to graphic objects. The “Application Framework” chapter goes into more detail about the construction of a window (the primary display surface). In the VisualWorks Cookbook, the “Custom Views” chapter covers string manipulation, fonts and other aspects of text (the primary visual object).

Background

Much like a newspaper photograph, a computer image is made up of tiny dots of color. Each dot makes one element of the picture, so it is known as a picture element—or *pixel*, for short.

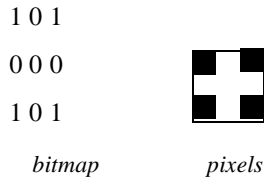


Figure 19-1 Bits in memory represent picture elements on the screen

On a black on white (monochrome) screen, each pixel is either on (black) or off (white). Its current state is represented in memory as either one (on) or zero (off). Thus, each bit in memory controls a single pixel, and the entire screen is represented as a two-dimensional array of bits. The array provides a map of the screen, so it's called a *bitmap*.

When the screen is capable of displaying more than two colors, a single bit is not sufficient to embody the range of choices. It may take two bits (where four colors are available) or three bits (for eight colors) or more. Though the “bitmap” is no longer a one-to-one mapping from bits in memory to pixels on the screen, it is still referred to as a bitmap.

Coordinate System

Each pixel represents one unit of width on the x-axis and one unit of height on the y-axis.

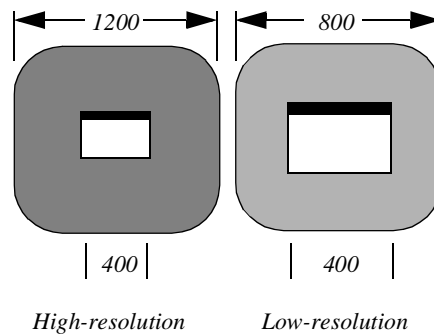


Figure 19-2 *Relative clarity remains constant regardless of the screen's resolution*

Different kinds of computer monitors vary in the number of pixels per inch or per centimeter, so a window that is 200 pixels wide will appear larger on a screen with lower resolution. However, the relative clarity of the window and its contents will be the same, which is generally the more important consideration.

SPIM graphics use a two-dimensional rectangular coordinate system, with x coordinates increasing from left to right on the graphic plane and y coordinates increasing from top to bottom.

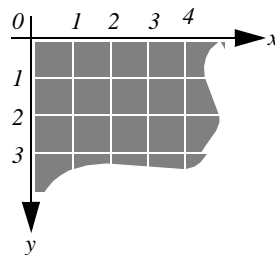


Figure 19-3 *Coordinate system*

Numbering starts from zero. Some windowing systems (such as the Macintosh's) place pixels between grid points, as shown in Figure 19-3, while other window systems (X and MS-Windows) place pixels on grid points. SPIM takes its lead from the window manager. This difference rarely matters, but it can cause a one-pixel misalignment in some circumstances and a "difference

of opinion” about whether the border of an object such as a polygon is to be repainted when that object is filled.

All graphic operations accept nonintegral coordinates, but such coordinates are rounded to the nearest integer. Coordinate values must be in the range from -32768 through 32767. These limits apply after translation, if any, has been applied. Translation is explained later in this chapter, on page 284.

Graphic objects are typically displayed in a window, and the window can be moved around by the user. For that reason, the origin of the window is used in most graphic operations rather than the origin of the screen. If the window has subviews, each subview maintains its own origin, and graphic operations use that origin. As a result, you rarely need to be concerned with translating coordinates when a window is moved or resized.

Points

An x-y coordinate pair is normally represented as an instance of **Point**. The following message creates a **Point** having an x-value of 100 and a y-value of 250. The spaces before and after the binary selector (**@**) are optional.

```
100 @ 250
```

You can also specify polar coordinates. The following example creates a **Point** whose coordinates lie on a circle of radius 100 at 45 degrees:

```
Point r: 100 theta: 45 degreesToRadians
```

Two constants are available, as well: **Point zero** returns `0@0`, and **Point unity** returns `1@1`.

A **Point** can perform the usual comparison and arithmetic functions—you can even add a scalar number to a **Point**, increasing both x and y by the desired amount. Table 19-1 lists some other useful messages you can send to a **Point**.

Table 19-1 Miscellaneous point functions

Message	Description
dist: aPoint	Distance from receiver to aPoint.
transpose	Answer a new point with x and y transposed.
grid: aPoint	Answer point nearest receiver on grid whose increment is specified by aPoint.
truncatedGrid: aPoint	As above, nearest preceding grid point
nearestPointOnLineFrom: point1 to: point2	As above, optimized for integer points.
dotProduct: aPoint	Dot product of receiver and aPoint
normal	Answer the receiver's normal vector.
unitVector	Answer the receiver scaled to unit length.

Rectangles

Rectangles are used in a variety of graphic operations, from setting the size of a window to specifying the bounding box of an ellipse. Because they are used so frequently, instances of **Rectangle** are especially well endowed with helpful behavior.

The usual way of creating a **Rectangle** is to send an **extent:** or **corner:** message to the origin point. Both of the following expressions create a rectangle 100 pixels wide, 250 pixels high, with its origin at 50@50:

```
50@50 extent: 100@250
50@50 corner: 150@300
```

The **extent:** message is more flexible because you need not calculate the bottom-right corner's absolute coordinates. However, in some situations, those coordinates may already be provided, while the width and height would have to be calculated.

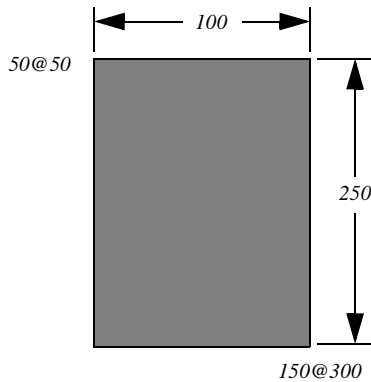


Figure 19-4 Creating a rectangle

When it is inconvenient to assemble the coordinates into `Points`, you can also create a `Rectangle` from the component x- and y-values:

Rectangle left: 50 right: 300 top: 50 bottom: 150

And when your application prefers not to distinguish between the origin and the corner point, you can let `Rectangle` do the comparison and create an instance:

Rectangle vertex: 300@150 vertex: 50@50

A suite of creation messages supports user-defined placement and sizing of rectangles, with or without grids.

Table 19-2 Rectangle placement and sizing messages

Message	Description
<code>fromUser</code>	User defines the rectangle via the mouse.
<code>fromUser: gridPoint</code>	User defines a rectangle whose size increment is determined by <code>gridPoint</code> .
<code>originFromUser: extent</code>	As above, with grid alignment controlled by <code>phasePoint</code> .

Table 19-2 *Rectangle placement and sizing messages*

Message	Description
fromUser: gridPoint phase: phasePoint	User defines the origin via the mouse; the size is determined by extent.
originFromUser: extent grid: scalePoint	As above, with the origin constrained to a grid whose spacing is defined by scalePoint. A button is assumed to be down
originFromUser: extent grid: scalePoint whileButton: button isDown: aBoolean	As above, with a specified button in a specified position.

Inquiring about a Rectangle's Dimensions

Once created, a Rectangle can tell you a number of things about its dimensions and its contents:

Table 19-3 *Messages for accessing a Rectangle's dimensions*

origin	width	area
corner	height	contains: aRectangle
left	leftCenter	containsPoint: aPoint
right	rightCenter	intersects: aRectangle
top	topCenter	
bottom	bottomCenter	

Scaling and Transforming Rectangles

Rectangles also handle a variety of scaling, merging and translating tasks.

Table 19-4 *Rectangle scaling, merging, translating messages*

scaledBy: aScalarOrPoint
expandedBy: aScalarPointOrRectangle
insetBy: aScalarPointOrRectangle
insetOriginBy: point1 cornerBy: point2
intersect: aRectangle

Table 19-4 Rectangle scaling, merging, translating messages

merge: aRectangle
areasOutside: aRectangle
moveBy: aPoint
moveTo: aPoint
translatedBy: aScalarOrPoint
align: point1 with: point2
amountToTranslateWithin: aRectangle

In the next section, you'll see how points and rectangles are used in the creation of display surfaces.

Display Surfaces

Graphic operations in Smalltalk display graphics on two-dimensional graphic media. All current graphic media are subclasses of the abstract class `DisplaySurface`, which represents host graphic media related to video display screens.

There are three types of display surface: `Window`, `Pixmap` and `Mask`. While a `Window` is used to display graphic objects on-screen, `Pixmaps` and `Masks` are used for behind-the-scenes manipulation of graphics. All three types of display surface employ a `GraphicsContext` as an intermediary between the surface and the objects to be displayed. We will introduce each of the three types of display surface, and then we will discuss `GraphicsContext`.

VisualWorks Windows

A `VisualWorks Window` corresponds to the window supplied by the host platform's window manager. It is a Macintosh window on the Macintosh, an X window on machines running X, and so on. For that reason, a `Window`'s border decorations and label bar take on the host window manager's look and feel.

`ScheduledWindow`, a subclass of `Window`, has a controller that permits the user to move, resize and close the window. To create a `ScheduledWindow` and then open it on the screen, execute the following:

```
ScheduledWindow new open.
```

By itself, a `ScheduledWindow` is not very useful. Try opening one and typing characters into it—as you will see, it does not provide application capabilities such as text editing. (To close the window, select **close** in its `<Window>` menu.) A `ScheduledWindow` handles the details of window resizing, raising and lowering, etc. It holds onto a `VisualComponent`, which is frequently a `View`. The view itself may contain subviews, and so on. Thus, `ScheduledWindow` is commonly described as being at the top of the view hierarchy.

The “Application Framework” chapter describes windows and window operations in more detail.

Pixmap

A `Pixmap` is the off-screen equivalent of a window. It is a rectangular surface, capable of storing an encoded color at each pixel location just as a window does. Unlike a window, the graphic contents of a `Pixmap` are not affected by damage events such as those caused by overlapping windows. Once you draw something on a `Pixmap`, you can be sure of retrieving that same object later. Another way of putting it is that a `Pixmap` retains its contents until they are explicitly overwritten. For this reason, a `Pixmap` is said to be a *retained medium*. It is not, however, retained across snapshots (i.e. quitting and restarting `VisualWorks`).

Pixmap and Color

A `Pixmap` stores a numeric color code for each pixel in its map. This code can be converted to an instance of `ColorValue` with the aid of a palette, which associates each numeric code with a specific color. Like a window, a `Pixmap` shares the display screen’s color palette. To get a `Pixmap`’s palette, send it the `palette` message. (Palettes are described on page 308).

The *depth* of a `Pixmap` is the number of bits needed to convey the range of possible colors in its palette. A two-color `Pixmap` has a depth of one because the two colors can be numerically conveyed by the state of a single bit. A four-color palette requires a depth of two, an eight-color palette requires three

bits, and so on. As with windows, **Pixmap**s have the same depth as the display screen.

A **Pixmap** can also tell you its **defaultPaint**, its **defaultBackgroundPaint** and its **defaultPaintPolicy**. For more information about color and paint policy, see “Policies for Rendering Color” on page 311.

Pixmap and the host clipboard

In windowing environments that support a graphics clipboard, a **Pixmap** can be copied from the host clipboard, via **fromClipboard**, and pasted to it, using **toClipboard**.

Masks

A **Mask** is used most often as a sophisticated clipping device that lets you trim unwanted parts of a picture. For example, you can display a detail from a complicated image by masking out surrounding regions. The mask can take any shape, such as a circle or even a car’s silhouette, so you can achieve advanced graphic effects involving merged images.



Figure 19-5 A cursor with and without a mask

For example, **Cursor** employs a mask to trim away “white” portions of the rectangular image, leaving only the desired shape (such as an arrow, or cross-hairs). Without a mask, the cursor would obscure a rectangular region of the display no matter what shape the cursor image was.

The value at each pixel location in a **Mask** indicates the portion of that pixel that is covered by the **Mask**’s graphic object. In a filled-circle mask, for example, a pixel that lies completely outside the circle has a value of zero. A pixel that is completely covered by the circle has a value of one. Intermediate values are not currently supported for masks, so borderline pixels get a rounded value of either zero or one.

Another way of thinking about a **Mask** is as a decal—“zero” pixels are the transparent backing while “one” pixels are the graphic object that is being transferred to a new medium. The standard term for this is *coverage*. Zero coverage implies no transfer of graphic content, while a **CoverageValue** of

one indicates complete transfer. Thus, each pixel location in a `Mask` has a `CoverageValue` rather than the `ColorValue` associated with a `Pixmap` pixel. The two types of coverage are designated as `CoverageValue` transparent (zero) and `CoverageValue` opaque (one).

Because a `Mask` is coverage-based rather than color-based, its contents cannot be directly *copied* onto a color-based display surface such as a window or a `Pixmap`—though you can *display* the mask on a color-based surface.

Host Residency of Display Surfaces

Display surfaces are created and destroyed by the host window manager. The bulk of a display surface, including its contents, are stored not in the Smalltalk object representing that display surface but in a host data structure referred to by the Smalltalk object. On a client-server window system such as X Window, the storage for a display surface resides in the server, not the client.

Because of their host residency, display surfaces other than scheduled windows do not survive snapshots. After a snapshot is restarted, your application must regenerate any required `Pixmap`s, `Masks` and unscheduled windows. (However, the `CachedImage` class gives you this ability—see “`CachedImage`” on page 295.)

Graphics Context

Every display surface uses an instance of `GraphicsContext` to manage graphic parameters such as line width, tiling phase and default font. Displaying operations are performed not by the display surface directly, but by its `GraphicsContext`.

Similarly, messages for modifying graphic parameters such as line width must be addressed to the appropriate `GraphicsContext`. That object applies the relevant parameters and then displays the object on the surface.

It’s important to understand that changes made to a `GraphicsContext` are forgotten immediately unless that `GraphicsContext` is stored by a variable in your application. This is because a display surface does not store an instance of `GraphicsContext`—an instance that could be altered in a persistent and often unintended way by unrelated graphic operations. Instead, a display surface manufactures a new instance of `GraphicsContext`, having the default parameters, each time it is sent the message `graphicsContext`.

This mechanism discourages tainting of the `GraphicsContext` in a way that ruins it for an unrelated graphic operation. Each graphic operation is responsible for setting up its own graphic context, and need not worry that the

context it gets from the view may have been modified. For this reason, you should never store a `GraphicsContext` in an instance variable or a class variable. If you must assign it to a variable, put it in a temporary variable so the changes remain local to the method.

Translation

Each `GraphicsContext` keeps track of an x-offset and a y-offset from the origin of the display surface. By setting these offsets, you can cause displayed objects to be shifted, or *translated*. This is useful as an alternative to altering the display coordinates of the graphic objects themselves.

Applying a translation to the `GraphicsContext` is sometimes more convenient than transforming the coordinates of individual display objects. For example, suppose you have two views each showing a portion of a graph. You can draw the entire graph on a large `Pixmap`, and then use translation to display the desired portion of the graph on each view.

A display surface's default translation is `0@0`. A `VisualPart`'s `GraphicsContext` has a default translation that reflects the object's position relative to the window's origin. For example, suppose the window occupies a rectangle 400 pixels wide and 500 pixels high. A view that occupies the lower right quadrant of the window would have a default translation of `200@250`.

Clipping

A `GraphicsContext` also maintains a *clipping region*—a rectangular viewport outside of which display objects are invisible (clipped away). You can control both the size and the location of this region.

Limiting the clipping region to an area that is smaller than the available display region can be useful when it is not convenient to alter the graphic object directly. Modifying the clipping region is equivalent to applying a rectangular `Mask` to the graphic object.

Use `clippingRectangle:` to set a `GraphicsContext`'s clipping region to a new rectangular area. To fetch the existing rectangle, send `clippingRectangleOrNil`, which returns `nil` when no clipping is in effect other than to the bounds of the display surface. If you would rather receive the display surface's bounds instead of `nil`, send `clippingBounds` to the `GraphicsContext`.

The clipping rectangle is specified in the coordinate system of the `GraphicsContext`. For a display surface, the default is `nil`; for a `VisualPart` such as a view, the default clipping region is the view's bounding box.

Line Characteristics

A `GraphicsContext` maintains three attributes specifically for lines and arcs: width, cap style and join style.

Line Width

Width refers to the thickness of the line, in pixels. The default is one pixel, and there is no practical maximum. The line is centered on the specified coordinates, so a 20-pixel horizontal line has 10 pixels of width above the coordinates and 10 pixels of width below the coordinates. The messages for fetching and setting line width are `lineWidth` and `lineWidth:`. The line width setting applies to lines, polylines, arcs and rectangular borders.

Line Cap Style

Cap style controls the appearance of line ends. Butt style (the default) provides no cap on the end; it is specified as `GraphicsContext capButt`. Rounded caps having a diameter equal to the line width are specified as `GraphicsContext capRound`. Rectangular caps that project beyond the line end by half of the line width are specified as `GraphicsContext capProjecting`. To retrieve and set the cap style, use `capStyle` and `capStyle:` messages.

Line Join Style

Join style refers to the appearance of the outside corner where two lines meet. Miter style (the default) features a squared-off joint; it is specified as `GraphicsContext joinMiter`. A rounded joint is specified as `GraphicsContext joinRound`, and a beveled joint is specified as `GraphicsContext joinBevel`. To retrieve and set the join style, use `joinStyle` and `joinStyle:` messages to a `GraphicsContext`.

Default Paint (color, opaqueness, and texture)

In SPIM, *paint* is the generic term for color, opaqueness and texture. While some graphic objects specify their own paint (`Image`, for example), the `GraphicsContext` needs to supply a default paint for uncolored objects. For a color-based display surface (`Window` or `Pixmap`), the default paint is `ColorValue black`. For a coverage-based surface (`Mask`), the default is `CoverageValue opaque`. To fetch and set the default paint, use `paint` and `paint:` messages.

A `GraphicsContext` also maintains a paint policy, which controls the rendering of paints that are not directly supported by the display device. For example, a color such as red is rendered with a suitable gray tone on a monochrome screen by default. The default on color systems depends on the depth of the screen (the range of colors it can render directly). Use `paintPolicy` and `paintPolicy:` to fetch and set the policy. Paints and paint policies are discussed in more detail in the “Color” chapter.

Tiling Phase

When a `Pattern` is used as the paint, the placement of the initial tile determines the location of all other tiles in the pattern. Shifting the origin of the first tile causes all tiles to be shifted similarly, affecting their alignment relative to borders and other graphic elements. The default tile phase is the `Point 0@0`, meaning the origin of the first tile is placed at the origin of the `GraphicsContext`’s coordinate system. Use `tilePhase` and `tilePhase:` messages to fetch and set the point.

`Patterns` and tiling are discussed in more detail on page 303. Tile phase also affects the placement of a halftone when the paint policy uses a dithering algorithm—discussed on page 313.

Default Font

The default font in a `GraphicsContext` is applied to a `String`, which has no font characteristics of its own. `Text` and `ComposedText` objects override the default font provided by the `GraphicsContext`. The default varies by platform—it can be determined by sending `defaultFont` to the display surface. To fetch the `ImplementationFont` currently in effect in a `GraphicsContext`, send a `font` message. To reset the default font (to an instance of `FontDescription`), send a `font:` message.

For more detailed information about fonts, see the “Text and Fonts” chapter of the *VisualWorks Cookbook*.

Displaying Geometrics

A visual object is expected to respond to a `displayOn:` message by rendering itself on the graphics context that is passed as an argument. Geometric objects such as circles and rectangles, however, are a special case. For example, sending `displayOn:` to a `Rectangle` is ambiguous because some situations require a filled rectangle while others call for a stroked rectangle (only the outline is displayed). For this reason, geometric objects respond to

`displayFilledOn:` and `displayStrokedOn:`, but not necessarily to `displayOn:`.

This presents a problem for containers, which only know how to say `displayOn:`—for example, when a view refreshes its display, it does so by sending `displayOn:` to its graphic elements. The solution to this communication problem lies in `FillingWrapper` and `StrokingWrapper`. A `FillingWrapper` translates `displayOn:` to `displayFilledOn:` for its component, and a `StrokingWrapper` performs the parallel service for its component. Thus, when you place a `Rectangle` inside a `FillingWrapper`, sending `displayOn:` to the wrapper causes a filled rectangle to be displayed.

In addition, a `StrokingWrapper` maintains the line width for its component (so it can accurately compute its bounding rectangle). The accessing messages are, predictably, `lineWidth` and `lineWidth:`.

As a convenience, geometric objects can provide their own stroking or filling wrapper. The `asStroker` and `asFiller` messages, when sent to a geometric, wrap it in the appropriate wrapper—then you can use `displayOn:`, as usual.

Graphic Attributes

While you can modify attributes when you display a specific object on the display surface, sometimes that is cumbersome. For example, suppose you have created a drawing application and your user has created a red rectangle. Your application must somehow associate redness with that rectangle so it will be rendered correctly each time the window is refreshed. The straightforward solution is to create a subclass of `Rectangle` that has a `paint` instance variable. You can see how this approach quickly escalates, however, because then all types of graphic objects must remember several possible attributes: color, line width, etc.

The `GraphicsAttributes` class provides a means of storing several common graphic attributes: line width, cap style, join style, phase, font and paint. A `GraphicsAttributesWrapper` is used to associate a set of graphic attributes with its graphic component.

Graphic Objects

A graphic object is an object that can be displayed on a display surface. Graphic objects can be colored or uncolored. An uncolored graphic object (a geometric object) simply describes a region in space; a colored graphic object (an image) also specifies the colors within the region.

Displaying a graphic object onto a graphic medium is conceptually the same as placing a stencil outlining the associated region on top of the medium and then painting only the area exposed by the stencil. For an uncolored object, the display operation paints every pixel within the stenciled area, using the `GraphicsContext`'s paint. Displaying a colored object paints the pixels of the exposed area with the colors of the corresponding pixels in the object. If a clipping rectangle is specified, it is intersected with the area covered by the graphic object to form the stencil.

The graphic objects supported by the imaging model are texts, lines, polylines, splines, Bezier curves, arcs, circles, rectangles and graphical images. In addition, the display surfaces themselves are also graphic objects—for example, you can display one window on another.

Texts

There are three types of text object: `String`, `Text` and `ComposedText`. A `String` is a collection of characters—it has no font information, so it uses the font provided by the `GraphicsContext`. `Text` and `ComposedText` override the default font with their own font(s). `ComposedText` also handles line-wrapping.

When a text object is drawn on a display surface, only the characters themselves are displayed. The background implied by the bounding box is masked out. This provides maximum flexibility when super-imposing text on a colored surface or on another graphic object.

The "Text and Fonts" chapter in the *VisualWorks Cookbook* discusses font control and other matters relating to the creation of text objects. In this chapter, we confine ourselves to the displaying of those objects.

The placement of text depends on its class. A `String` or `Text` is placed with its left baseline at the specified position. Because a `ComposedText` can span multiple lines, the origin of its bounding box is placed at the specified position.

Lines, Polylines and Polygons

A line segment connects two points, named `start` and `end`. A polyline connects three or more points (its collection of `vertices`). A polygon is a polyline that is filled rather than stroked. (A point is inside the polyline if an infinite ray originating from the point crosses the polyline an odd number of times. If the polyline is not closed, it is implicitly closed before the even/odd rule is applied.) Rectangles are treated specially.

All of these objects can be drawn by specifying a set of points to a `GraphicsContext`. Lines and polylines are displayed with the `lineWidth`, `capStyle` and `joinStyle` provided by the `GraphicsContext`. All three use the default paint of the `GraphicsContext`.

When your application draws geometric shapes that do not interact, it may be satisfactory to render them via the graphics context. But for long-lived or interacting geometrics, it is usually better to create an instance of `LineSegment` or `PolyLine`.

Rectangles

Rectangles can be created as described earlier, by specifying the origin and the opposite corner, rather than all corner points as with other polygons. `GraphicsContext` has specialized protocol for displaying rectangles, both filled and unfilled.

To display a filled `Rectangle`, send a `displayRectangle:` message to the desired `GraphicsContext`, first setting `lineWidth` and `joinStyle` if necessary. To display an unfilled rectangle, use a `displayRectangularBorder:` message.

Translation Protocol

Sometimes it's convenient to draw a polyline, polygon or rectangle as if its origin point were at 0@0, then position the object elsewhere relative to the display surface. For that reason, a variant of the displaying messages allows you to specify the point at which the object's origin is to be positioned.

The variants are as follows:

```
gc displayPolyline: pointCollection at: aPoint
gc displayRectangle: aRectangle at: aPoint
gc displayRectangularBorder: aRectangle at: aPoint
gc displayPolygon: pointCollection at: aPoint
```

Splines and Bezier Curves

Besides circular and elliptical arcs, which are discussed below, two kinds of curve are provided: `Spline` and `Bezier`. A `Spline` is similar to a polyline in that it connects a collection of vertices; the difference is that it smoothes the corners. A `Bezier` curve has a `start`, an `end` and two control points—each control point exerts gravity on the line segment connecting the start and end.

Each of these curve classes has a class variable for controlling the flatness of the curve. Both support comparison, intersection testing, scaling and transforming. A `Spline` can also be asked whether it folds back on itself (`isCyclic`).

Arcs, Circles and Wedges

An arc is a curved line defined by three elements of information:

- n The smallest rectangle that can contain the ellipse of which the arc is a segment (adjusted for line width).
- n The angle at which the arc begins, measured in degrees clockwise from the 3 o'clock position (or counterclockwise for negative values).
- n The angle traversed by the arc, known as the sweep angle. The sweep angle is measured from the starting angle (not necessarily the 3 o'clock position) and proceeds clockwise for positive values and counterclockwise for negative values.

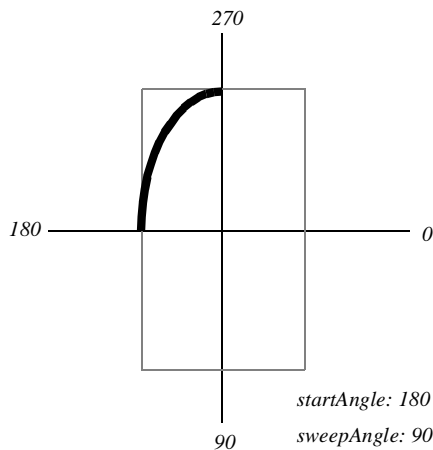


Figure 19-6 *Defining an arc*

A complete ellipse is an arc with a sweep angle of 360 degrees. A circle is an arc with a square bounding box and a sweep angle of 360 degrees.

A wedge is a filled arc (or circle or ellipse). If the arc does not describe a closed ellipse, the ends of the arc are connected to the center of the ellipse to define the filled region. The common case of a filled circle is referred to as a dot, and is defined by a diameter.

As with a straight line, an arc uses the `lineWidth` and `capStyle` of the `GraphicsContext` that draws it. When the line width is greater than one pixel, the arc is centered on the elliptical path. An arc or wedge uses the paint provided by the `GraphicsContext`.

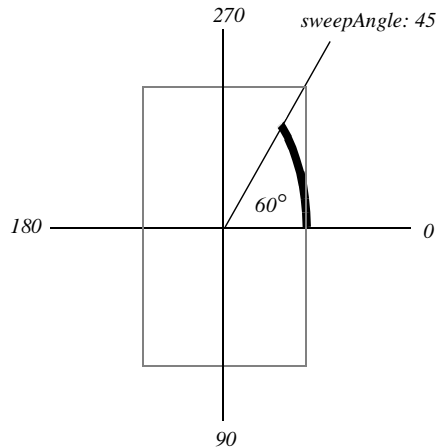


Figure 19-7 *The skewed coordinate system of a noncircular ellipse*

The angles are specified in the coordinate system of the ellipse. That coordinate system is skewed for noncircular ellipses. For example, the angle between three o'clock and a line from the center of the ellipse to the top right corner of the bounding rectangle may form an angle of 60 degrees in actuality but it is always specified as 45 degrees.

To make an object of an arc or a circle, use an instance of `EllipticalArc` or `Circle`. A wedge is an `EllipticalArc` that is filled rather than stroked.

A `Circle` and an `EllipticalArc` can perform comparisons, intersection testing, scaling and transformation. Either can provide its `center`, `startAngle` and `sweepAngle`. In addition, a `Circle` can provide its `radius`, `diameter` and `area`.

Graphical Images

An **Image** is a graphic object composed from a rectangular array of pixels. It is similar to a **Pixmap** and a **Mask** in many respects. The chief differences are:

- n An **Image** is stored in Smalltalk memory, so it survives across snapshots. For that reason, it is sometimes used as a storage device for **Pixmap**s and **Masks**, which die when you **Quit** from the system.
- n An **Image** is not a display surface, so you can't display other graphic objects on it as a means of assembling the desired picture.
- n An **Image** can be either color-based or coverage-based, depending on the nature of its palette.

Representation

Like **Pixmap**s and **Masks**, an **Image** employs a bitmap to represent its pixel colors or coverages. A very simple **Image** can be constructed by manipulating the bits in the map directly, but this is unwieldy for complicated pictures. Frequently, a scanner or a drawing tool is used to create the desired arrangement of pixels—then you can create an **Image** from the on-screen representation or from the bitmap.

The **Image** hierarchy of classes looks like this:

```
Object
  VisualComponent
    Image
      Depth1Image
      Depth2Image
      Depth4Image
      Depth8Image
      Depth16Image
      Depth24Image
```

Figure 19-8 *Image hierarchy*

Class **Image** is an abstract class providing the general protocol for images. Its concrete subclasses provide specific representations for images of different depths. As shown in the hierarchy, Smalltalk supports images with a packing depth (or bits per pixel) of 1, 2, 4, 8, 16 or 24. Images can, however, have different logical depths. For example, an image with a logical depth of 3 can be stored in a **Depth4Image**, wasting one bit per stored pixel.

For each pixel (picture element), an **Image** stores the value of the picture at that position—the color or coverage value of the pixel, depending on whether the image is color- or coverage-based.

To save space, **Image**s encode each pixel value as a nonnegative integer, rather than directly storing the color or coverage value. The system stores the encoded bitmap as a byte array. Along with the bitmap, an **Image** carries a zero-based palette mapping the encoded pixel values to the corresponding color or coverage values. The depth of an image is the number of bits available to store each encoded pixel value. An image of depth 1 can support encoded pixel values of 0 and 1, an image of depth 2 supports values from 0 to 3, and so on.

Image data is stored in a format that is independent of the native bitmap format of the host window system. For common operations, you need not concern yourself with this format, but we present it here for the sake of completeness:

- n Pixels are stored in row-major order. Within a row, pixels are ordered by increasing x-coordinates.
- n Rows are stored in top-to-bottom order (increasing y-coordinates).
- n Pixels are represented in chunky (Z) format, where bits of a pixel are stored contiguously.
- n For pixels of size greater than eight bits, the pixel bytes are stored most-significant-first.
- n For pixels of size less than eight bits, pixels are ordered most-significant first within each byte.

There are two approaches to creating an instance of **Image**—build it from scratch or import a picture that you have scanned in or assembled with a drawing tool.

Importing an image from another application involves copying it off the display screen. That assumes you have a means of displaying it, either via a drawing tool, a bit editor, or similar software.

When you send a **fromUser** message to the **Image** class, the cursor changes to cross-hairs. Drag a selection rectangle around the desired portion of the display screen, then release the <Select> button. (This only works if the colors in the captured area are all in the screen palette maintained by **Smalltalk**.)

Packed Rows

A row that has been padded to the appropriate multiple of 32 bits is called a *packed row* (i.e., packed with extra zeros). Reading and writing a row of the bitmap is often more convenient using the unpacked version. However, that involves creating an intermediate **ByteArray** containing one byte for each

pixel. In intensive applications, this wastefulness can become noticeable. In those situations, you can use an alternate set of bitmap accessors that relies on the packed row format:

packedRowAt: rowIndex
packedRowAt: rowIndex **into:** anArray
packedRowAt: rowIndex **into:** anArray **startingAt:** destinationIndex
packedRowAt: rowIndex **putAll:** anArray
packedRowAt: rowIndex **putAll:** anArray **startingAt:** sourceIndex

Compatibility with the Display Surface's Palette

An `Image`'s palette can be either color-based or coverage-based, as described on page 307. The type of palette determines what kind of display surface the image can be displayed on and copied to. A coverage-based `Image` can be displayed on any surface, just as a `Mask` can, while a color-based `Image` can be displayed on a `Window` or a `Pixmap`. When copying a region from an `Image` to a display surface, however, the two objects must have similar palettes.

To create a display surface bearing an `Image`'s contents, send `asRetainedMedium` to the `Image`. A `Pixmap` is returned when the `Image` has a color-based palette, and a `Mask` is returned when the palette is coverage-based. This operation is equivalent to creating a new `Pixmap` or `Mask` and then displaying the `Image` on it.

Image Processing

`Images` know how to perform the following transformations: growing, shrinking, flopping, rotating and filling. For each such message, a variant is provided for specifying a scratch `Image` to hold the return object—this avoids the creation of a new `Image` with each transformation.

Growing or Shrinking an Image

To increase the size of an `Image` in the x-dimension, the y-dimension, or both, send `magnifiedBy:` `aPoint` to the `Image`. The x and y values of `aPoint` are used as multipliers on the width and height of the `Image`.

Bit Processing

A variety of layering effects can be produced by combining two images (or two `ByteArrays`) with a filtering algorithm. A set of 16 predefined algorithms—called *combination rules*—are available. They are identified by the integers 0 through 15, though the more commonly used ones can also be accessed via mode constants in the `RasterOp` class. Thus, the message `RasterOp erase` returns the integer 4, which identifies the combination rule for erasing shared pixels from the destination image. (`RasterOp` is a graphics support class that performs raster operations on bitmaps.)

Images also support a tiling operation that is very similar to the copying operation in that it makes use of the combination rules. The source image is tiled onto a region in the destination image with the selected filter.

CachedImage

A specialized class called `CachedImage` has been created to combine the displaying speed of a display surface with the longevity of an `Image`. A `CachedImage` holds onto both a real image and a `Mask` or `Pixmap`. It displays from the display surface; after a snapshot has destroyed the display surface, `CachedImage` recreates it automatically from the stored `Image`. This eliminates the need for your application to recreate `Masks` and `Pixmaps` explicitly after a snapshot. For example, the system uses `CachedImages` to hold commonly used graphic elements such as the insertion-point triangle.

To create a `CachedImage`, you must supply the starting `Image`. From then on, you must treat it as a display surface (don't try sending image-processing messages to it, for example).

Cursors

A `Cursor` represents the pictorial element that tracks mouse movements. VisualWorks provides a wide variety of built-in cursors, or you can build your own. You can change the displayed cursor permanently or change it while a particular piece of code executes.



Figure 19-9 VisualWorks' built-in cursors

Several instances of `Cursor` are predefined for use by the system and in your applications. While some of these are highly specialized (such as the memory management cursors), many will be useful in your applications. Figure 19-9 illustrates them, along with their names. To access one of these predefined cursors, send the cursor name as a message to the `Cursor` class.

A `Cursor` is composed of two `Images`: a color image as well as a coverage-based image used to create a mask, each of which is 16 pixels on a side. Where the mask image is transparent, the cursor is transparent. Where the mask image is opaque, the colors of the colored image are visible.

Each `Cursor` has a *hot spot*, which is its control point. For example, the default `Cursor` is an arrow that points to the top left corner of the display: its hot spot is at the point of the arrow, or 1@1. The cross-hair cursor has its hot spot at the center, where the two lines cross, or 7@7.

Displaying a Cursor Temporarily During an Operation

When you just want to apply a distinctive cursor during a particular operation, place the affected code inside a block and use that block as the argument to a `showWhile:` message sent to the desired cursor.

At the end of a `showWhile:` operation, the cursor automatically returns to its original state, so your application doesn't need to keep track of the old cursor.

A less courteous approach is to change the cursor with a `show` message to the desired instance of `Cursor`, which leaves the cursor in its new state until overridden by another `show`. Since the system frequently changes the appearance of the cursor, as when framing a view, this approach is not very reliable and its use is declining in favor of `showWhile:`.

Icons

An Icon represents the pictorial element used to identify a collapsed window. To install the icon in a window is a simple matter—just send an `icon:` message to the desired window.

Animation

Animation is an illusion created by drawing a graphic object in successive locations and erasing it in the abandoned locations, perhaps modifying it slightly at the same time.

For each location at which the object is displayed, the basic steps are:

- n Store the background to be obscured
- n Draw the object
- n Restore the background

This direct approach is satisfactory in some limited circumstances but generally results in a side effect known as *flashing*.

Flashing is caused by the fact that the object is not visible during the time between its erasure at the old location and its depiction at the new location. It looks like a light flashing on and off. Eliminating flashing requires a more sophisticated technique for erasing, one that erases only the pixels that are not needed to depict the object in its new location. This mechanism is available in the form of the `follow:while:on:` method, defined by `VisualComponent` for all of its subclasses—`Image`, `ComposedText`, etc.

The `follow:while:on:` method provides smooth animation for a single object. It does not handle more complicated effects involving changes in the graphic object (a walking robot, for example) or multiple objects all moving at the same time. For those situations, it's generally better to employ a technique known as *double buffering*.

Double buffering involves drawing the next scene (on a `Pixmap`, typically) while displaying the current scene on a window. The `Pixmap` is then displayed on the window, an operation that is instantaneous compared with the separate displaying operations required to assemble the scene. The

Pixmap acts as a graphic buffer that stands in for the window's frame buffer—hence the term “double buffering.”

Integrating Graphics into an Application

Displaying graphic objects directly onto a window is fine for ad hoc testing. However, the window has no knowledge of such contents and so it has no way of repairing damage to the contents when another window overlaps it. In this section, we discuss techniques for integrating graphics into your application via the dependency mechanism.

For the purposes of this discussion, we recognize two types of graphic element. The first type is *static* because it never varies and therefore only has to be drawn once and then linked into the window's damage-repair mechanism (described below). The second type is *dynamic* because it changes to reflect some aspect of the application's model—a graphic temperature gauge, for example. A dynamic graphic object has to be redrawn in response to changes in the model as well as window damage.

Integrating a Static Graphic

Each window contains one *component*, which can be a view, some other visual component such as an image, or a **CompositePart** capable of holding any number of visual components, including other composites. Thus, the window contains a hierarchy of components. These components interact by sending messages “up the tree” and “down the tree.”

In particular, components send a notice of damage up the tree along with a rectangle representing the damaged area (typically the component's own bounds). The window responds by sending a copy of its **GraphicsContext** down the tree to the affected components with instructions to redisplay themselves. In practice, a **VisualComponent** (of which views and graphic objects are subclasses) that wants to redisplay its contents sends **self invalidate**. **VisualComponent** tacks on the bounding box as the damaged area, forwarding the following up the tree:

```
self invalidateRectangle: damagedRectangle
```

The `invalidateRectangle:` method translates the message into the form used by the dependency-control apparatus:

```
self changed: #invalidate with: aRectangle
```

The window's response is to send `displayOn: myGraphicsContext` to its component. If the component is a composite, it forwards the `displayOn:` message to each of its subcomponents. If it is a leaf component, it performs the appropriate displaying operations.

For a discussion of windows, views and the damage-repair mechanism, see the chapter "Application Framework."

For a static graphic object, it is sufficient to install it into this component hierarchy. To install it directly as the window's component:

```
aWindow component: aVisualComponent
```

To install it in a composite:

```
aComposite add: aVisualComponent at: aPoint.
```

Integrating a Dynamic Graphic

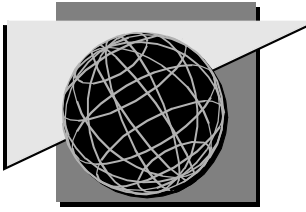
For a dynamic graphic object, it isn't reasonable to install it as a component because the original version would be redisplayed by the damage-control updates rather than the current version. Instead, we display such graphics on an installed component, usually a view, and we use the view's `displayOn:` method to display the current graphic. Since `displayOn:` is triggered by the dependency-control mechanism, this approach covers both damage events and model changes:

- n A damage event triggers `displayOn:` as described for static graphic objects.
- n A change in the model can be made to trigger `displayOn:` by sending a `self changed` message.

Thus, dynamic graphic objects need to be integrated into the view, and the model must notify its dependents when its state is changed:

- n The model sends `self changed` in the method that updates whichever aspect of the model the graphic object mirrors.

- n The view rebuilds the graphic object and displays it, in the `displayOn:` method.



Chapter 20

Color

In this chapter, we show you how colors and patterns are synthesized. We begin with the simple application of colors and patterns to uncolored objects such as lines and rectangles. Next we discuss the use of color with images and other graphic objects that store their own color information.

In the third section, we show how to convert an object created with one palette of colors to a different palette—for example, displaying a multicolored image on a monochrome display. There are various techniques for mapping one palette to another in such situations, and the final section details the three rendering techniques available in the system.

Types of Color

The Paint hierarchy provides three kinds of paint:

Paint

Pattern

SimplePaint

ColorValue

CoverageValue

We'll start with **Pattern** and then go on to the solid paints, which lead us naturally into other aspects of color such as palettes.

Pattern

A **Pattern** is an arrangement of pixels created by replicating a *tile* throughout a painted region much as ceramic tiles are laid out on a kitchen counter. The most familiar example is the “gray” background used by many window managers—an effect created by employing a four-pixel tile. The tile from which a pattern is generated can be an **Image**, a **Pixmap** or a **Mask**.

Sometimes, as with kitchen tile, the placement of that first tile can be critical to the success of the pattern. By default, the first tile is placed with its upper left corner at the origin of the display surface's `GraphicsContext`. You can adjust this location—called the *tile phase* because it controls the location of all tiles in the pattern—by sending a `tilePhase: aPoint` message to the `GraphicsContext`.

Coverage

A `CoverageValue` identifies the fraction of a pixel that is covered. Since a pixel, by its nature, must be displayed in its entirety, only the values 0 and 1 are typically used. (Fractional coverages can be specified, however, as explained in the discussion of coverage palettes on page 307.)

`CoverageValue` is the paint basis for `Masks`, as described in the previous chapter. An `Image` can also be coverage-based—such an `Image` typically is used as a storage medium for a `Mask`, which does not survive after the system is shut down.

A `CoverageValue` can be created by name or by value:

```
CoverageValue transparent
CoverageValue coverage: 0
CoverageValue opaque
CoverageValue coverage: 1
```

Color

A solid color is an instance of `ColorValue`, which is made up of red, green and blue components because most color monitors simulate complex colors by combining those primary colors. Since different display devices have different capabilities, Smalltalk chooses a substitute when the specified color is not available. The exact mechanism for this substitution is controlled by the paint policy, as described in “Policies for Rendering Color” on page 310.

On a monochrome display, for example, a red rectangle is rendered with a medium gray halftone, yellow appears as a lighter gray, and so on. In this case, the halftoning occurs on the basis of luminance so that the gray tone simulates the luminosity of the intended color.

There are three ways to create an instance of `ColorValue`:

- n Specify the color by name (for a select group of colors)

- n Specify the red, green and blue components
- n Specify the hue, saturation and brightness

Predefined Color

The easiest (but least flexible) way to create a `ColorValue` is to make use of a color constant provided by the `ColorValue` class. For example, sending a `blue` message to `ColorValue` returns the color blue.

RGB Color

On machines that support a large number of colors, the set of color constants is too limited for many applications. You can create more precise gradations of color by specifying the red, green and blue intensities.

Each intensity value is expressed as a fraction from zero through one. (When you open an inspector on a color, however, you will notice that the intensities are scaled up to integers for internal purposes.)

This approach also lends itself much more readily to algorithmic generation of colors, in which numeric values are represented as colors. A familiar example is the topographic map, which uses different shades of one or more colors to represent different elevations.

HSB Color

For many applications, such as those involving three-dimensional shading effects, neither of the color-creation schemes we've discussed is appropriate. How do you create the illusion of deepening shadows on a round surface, for example, if you can only add and subtract red, green and blue (rather than black)? In such situations, it is more useful to think in terms of hue, saturation and brightness.

In this color-mixing technique, known as HSB color, each component is specified as a value from zero to one, as with RGB color. However, the mix of red, green and blue in the color is communicated entirely by the hue. The intensity of that hue is controlled by the remaining parameters.

The *hue* is determined by a color's placement on a linear scale that begins at zero and ends at one. Both zero and one represent red, providing the needed circularity. Green is one-third of the way across the scale (hue = 0.333), and blue is two-thirds of the way (0.667). The secondary colors are in between the primaries: yellow (0.167), cyan (0.5) and magenta (0.833).

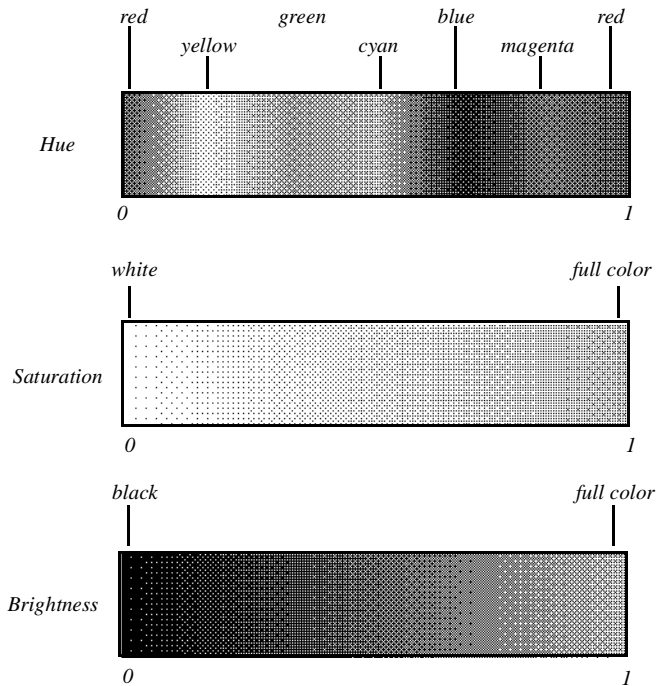


Figure 20-1 HSB color components, each value starting at zero and progressing toward one

Saturation is also a scale from zero to one, on which zero represents minimum vividness (pure white) and one represents maximum vividness (pure color). In other words, the higher the saturation, the more color is sprayed on a white background. You can also think of it as a way of mixing white into your color. Paleness makes an object appear farther away, so increasing the saturation tends to make an object appear closer.

Brightness is a similar scale, on which zero represents minimum brightness (pure black) and one represents the pure color. This is tantamount to spraying increasing densities of color on a black background, or mixing black into your chosen hue. This is useful for representing shadows.

Palettes

A **Palette** represents the collection of colors available for coloring any given pixel. For uncolored objects such as lines and circles, the palette is provided by the display surface. For colored objects such as images, however, you can create a custom palette. This is because an image encodes the color of each pixel as a numeric value in a bitmap, so a palette is needed to translate those numeric values to **ColorValues** or **CoverageValues**.

The palette class hierarchy is as follows:

```
Object
  Collection
    Palette
      ColorPalette
        FixedPalette
        MappedPalette
          MonoMappedPalette
      CoveragePalette
```

Coverage Palettes

A **CoveragePalette** is used by **Masks** and masking images, and specifies levels of transparency. It has a **maxPixelValue**, which determines the number of levels of transparency. Usually, **maxPixelValue** is set to 1, because a pixel can only be fully transparent (pixel value 0) or fully opaque (1).

However, it is conceivable that you would want to allow for intermediate levels of translucence. By specifying the **maxPixelValue**, you can create an image having any number of coverage levels (currently, masks are restricted to two levels).

Color Palettes

A color palette can have either of two representations: fixed or mapped. A **FixedPalette** breaks a numeric pixel value into three fields (red, green, and blue), each of which controls the intensity of that primary color. A **MappedPalette** stores a table of colors, so each numeric pixel value can be associated with an arbitrary color. A **MonoMappedPalette** is a **MappedPalette** that is specialized for the case in which the palette contains only black and white.

Mapped palettes are appropriate for images on color-mapped display screens and for images that use a small number of colors. Fixed palettes support true-color display screens that don't use a hardware color map. Such true-color screens typically support a large number of colors, making a mapped-palette representation impractical because of the size of color table required—a typical true-color screen has a depth of 24, which would require a color table with more than 16 million elements if a mapped-palette representation were used.

Color Palette Creation

Different types of palettes are created in different ways.

To create a mapped palette, send a `withColors:` message to `MappedPalette`, specifying an array of colors used to initialize the palette.

A fixed palette uses RGB values. Depending on the depth of the image, one set of RGB values might occupy 8 bits, 24 bits or 32 bits (or even something in between). When you create a fixed palette, you must arm it with the means to locate the red bits, the green bits and the blue bits. You do so by indicating the number of the bit that begins each RGB component as well as the maximum value for that component. In the creation message, the starting bit is called the *shift* value and the maximum value is called the *mask* value.

Eight-bit Color Palettes

Fixed palettes for 8-bit pixel values are structured in which the high three bits specify the red component, the next three bits the green component, and low two bits the blue component.

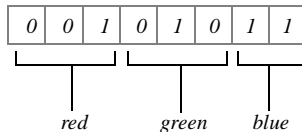


Figure 20-2 8-bit color palette

Performance Note about Palettes and Image Display

The composition of an image's palette greatly affects the amount of time required to display the image. An image can be displayed quickly in either of two circumstances:

- n Its palette is the same as that of the display surface
- n Its palette contains only two colors (not necessarily black and white) and those two colors can be rendered without halftoning.

Otherwise, displaying the image requires the creation of a temporary image that approximates the appearance of the image using the palette of the display surface. The creation of such a temporary image can take a substantial amount of time, especially if halftoning is required.

One noteworthy case requiring such automatic image generation is when an image created for one machine (a Macintosh workstation, for example) is transported to and displayed on another with a different screen palette (such as a Sun workstation).

To avoid generating a temporary image each time such an image is displayed, manually convert the image to the native palette once and then display the converted image rather than the original. For example, to convert an image to the color palette of the default screen (and therefore also of all windows and pixmaps on the default screen), perform:

```
anImage convertToPalette: Screen default colorPalette
```

By default, the `convertToPalette:` operation employs a `NearestPaint` renderer. In some cases, a different renderer (as described later) gives better results. A variant of `convertToPalette:` lets you specify the renderer:

```
anImage
  convertToPalette: Screen default palette
  renderedBy: OrderedDither new.
```

Device Color Map

The window manager's color map is not accessible from within Smalltalk. The screen's `colorPalette` is assembled based on that color map, as indicated in the following table. In the Comment column, "Fully populated" means the VisualWorks palette is the same as the device color map. "Partially populated" means VisualWorks uses only a portion of the color map, leaving enough unused cells so neighboring applications will have a chance to allocate their colors, too. When the platform provides a hint as to the default set of colors to be shared by applications, we use that set.

Table 20-1 Screen depth and associated windowing systems

Screen depth	Window system	Palette type	Comment
1	All	Mapped	Fully populated
2	All	Mapped	Fully populated
4	All	Mapped	Fully populated
8	X	Mapped*	Partially populated
8	MS-Windows	Mapped	Partially populated
8	Macintosh	Mapped	Fully populated
15	MS-Windows	Fixed	RGB values
16	All	Fixed	RGB values
24	All	Fixed	RGB values
32	All	Fixed	RGB values

* Using X, an 8-bit color map can be made fixed instead of mapped.

Policies for Rendering Color

When an image makes liberal use of the color turquoise, what should a black-and-white window do when asked to display that alien color? How about a color window that doesn't happen to have just the right shade of turquoise in its palette?

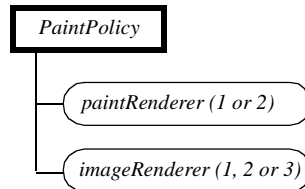
There are many conceivable techniques for making such decisions. Three of the most common techniques for rendering unknown colors are embodied in the following Smalltalk classes: `NearestPaint`, `OrderedDither` and `ErrorDiffusion`.

Any of the three can be used to render an image. Only the first two are appropriate for rendering paints. So a `PaintPolicy` object holds onto both a `paint-Renderer` and an `imageRenderer`, which may be the same.

You can ask a `GraphicsContext` for its `paintPolicy`, but make a copy of it before changing it unless you want the change to affect all display surfaces. This is because the default `Screen` maintains a policy that is shared by all `GraphicsContexts`.

To examine yours, inspect:

Screen default defaultCoveragePolicy
Screen default defaultColorPolicy



- ① *NearestPaint*
- ② *OrderedDither*
- ③ *ErrorDiffusion*

Figure 20-3 *The three kinds of color renderers*

The default renderers are determined as follows:

Table 20-2 *Default renderers*

NearestPaint	Used by Pixmaps and Windows on color systems
Ordered-Dither	Used by Masks on all types of screens
Ordered-Dither	Used by Pixmaps and Windows on monochrome or gray-scale systems

NearestPaint

A **NearestPaint** simply chooses the nearest available paint from the screen's palette. When that palette is limited, as on a monochrome screen, the results can be dramatic but are more often disappointing. To use a not-too-extreme example, imagine a magenta image on a chartreuse background. Both of these colors are luminous enough to be converted to white according to **NearestPaint**, so you're left with an empty rectangle of whiteness.

Where the colors in the image are more varied, the result is a stark rendition akin to a photograph printed with lots of contrast.

On color screens, however, **NearestPaint** usually produces satisfactory results and always gives the best performance.

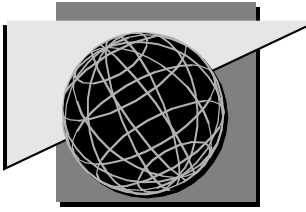
OrderedDither

An **OrderedDither** employs a threshold array to synthesize unrecognized colors by blending neighboring colors from the screen's palette. This has the effect of smoothing the transition from one palette color to the next in a continuous tone. While the result is often more pleasing than with **NearestPaint**, you pay a price in performance.

ErrorDiffusion

An **ErrorDiffusion** uses a more sophisticated blending algorithm. When it makes a choice from the screen's palette, it keeps track of how far off that choice was from the requested color. When this error accumulates sufficiently, the renderer uses the color on the other side of the threshold.

For example, suppose that a region of the image uses a red-brown color. The screen's palette has red and it has brown, but not the in-between color being requested. An **ErrorDiffusion** supplies red at first, but keeps track of the numeric difference between red and the red-brown. When that remainder accumulates to a breakpoint, a brown pixel is displayed even though the red-brown is closer to red. Thus, red and brown pixels are blended in proportion to the redness and brownness of the desired color. As you might expect, this technique is even more compute-intensive than the other two.



Chapter 21

Weak Arrays and Finalization

Prior to Release 4.0 of Objectworks\Smalltalk (the predecessor to VisualWorks), all object pointers (OOPs) were treated as *strong* pointers. A strong pointer is a reference that cannot be broken by any of the virtual machine's garbage collection mechanisms. Thus, if any object is reachable from the system roots via a chain of strong references, that object is exempt from being reclaimed as garbage by the object engine (OE).

In most cases, the fact that all references were strong was desirable. After all, most objects are not prepared to have the objects to which they refer suddenly disappear with the rest of the garbage. In some circumstances, however, strong references caused objects to live longer than their designers intended.

Suppose, for example, you wanted to profile the performance characteristics of an application. You might place some of the objects created by that application into an array so you could tabulate statistics on them. Unfortunately, the mere fact that you referenced these objects from such an array guaranteed that the objects would not be reclaimed as garbage even if the application code ceased to reference them.

This unintended side effect can now be avoided by using the new class `WeakArray`.

Weak Arrays

A `WeakArray` is similar to an ordinary `Array`, the prime difference being that a `WeakArray` references its elements weakly. Unlike a strong reference, a weak reference is ignored by the garbage collector.

Thus, when an element of a `WeakArray` is no longer referenced by any object other than another `WeakArray`, then that element is eligible for reclamation by the OE. During reclamation, the reference to that element is removed from the `WeakArray` and replaced by zero.

Only the indexable variables of the `WeakArray` class are weak references. The named instance variable, `dependents`, is strong. Further, this is the only class whose references can be weak. Even subclasses of this class can contain only strong references.

It is possible, however, to add named instance variables to this class, if you are willing to redefine the class. As stated above, such variables will be strong. The fact that this is the only class that can have weak references may seem to be a substantial restriction, but you can easily construct more complicated objects with a mix of strong and weak references by using a `WeakArray` as a subcomponent.

Finalization

`WeakArrays` also provide the system with a way of performing some final set of actions when an object expires. For example, an application might want to release some external resource when the objects using that resource have all been garbage collected. If the system were able to notify the application that the objects using the external resource had all expired, then the application would know that it was safe to loosen its hold on the external resource.

That mechanism involves sending a `changed` message to any `WeakArray` that has had one of its elements zeroed out as described above. This notification is then propagated to each of the dependents of that `WeakArray`, allowing them to take the actions necessitated by the death of the `WeakArray`'s element.

Of course, any such dependent will need to store whatever information it needs prior to receiving such notification, because the object that was once an element of the `WeakArray` will already have been destroyed. The dependent must also ensure that it can subsequently locate that information based solely on the dead element's index in the `WeakArray` (the dependent can find the index of a `WeakArray`'s dead element by invoking the `indexOf:replaceWith:startingAt:stoppingAt:` primitive).

To be more exact, the dependents of a given `WeakArray` are notified that one or more of its elements have expired as follows:

- n When an element of a `WeakArray` expires, the OE zeros out the slot in the `WeakArray` that was previously occupied by the now dead object.
- n In addition, the OE places this `WeakArray` on a finalization queue that is managed by the OE, and then signals the `FinalizationSemaphore`.

- n Signalling the `FinalizationSemaphore` causes the `FinalizationProcess` (which is generally waiting on the `FinalizationSemaphore`) to resume, and the `FinalizationProcess` then sends a `changed` message to every `WeakArray` on the finalization queue (it uses a primitive to fetch the `WeakArrays` that are on the finalization queue).
- n Eventually, every dependent of each `WeakArray` that suffered a loss will receive an `update` message.

Let's return to the example of an application that has a set of objects that act as proxies for external resources. The application wishes to free these external resources when the proxies are no longer in use. Further, assume that the proxies know which external resource they are associated with by virtue of a proxy instance variable that contains an external handle.

The application could arrange for the external resources to be freed automatically by simply placing the proxy objects in a `WeakArray` and copying their associated external handles into the corresponding locations of a strong `Array`. Then, when one or more of the proxy objects was no longer in use, the memory manager would reclaim the proxy object, zero out its location in the `WeakArray`, place the `WeakArray` on the finalization queue, and signal the `FinalizationSemaphore`, eventually resulting in an `update` message being sent to the application, assuming that the application had registered one of its objects as a dependent of the `WeakArray`. The application could then identify which proxy objects actually expired and free their associated external resources as follows:

```
weakArrayOfProxies
  forAllDeadIndicesDo:
    [:deadIndex | externalConnection
      freeResource: (externalHandleArray at: deadIndex)]
```

Note that there is also protocol to make nil the value at each dead index of the `WeakArray` as it is uncovered (`nilAllCorpsesAndDo:`) as well as for replacing the value with an arbitrary object (`forAllDeadIndicesDo:replacingCorpsesWith:`). Because these methods use the `indexOf:replaceWith:startingAt:stoppingAt:` primitive, which finds a given element and replaces it atomically, they can be used to prevent another process from mistakenly duplicating the finalization actions.

This scheme requires some extra work on the part of the application, because it forces the application to save a copy of the external handles in a parallel array. However, it completely avoids the problems that can occur if the proxy object that we are finalizing is resurrected, either by the code performing the

finalization or by some other code that happens to get a handle on the proxy object before it is actually destroyed by the OE and after the finalization action has been completed.

Instance variable for `WeakArray`:

`dependents` <nil | Object | DependentsCollection> those objects that must receive notification when one of the `WeakArray`'s elements dies.

Class variables for `WeakArray`:

`FinalizationProcess` <Process> that is responsible for sending a changed message to any `WeakArray` that has suffered a death

`FinalizationSemaphore` <Semaphore> that is signalled by the OE whenever a `WeakArray` has suffered a death

`QueueOverflowSignal` <Signal> that indicates that the OE's finalization queue has overflowed. It may be appropriate in this event to send a changed message to every `WeakArray`

WeakDictionary

A `WeakDictionary` is a dictionary whose `valueArray` is a `WeakArray`. Such a dictionary is fully protocol-compatible with `IdentityDictionary`. The lookup is done using `==` rather than `=`.

For finalization, `WeakDictionary` also stores an array of *executors* for its elements. The default executor for each element is a shallow copy of the element. An element's executor is responsible for finalization after the element has been reclaimed. An element with special finalization requirements should implement the `finalize` message, which is sent to the executor to actually perform the finalization. The default implementation of `finalize` in the `Object` class performs no finalization.

Instance variable for WeakDictionary:

```
executors<Array> the array in which the shallow copies of the values
are stored
```

HandleRegistry

A HandleRegistry is a WeakDictionary whose values all respond to a key message. The elements of a HandleRegistry are *registered* using their response to the key message as the dictionary key and using the element as the value. Access functions are all implemented as critical regions so that multiple processes can operate on an instance at the same time.

Instance variable for HandleRegistry:

```
accessLock<Semaphore> Mutex semaphore protecting accesses
```

The class hierarchy for these two classes is as follows:

```
Object ()
  Collection ()
    Set ('tally')
      Dictionary ()
        IdentityDictionary ('valueArray')
          WeakDictionary ('executors' 'accessLock')
            HandleRegistry ()
```

Finalization Example

To illustrate the finalization mechanism outlined above, we provide an example in the form of code for an Executor class. An Executor is an object that executes the last will and testament of a familyMember. To try it, enter the code into the system, then evaluate the expression in the class comment.

Class definition:

```
Object subclass: #Executor
instanceVariableNames: 'familyMembers familyWills'
classVariableNames: "
```

```
poolDictionaries: "  
category: 'Finalization-Example'
```

Class comment:

The Executor class is a simple example of how finalization can be achieved by using WeakArrays. After entering the code into the system, evaluate the expression: "Executor example inspect".

Instance Variables:

```
familyMembers  
    <WeakArray> containing the name string of each family  
member.  
familyWills  
    <Array> of blocks that will print the last will and testament of  
the corresponding person in the familyMembers array on the  
Transcript.
```

Instance methods for finalization:

```
readLastWillAndTestamentOfTheDeparted  
    "Read the will of each family member who has died."  
  
familyMembers nilAllCorpsesAndDo: [:deadIndex |  
    (familyWills at: deadIndex) value]
```

Instance methods for updating:

```
update: anAspectSymbol with: aParameter from: aSender  
    "Finalize all finalizable entries of aSender."  
  
(aSender == familyMembers and:  
    [anAspectSymbol = #ElementExpired])  
    ifTrue: [self readLastWillAndTestamentOfTheDeparted]  
    ifFalse: [^self]
```

Instance methods for accessing:

```
familyMembers: aWeakArray
```

```

familyMembers removeDependent: self.
familyMembers := aWeakArray.
familyMembers addDependent: self

```

```

familyWills: anArray
familyWills := anArray

```

Class methods for example:

example

```
"Executor example inspect"
```

```
| family wills familyLawyer |
```

```
family := WeakArray
```

```
  with: 'cain' copy
```

```
  with: 'abel' copy
```

```
  with: 'eve' copy
```

```
  with: 'adam' copy.
```

```
wills := Array
```

```
  with: [Transcript show:
```

```
    'Cain has died. Bequeaths his assets to the church.']; cr]
```

```
  with: [Transcript show:
```

```
    'Abel has died. Killed by Cain for his assets.']; cr]
```

```
  with: [Transcript show:
```

```
    'Eve has died. Bequeaths her assets to Abel.']; cr]
```

```
  with: [Transcript show:
```

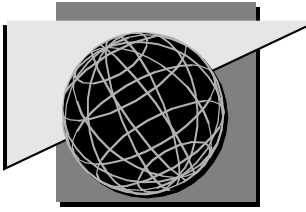
```
    'Adam has died. Bequeaths his assets to Eve.']; cr].
```

```
familyLawyer := Executor new.
```

```
familyLawyer familyWills: wills.
```

```
familyLawyer familyMembers: family.
```

```
^familyLawyer
```



Chapter 22

Parsing and Compiling

Although not often used directly, there are three classes that parse and compile Smalltalk programs: **Scanner**, **Parser** and **Compiler**. The **Scanner** parses a string into a sequence of tokens (numbers, names, punctuation, etc.) according to the lexical rules of the Smalltalk language. The **Parser** parses a string into a complete expression or method definition. The **Compiler** compiles a string into a method.

In contrast to many other classes, instances of these classes are nearly pure functions. They carry out a single operation and are then abandoned rather than retained.

These classes provide several more public messages than are documented here. These other messages serve more specialized uses within the compiler.

Scanner

To create an instance of **Scanner**, use **new**. To convert a string to a sequence of tokens, use **scanTokens:** as in the expression:

```
tokenArray := Scanner new scanTokens: aTextOrString
```

The string is interpreted approximately as though it were an **Array**, each word being converted to the equivalent literal (number, symbol, etc.) and installed as an element. However, the pound sign (#) that introduces a literal array is incorrectly treated like a binary operator, and the words 'nil', 'true', and 'false' are not treated specially. For example, the following expression is true:

```
(Scanner new scanTokens: '3.5 is: GPA') = #(3.5 #is: #GPA)
```

Parser

To create an instance of `Parser`, use `new`.

To extract the selector from the source string of a method:

```
selector := Parser new parseSelector: aString
```

For example, the following expression is true:

```
(Parser new
  parseSelector: 'from: here to: eternity
    ^eternity - here')
= #from:to:
```

To parse an entire method or a `doIt` (just like a method without the initial pattern), use an expression such as the following:

```
methodName := Parser new
  parse: sourceStream
  class: aClass
  noPattern: noPattern
  context: nil
  notifying: anEditor
  ifFail: aBlock
```

The `noPattern` argument is `true` for a `doIt`, `false` for a method. If the source was constructed by a program, or a `TextEditor` for interactive use, `anEditor` should be `nil`. If the source is not syntactically legal, this expression returns the result of evaluating `aBlock`; otherwise, it returns an instance of `MethodNode`. For more details, see the category 'System-Compiler-Program Objects'.

Compiler

`Compiler`'s class methods provide the most interesting public behavior, so there is usually no need to create an instance.

To evaluate a string as a Smalltalk expression:

Compiler

evaluate: aString
for: anObject
notifying: anEditor
logged: logFlag

The string will be evaluated as though it were the body of a method that had been invoked with `anObject` as the receiver. If `logFlag` is true, the string is written on the changes log. For example, the following expression returns 7:

Compiler

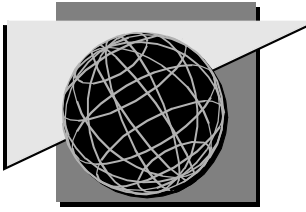
evaluate: 'x + y'
for: 3 @ 4
logged: false

As for parsing, `anEditor` should be nil for noninteractive use, or a `TextEditor` for interactive use. To compile a source method into a `CompiledMethod` object, use an expression of the following form:

aMethod := Compiler

compileClass: aClass
selector: aSymbol
source: aString

This message will rarely be useful, however. More useful methods in `Behavior` (such as `compile:notifying:`) perform the compilation and also install the result in the method dictionary of a class.



Chapter 23

Memory Management

This chapter explains how Smalltalk manages object memory. This information may be helpful in tuning certain memory-intensive applications. Keep in mind, however, that the facilities and policies described in this chapter are subject to change from release to release.

The chapter is divided into three sections. In the first section, we discuss the layout of memory. The second section describes the facilities for reclaiming unused memory space. These reclamation facilities adhere to specific policies, which can be modified to suit your needs. These policies are discussed in the third section.

Memory Layout

Smalltalk makes a number of demands upon the address space that is provided to it by the operating system. For example, each of the following can consume a fair amount of this address space:

- n The code and static data that make up the object engine
- n The dynamic allocations made by the “C” run-time libraries (such as **stdio** buffers)
- n The dynamic allocations made by the window-system libraries
- n The dynamic allocations required to accommodate the interrupt stacks (if any)
- n The dynamic allocations made by the OE

In this section, we are only concerned with the last item, allocations that are made by the OE. The OE allocates two types of memory space: a set of fixed-size OE spaces, and Smalltalk object memory.

Fixed-size OE Spaces

The OE allocates the following fixed-size memory spaces at system start-up time:

- n CompiledCodeCache
- n StackSpace
- n NewSpace
- n LargeSpace
- n PermSpace

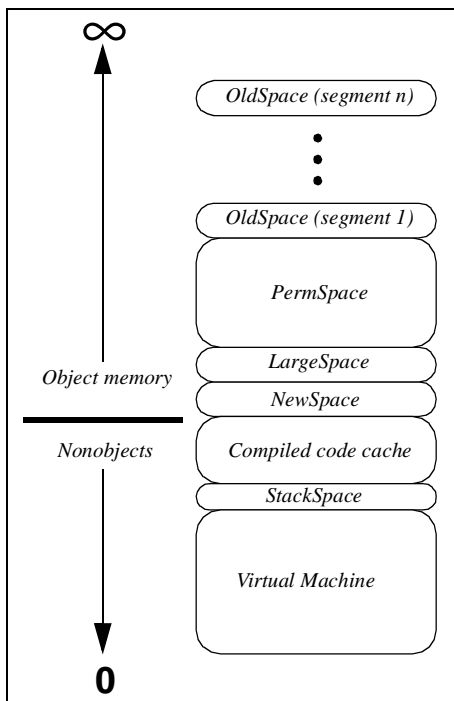


Figure 23-1 Memory Layout Map

Each of these spaces is used by the OE to house program elements of a particular type. The default size of most of these spaces can be altered at system start-up (see the class `ObjectMemory` for details).

CompiledCodeCache

To avoid the overhead of interpretation, the OE does not interpret the Smalltalk bytecode instruction set. Instead, it executes a given Smalltalk method only after that method has been compiled into the platform's machine code.

This compilation is performed automatically by the OE (the compilation is transparent to the user), and the resulting machine code is then placed in the CompiledCodeCache.

For example, when a Smalltalk method is executed for the first time, it is automatically compiled into machine code by the OE, and the resulting machine-code version of that method is then placed in the CompiledCodeCache so that it can be executed. Once executed, this method's machine-code is left in the CompiledCodeCache for subsequent execution.

As its name suggests, this space is only used as a cache, since it would require an excessive amount of memory to permanently house the machine-code version of every Smalltalk method. If the cache begins to overflow, those methods that have not been executed recently are simply flushed from the cache. This approach gives Smalltalk much of the speed that comes with executing compiled code, and most of the space savings and all of the portability that come with interpretation.

The size of this cache varies, depending on the density of the platform's instruction set. Default sizes range from 512 to 640 KB for platforms with CISC-based processors to 1 MB for RISC platforms. These sizes are large enough to contain the machine-code working sets of most applications.

StackSpace

Each process that is active in the virtual image (VI) is associated with a chain of contexts. These contexts come in two forms: the standard object format and the frame format. If a Smalltalk program tries to send a message to or access an instance variable of a given context, then that context must be in standard object form and housed in object memory. If it is not already in standard object form, then it is converted. The conversion to and from standard object format is transparent to the user.

On the other hand, when the method associated with a given context is actually being executed, that context must be in frame format and housed in the StackSpace. Once again, the conversion to and from this form is automatic. The frame format of the contexts has been designed to mate well with the typical machine's subroutine-call instructions.

Like the `CompiledCodeCache`, the `StackSpace` is used as a cache. If there isn't enough room in the `StackSpace` to store all of the contexts of all of the active processes, then the OE simply converts some of these contexts to standard object form and places them in object memory. Later, when the system needs to execute the methods associated with these contexts, it will convert the contexts back to frame format and place them back in the `StackSpace`.

The default size of this space varies from 20 KB to 40 KB, depending upon whether a given platform handles interrupts in another region of memory, or whether it needs to handle these interrupts in `StackSpace`. You can reduce the size of `StackSpace`, at the cost of forcing the OE to convert contexts more frequently from frame format to standard object format and back again. Or you can increase its size, at the cost of the additional memory.

NewSpace

`NewSpace` is used to house newly created objects. It is composed of three partitions: an object-creation space, which we call `Eden`, and two survivor subspaces.

When an object is first created, it is placed in `Eden`. When `Eden` starts to fill up (i.e., when the number of used bytes in `Eden` exceeds a threshold known as the *scavenge threshold*), the system's scavenging mechanism is invoked. Objects that are still reachable from the system roots are placed in whichever `SurvivorSpace` happens to be unoccupied at the time (one is always guaranteed to be unoccupied). Thereafter, objects that survive each scavenge are shuffled from the occupied `SurvivorSpace` to the unoccupied one. When the occupied `SurvivorSpace` begins to fill up (i.e., when the number of used bytes in the occupied `SurvivorSpace` exceeds a threshold known as the *tenure threshold*), the oldest objects in `SurvivorSpace` are moved to a special part of object memory called `OldSpace`. When an object is moved from `NewSpace` to `OldSpace`, it is said to be *tenured*. Both the *scavenge threshold* and the *tenure threshold* can be set dynamically (see the class `ObjectMemory` for details).

The default size of `Eden` is 200 KB, and each `SurvivorSpace` (they are always identical in size) is 40 KB.

LargeSpace

`LargeSpace` is used to house the data of large byte objects (bitmaps, strings, byte arrays, uninterpreted bytes, etc.). By "large," we mean larger than 1 KB.

When a large byte object is created, its header is placed in Eden and its data in LargeSpace. This arrangement permits the scavenger to move the object's header from Eden to a SurvivorSpace without having to move the object's data. In fact, the data that is housed in LargeSpace is only moved when LargeSpace is compacted, as part of a compacting garbage collection or to make room for another large byte object or in preparation for a snapshot.

Of course, the data of any object can be housed in LargeSpace, but small objects and large pointer objects are only placed in LargeSpace if there is no other place to house them. The data of large pointer objects is not housed in LargeSpace because it would take up valuable space without saving the scavenger much work. (Since such data is composed of oops, the scavenger has to scan it anyway, and it's not expensive to move the data while scanning it.)

If there are too many large objects to fit in LargeSpace, older ones are moved to object memory proper.

When the amount of data housed in the LargeSpace exceeds a threshold known as the `LargeSpaceTenureThreshold`, the scavenger is informed that it should start to tenure the headers of large objects. During the next scavenge, the headers of the oldest large objects are tenured to OldSpace. However, the data of these large objects will not be moved from LargeSpace until the allocator actually runs out of space in LargeSpace. Only at that time will the data of these older large objects be moved to OldSpace. The `LargeSpaceTenureThreshold` can be set dynamically.

The default size of LargeSpace varies from 200 KB to 500 KB, depending on whether the backing stores for windows on a given platform are allocated as Smalltalk objects or not.

PermSpace

PermSpace is used to hold all semi-permanent objects. Because they are rarely ready to die, the objects housed in PermSpace are exempt from being collected by any of the reclamation facilities other than the global garbage collector. By removing such objects from OldSpace, the time required to reclaim the garbage that may be present in OldSpace is reduced many times.

In the delivered image, most of the objects in the system are housed in PermSpace. Newly created objects that are placed in OldSpace by the scavenger are not automatically promoted to PermSpace. Moving Oldspace objects into PermSpace (and thus improving the efficiency of garbage reclamation) is done by creating an image by choosing **File?Perm Save As...** in the VisualWorks main window. Creating an image in this way is similar to making a snapshot except that all of the objects that are currently in OldSpace will be

promoted to PermSpace when the new image is loaded back into memory at startup time. On the other hand, you can cause all of the objects in PermSpace to be loaded into OldSpace at startup time if you create an image using **File?Perm Undo As...** in the VisualWorks main window.

Note that the current state of object memory is not changed by the act of creating a new image using the perm-save or perm-undo commands. In other words, only the newly created image will contain a modified PermSpace. For example, if you use **File?Perm Save As...** to create an image and later in that same session you create a normal snapshot on top of that image, PermSpace will be unaffected.

To place your application code in PermSpace, do the following steps before deploying an image containing the application:

1. Create an image using the **File?Perm Save As...** command. Then choose **File?Exit VisualWorks...** and start the new image. All of the objects that were formerly in OldSpace will be loaded into PermSpace, including the application code.
2. A number of transient objects will also inhabit PermSpace, such as those needed to display windows on the screen—to remove them, perform a global garbage collection.
3. Create a normal snapshot.
4. To make subsequent loads on the same platform even faster, you may want to load the new image back into memory and perform one last snapshot. This last step is useful because the global garbage collector compacts the objects in PermSpace, which forces the load code to relocate these objects at startup time. By performing one extra snapshot, these objects will not need to be relocated on subsequent loads when it is possible for the OE to load them into their former locations. (To produce a shareable image on a Sequent computer, this final step is required)

Smalltalk Object Memory

In addition to the above fixed-size memory spaces, the system also manages a variable-size space known as OldSpace. OldSpace is a warehouse for all objects that are not housed in one of the fixed-size spaces described above.

OldSpace

Unlike the above spaces, however, the size of OldSpace is not frozen at startup time. Instead, it is configured at startup time with a default of 1 MB of free space. When OldSpace begins to run short of free space, the system has

the option of increasing its size. This growth is accomplished by means of a primitive that attempts to acquire additional address space from the operating system. The decisions regarding when to grow OldSpace and by how much are controlled by an instance of **MemoryPolicy**. See that class for the default policy.

Although OldSpace can be thought of as a single contiguous chunk of memory, it is implemented as a linked list of segments occupying the upper portion of the system's heap. OldSpace's growth capability dictates this approach because, for example, I/O routines frequently allocate portions of the heap for their own use, creating intervening zones that divide OldSpace into separate segments. In a growing system, then, OldSpace may be composed of multiple segments. When these multiple segments are written out at snapshot time, they are stripped of their free space to save file space. In addition, to avoid fragmentation, they are coalesced into one large segment when the snapshotted image is loaded back into memory at startup time.

Each OldSpace segment is composed of two parts: an object table (OT) that is used to house the old objects' headers, and a data heap that is used to house the objects' data. The data heap is housed at the bottom of the segment and grows upward; the object table is housed at the top of the segment and grows downward. Both the object table and the data heap are compacted by the compacting garbage collector.

Since the OT and the data heap grow toward each other (thereby consuming the same block of contiguous free space from different directions), the system should never run out of space for new object headers while still having plenty of space for object data, and vice versa. Nor is there any arbitrary limit on the total size of OldSpace, the total size of a given OldSpace segment, or the number of OldSpace segments that can be acquired. The only memory-related resource that the system can run out of is address space. On real-memory machines, this translates to available real memory. On virtual-memory machines, it corresponds to available swap space.

In addition, the system maintains a threaded list of free object table entries and a threaded free list of free data chunks. The incremental garbage collector recycles dead objects by placing their headers and bodies on these lists, and the OldSpace allocator tries to allocate objects by utilizing the space on these lists before dipping into the free contiguous space between the object table and the data heap of each segment. Finally, a certain portion of the free contiguous data is reserved for use by the OE to ensure that it can perform at least one scavenger in extreme low-space conditions, thereby providing the system with one final opportunity to take the appropriate action.

Remembered Table

The remembered table (RT) is a special table that contains one entry for each object in OldSpace or PermSpace that is thought to contain a reference to an object housed in NewSpace.

The objects in the RT are used as roots by the scavenger—if an object is not transitively reachable from either the RT or the StackSpace, it will not survive a scavenge. The RT is expanded and shrunk as needed by the OE. It is expanded if the OE tries to store more entries than the RT can currently house, and it is shrunk during garbage collections when it has become both large and sparse, which can occur if a large number of entries were added and subsequently removed.

OldRemembered Table

The old remembered table (OldRT) is a special table that contains one entry for each object in PermSpace that is thought to contain a reference to an object housed in OldSpace or LargeSpace.

The objects in the OldRT are used as roots by the incremental garbage collector and the compacting garbage collector—if an object is not transitively reachable from the OldRT, it will not survive a garbage collection. The OldRT is expanded and shrunk as needed by the OE. It is expanded if the OE tries to store more entries than the OldRT can currently house, and it is shrunk during garbage collections when it has become both large and sparse, which can occur if a large number of entries were added and subsequently removed.

Facilities for Reclaiming Space

The OE has several facilities for reclaiming the space occupied by objects that are no longer accessible from the system roots:

- n Generation scavenger
- n Incremental garbage collector
- n Compacting garbage collector
- n Global garbage collector
- n Data compactor

Except for the scavenger, the OE does not invoke these facilities directly. Policy decisions such as this are controlled at the Smalltalk level—see the `ObjectMemory` and `MemoryPolicy` classes for these default policies.

Generation Scavenger

The primary reclamation system is a generation scavenger. The scavenger flushes objects that expire while residing in NewSpace (which typically applies to more than 95 percent of objects).

Briefly, the scavenger works as follows. Whenever Eden is about to fill up, the scavenger is invoked. It locates all of the objects in Eden and the occupied SurvivorSpace that are reachable from the system roots. It then copies these objects to the unoccupied SurvivorSpace. Once this copying is done, Eden and the formerly occupied SurvivorSpace contain only corpses—they are effectively empty and can be reused. The scavenger uses the objects in the remembered table and the objects referenced from the StackSpace as roots.

The scavenger's operation is imperceptible to the user. To ensure that this is so, the scavenger will start to tenure objects from NewSpace and place them in OldSpace if the number of survivors starts to slow down the speed of the scavenger's operation.

Incremental Garbage Collector

Unlike the scavenger, which only reclaims objects in NewSpace and LargeSpace, the incremental garbage collector (IGC) reclaims objects in OldSpace, NewSpace and LargeSpace. It does so incrementally, recycling dead objects by placing their headers and their bodies on the appropriate threaded free list.

The IGC can be made to stop if any kind of interrupt occurs, or it can be made to ignore all interrupts. In addition, you can specify the amount of work that you want the IGC to perform, both in terms of the number of objects scanned or the number of bytes scanned—it will stop as soon as either condition is satisfied.

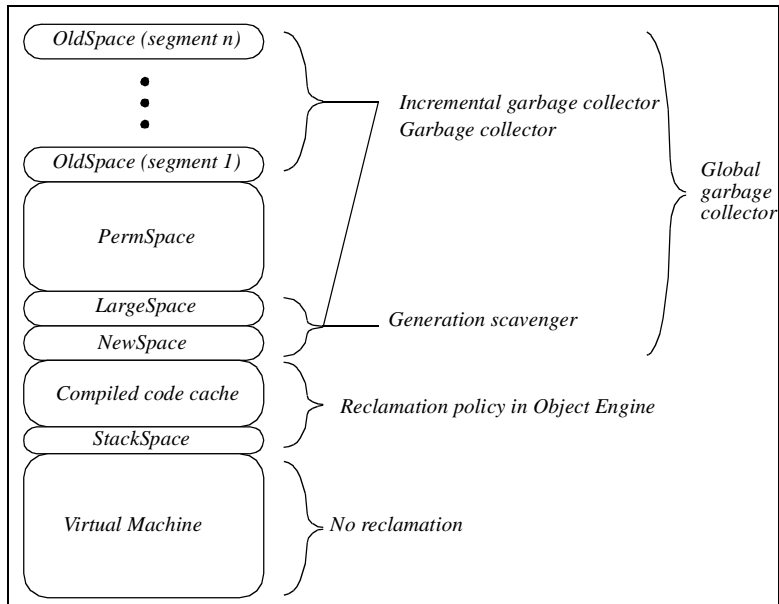


Figure 23-2 VisualWorks' reclamation facilities

The IGC has five distinct phases of operation:

- n Resting — the IGC is idle.
- n Marking — the IGC is marking live objects.
- n Nilling — the IGC is nilling the slots of WeakArrays whose referents have expired.
- n Sweeping — the IGC is sweeping the OT, placing dead objects on the threaded free lists.
- n Unmarking — the IGC is unmarking objects as a result of the mark phase being aborted, either at the user's request or because the IGC ran out of memory to hold its mark stack.

The typical order of operation is:

1. resting
2. marking
3. nilling
4. sweeping
5. resting

The unmarking phase is only entered if the mark phase is aborted, and it leaves the IGC in the resting phase when it is finished unmarking all objects. Each of the above phases is performed incrementally; that is, each can be interrupted without losing any of the work performed prior to the interruption. The IGC never performs more than one phase per invocation. This provision permits clients to specify different workloads and different interrupt policies for the different phases. Consequently, clients will need to wrap their calls to the IGC in a loop if they want it to complete all of the phases. There is protocol for doing this in the `ObjectMemory` class.

The OE never invokes the IGC directly. Only Smalltalk code can run it. A typical memory policy might be to run the IGC in the idle loop, in low-space conditions, and periodically in order to keep up with the OldSpace death rate. See the `MemoryPolicy` class for the default policy.

Compacting Garbage Collector

The compacting garbage collector is a mark-and-sweep garbage collector that compacts both object data and object headers. This garbage collector marks and sweeps all of the memory that is managed by the OE except for PermSpace, whose objects are treated as roots for the purposes of this collector. This garbage collector is never invoked directly by the OE, since the duration of its operation could be disruptive to the Smalltalk system.

Global Garbage Collector

The global garbage collector is a mark-and-sweep garbage collector that is identical to the compacting garbage collector except that it marks and sweeps all of the memory that is managed by the OE, including PermSpace. This garbage collector is never invoked directly by the OE, since the duration of its operation could be disruptive to the Smalltalk system.

You might want to invoke the global garbage collector when you suspect that there are many garbage objects in PermSpace. This would reduce the size of the image file produced by a subsequent **File?Save As...** It would also reclaim the space occupied by garbage objects in OldSpace, NewSpace and LargeSpace that are only kept alive by references from garbage objects housed in PermSpace.

Data Compactor

The system also has an OldSpace data compactor. Because this facility does not try to compact the object table, or mark live objects, it runs considerably

faster than either of the two garbage collectors. It should be invoked when OldSpace data is overly fragmented.

Memory Policy Classes

The system contains two classes that control all reclamation and growth policy: `ObjectMemory` and `MemoryPolicy`. The OE only supplies the base mechanisms for growing object memory and reclaiming dead objects. It is up to the Smalltalk memory-management code to utilize these mechanisms in a judicious manner.

ObjectMemory

An instance of `ObjectMemory` represents a snapshot of object memory as it existed when that instance was created. The information contained in this object can be used to guide policy decisions for managing object memory (see the class `MemoryPolicy` for one such policy). This class also contains protocol for manipulating the state of object memory. In general, if you want to access the current state of object memory, you would create an instance of this class and then send messages to that instance. If, on the other hand, you want to directly manipulate the state of object memory (for example, to grow object memory, to compact object memory, or to reclaim dead objects that exist in object memory), you would do so by sending a message directly to the class itself.

Because the information contained in this class is implementation dependent and because it may vary from release to release, it is recommended that this information only be accessed directly by the low-level system code that implements the various memory policies. Such policy objects should provide an adequate set of public messages that will permit high-level application code to influence memory policy without resorting to implementation-dependent code.

MemoryPolicy

This class implements the system's standard memory policy. Memory policy objects are given the opportunity to take action during the following circumstances:

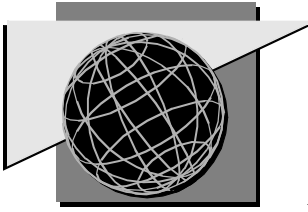
- n During the idle loop
- n When the system runs low on space

In addition, memory policy objects are responsible for determining precisely what constitutes a low-space condition.

An instance of `MemoryPolicy` will take the following actions in these circumstances.

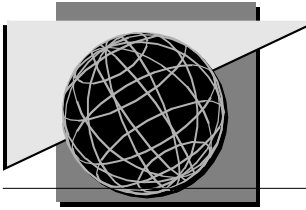
Table 23-1 *Actions Taken by MemoryPolicy*

Action	Description
idle-loop action	Runs the incremental GC inside the idle loop, provided that the system has been moderately active since the last idle-loop GC. Lets the idle-loop GC run until it is interrupted.
low-space action	<p>Responds to true low-space conditions. If the system is biased toward growth, then it attempts to grow object memory. If, however, it is not biased toward growth, or if object memory cannot be grown, then it tries various ways of reclaiming space. Failing that, it tries one last time to grow object memory. Failing that, it summons the low-space notifier.</p> <p>The most interesting of these steps is the reclamation step. An instance of this class will perform a full, compacting GC only if the free entries in the object table are consuming a significant percent of OldSpace. If, on the other hand, a compacting GC is not needed, the policy object will try to reclaim space by simply finishing the incremental GC (if one is currently in progress). If that doesn't free up enough space, then the incremental GC is run from start to finish without interruption. Finally, a data compaction is performed if OldSpace is sufficiently fragmented.</p>



Part IV

Application Delivery



Chapter 24

Overview of Application Delivery

When you have finished programming your application, you need to extract it from the VisualWorks environment in a form that makes it available and ready for use by your intended end users. This process is called *delivering* or *deploying* an application.

Different Ways to Deliver an Application

There are as many ways to deliver an application as there are to develop one:

- n You can deliver your application as a single image by removing unwanted code from your development image (known as *stripping* the image), saving the image, and then delivering that image to your customers, *or*
- n You can divide your application into small, separately-loadable units called *parcels* and deliver any number of parcels with a minimal base image, *or*
- n You can deliver part of your application in the image and parts as accompanying parcel files.

Single Image File

A single image file works well for saving and quickly restarting a system configuration. A single image file is often too large for easy distribution and too large-grained to provide adequately for the individual support of subsystems or sub-applications.

Parcels

Parcels, files that can contain application objects in a non-textual format, that can be rapidly loaded into an image without the use of a compiler. Parcels:

- n Allow you to deliver a very small base image.
- n Can be loaded at startup or run time.

- n Allow you to incrementally update your application without supplying an entire new image.
- n Allow you to customize your application at run time based on the needs to various kinds or users.
- n Allow you to tailor the memory footprint of your running application by providing a wider range of delivery configurations.

Development and Deployment Life-Cycle

There are a variety of processes for writing, parceling, and deploying applications. At one extreme, you can write and test your entire application and then break it into parcels for delivery. At the other extreme, you can write, test, and deliver your application one parcel at a time. These two methods are outlined below.

In either method, if you forget something in the deployment image, you can either:

- n Rebuild the deployment image.
- n Put the missing piece in a parcel.

Method 1: Delivery Combined with Development

If you are just starting development on an application or near the beginning of the project, you are in a good position to incorporate delivery as part of your development effort. You can write your application parcel by parcel, beginning with core parcels or ones that must be present at start-up and working your way through the application:

1. Create a deployment image.
 - Assess the anticipated needs of your application and create a deployment image that has the required VisualWorks support classes. This deployment image may be as simple as a shell that is capable of loading parcels for testing.
2. Build some discrete part of your application. It may have stubs for parts that are not yet built. Keep your delivery goals in mind.
3. Parcel that part of your application.
4. Test the parcel with the deployment image.
5. Repeat steps 2-4 until done.

Method 2: Delivery After Development

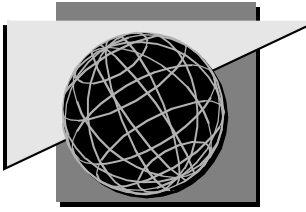
If you have a completed application that you want to deliver or are near the end of a project, you can save application delivery for the final step:

1. Develop your application as you usually do.
2. Divide your application into parcels based on your delivery goals and save those parcels to parcel files.
3. Create your minimal base image (*deployment image*) by stripping your development image.
4. Test your deployment image with your parcels. Fix as needed.

More Information

This part of the *VisualWorks User's Guide* is divided into three chapters:

- n Chapter 25, "Parceling an Application," describes parcels in detail and explains how to create and load them.
- n Chapter 26, "Creating a Deployment Image," describes a deployment image in more detail and explains how to create, start, and debug them.
- n Chapter 27, "Creating Applications without Graphical User Interfaces," explains how to create and deliver applications that do not rely on direct user interaction, and may run on computers that have no console or windowing system.



Chapter 25

Parceling an Application

What Are Parcels?

A *parcel* is a group of implementation objects gathered together into a single unit. A parcel is a fragment of your application, usually grouped by sub-application or deployment needs. They are discrete, loadable units that can be incrementally updated and changed.

an application's class and methods

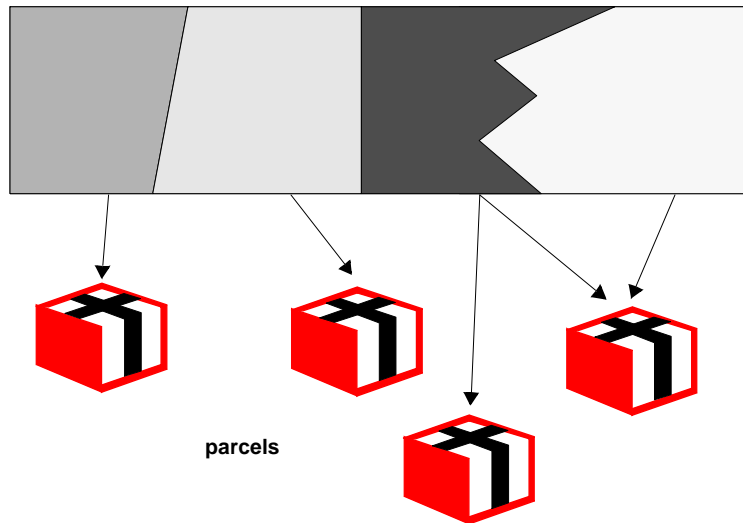


Figure 25-1 Application Divided Into Parcels

Characteristics

Parcels are instances of class `Parcel`. Each parcel has a name and a dictionary of arbitrary property strings (including a version string and a comment string).

Contents

A parcel may contain:

- n An arbitrary group of one or more class definitions.
- n An arbitrary group of one or more method definitions. A method whose class is not defined in the same parcel is called an *extension* to that class.
- n Named object such as data objects.

A single class definition, method definition, or object may appear in any number of parcels. A parcel may contain a class definition without containing any of the protocols or method definitions in the class.

Restrictions

A class's instance and class side definitions must be contained in the same parcel; they cannot be broken apart. A parcel cannot change a class definition that is already in the image.

Named objects have the following restrictions:

- n They cannot be instances of the following classes:

CDatum	Context	Controller
Exception	ExternalInterface	GraphicsContext
GraphicsDevice	GraphicsHandle	GraphicsMedium
LensContainer	LensGlobalDescriptor	LensSession
OSHandle	Process	Semaphore
Signal	VisualPart	WeakArray

- n They cannot be block closures that have stack contexts associated with them.

Parcel Files

Parcels are saved in parcel files. Parcel files contain only compiled code in a binary format. They do not contain the source code for the defined classes and methods they contain. As a result, parcel files can be quickly and atomically loaded into an environment without the use of a compiler.

Creating Parcels

Deciding What to Parcel

Dividing an application into parcels is as much an art as a science. Your goal should be to create parcels that are small enough to be distributed and loaded quickly. A guideline is to keep parcel files smaller than 100K, but parcel files may be any size that accommodates the needs of your application.

When parceling an application think about how you want to load the parcels into your deployment image or running application. Which parts must always be running? Are there parts that should start up together? Are all of the end users going to use all of the application's features, or are there some features that might be used infrequently and thus should be loaded only when needed?

Keep in mind dependencies between parts of your code. Be conscious of:

- n Subcanvases.
- n Embedded and linked data forms.
- n Inherited behavior.
- n Resources such as bitmaps that are used from a central location.
- n Class variables that are used by other classes. For example, if one of your classes keeps the name of the application's working directory in a class variable, it should be loaded first.

Specifying Parcels and their Contents

Once you have an idea how you want to divide your application, you can begin putting your application in parcels:

1. Open the Parcel List by choosing **Tools?Parcel List** from the VisualWorks main window. The Parcel List shows all of the parcels in the image. Initially, the Parcel List is empty.

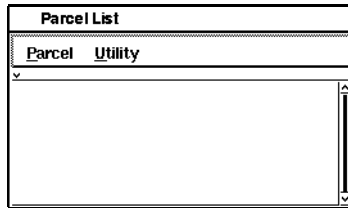


Figure 25-2 Parcel List

(For a complete description of the Parcel List, see page 161.)

2. Create a new parcel by choosing **Parcel?New** and providing a parcel name. A parcel name may be any string; however, the Parcel List strips out blank spaces before and after parcel names and reduces embedded series of blank spaces to a single space. Parcel names must be unique within the image.
3. Display the contents of the new parcel by double-clicking on the parcel name. VisualWorks opens the Parcel Browser.

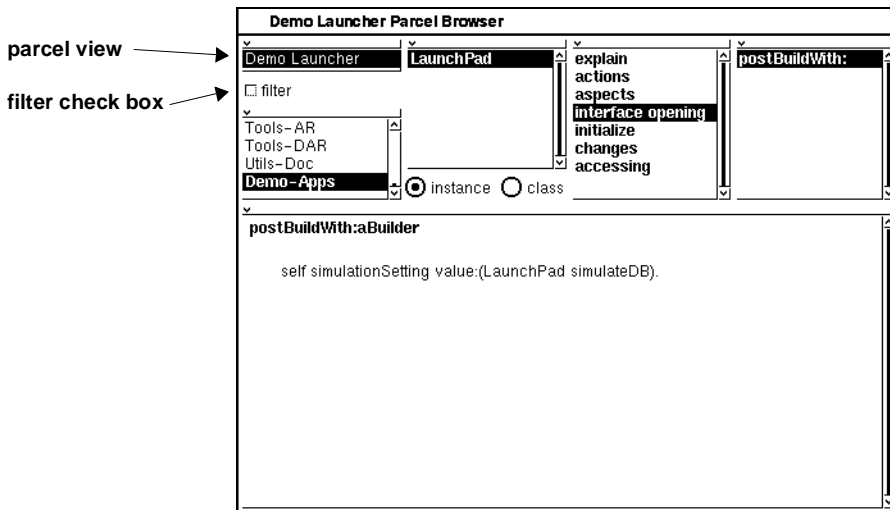


Figure 25-3 Parcel Browser

The Parcel Browser looks like the System Browser, but differs from it in three ways:

- n It has an additional *parcel view* which displays the name of the parcel and provides commands that affect the parcel as a whole.

-
- n Its display can be filtered to show only the definitions that are in the current parcel.
 - Items that are in the parcel are in bold.
 - Items that are in the system but not in the parcel are in regular typeface.
 - Classes that are in italics are not in the parcel but contain methods that are.
 - n The definitions displayed in the Parcel Browser cannot be edited.
(For a complete description of the Parcel Browser, see page 163.)
4. Add class and method definitions to the parcel by selecting items and choosing **add to parcel** from the <Operate> menu.
- When you:
- n Select a category, all of the classes, protocols, and methods in that category are added.
 - n Select a class, all of the protocols and methods in that class are added.
 - n Select a protocol, all of the methods in that protocol are added.
 - n Select a method, that method is added. Methods that are in a different parcel than their parent class are *extensions*. The names of classes that are extended appear in italic font.
- Note: If you write new methods for a class after the class has been added to a parcel, the new methods are not placed in the parcel automatically. You must explicitly add the class's new methods to the parcel. The same is true if you write a class in an already parcelled category or a new method in an already parcelled protocol.*
5. Remove any class and method definitions you don't want by selecting those items and choosing **remove from parcel** from the <Operate> menu.
6. (Optional) Add a parcel comment.
- a. Choose **comment** from the parcel view's <Operate> menu. The parcel's comment is shown in the code view.
 - b. Edit the comment. A comment may be any string.
 - c. Accept the new comment.
7. (Optional) Add a version string.
- a. Choose **version** from the parcel view's <Operate> menu. The parcel's version is shown in the code view.
 - b. Edit the version. A version may be any string.

- c. Accept the new version string.
8. You may view a summary of the classes that are in the parcel and the classes that are extended by methods in the parcel by choosing **summary** from the parcel view's <Operate> menu. The summary appears in the code view.

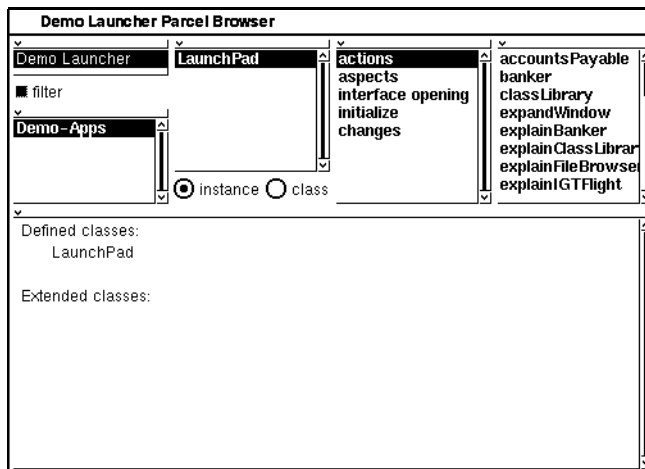


Figure 25-4 Summary of Parcel Contents

9. When you are done assigning classes and methods to the parcel, you must save it. Until a parcel is saved, it is considered *dirty*. A parcel is dirty when:
 - n Definitions have been added to or removed from the parcel since it was last saved.
 - n The parcel's version or comment has been changed since the parcel was last saved.
 - n Any of the definitions in the parcel have been modified since the parcel was last saved.

You can save a parcel from either the Parcel Browser or the Parcel List.

- n In the Parcel Browser, choose **save as...** from the parcel view's <Operate> menu.
- n In the Parcel List, select the parcel name and choose **Parcel?Save As....**

VisualWorks saves the parcel to a parcel file. Each parcel must be saved to its own parcel file.

10. Repeat this process until your entire application has been divided into parcels and saved. To help with this process, the Parcel Browser contains many of the navigation commands you know from the System Browser.

Loading Parcels

There are three ways to load parcels:

- n In a development image, you can load a parcel by choosing **Parcel?Load** in the Parcel List and specifying the name of the parcel's file.
- n In a deployment image, you can specify parcels on the command-line at startup.
- n In either image, you can load parcels programmatically from within your application.

Parcels can only add to the system. Loading a parcel will not remove items from the image or rename items. In contrast, filing in and filing out code is treated as a set of change instructions and can remove or rename items.

At Start Up

Deployment images that were created using Image Maker can load parcels during startup.

When a deployed image starts up, it looks in the working directory for a parcel configuration file with the filename ***image-name.cnf***, where *image-name* is the same as the image file's prefix. If such a file exists, the image loads the parcel files named in the file. Parcel file names should be listed one per line and are resolved with respect to the working directory.

You also can use command-line arguments to specify parcels to be loaded at start-up:

- n **-pcl *filename*** loads the specified parcel file.
- n **-cnf *filename*** loads all of the parcels listed in the specified configuration file.

From within an Application

Your application may load parcels as needed. For example, when your application's user starts a new tool or opens a new window within your application, your application may load the parcel that contains that tool or window.

The following line of code loads a parcel from a parcel file called **parcel-name.bin**:

```
Parcel loadParcelFrom: 'parcelname.bin'
```

You should decide what you want your application to do if the parcel is already loaded. You can have your application reuse the already-loaded parcel or reload the parcel from a file. You should consider how easily and regularly you need to replace your application's parcels with new, up-to-date parcels. You should also consider how quickly you want your application to respond.

Reuse

You probably want to reuse the already-loaded parcel if:

- n Your deployed application is well-tested and doesn't change often.
- n Your users routinely restart the application, at which point they would receive newly loaded parcels.
- n You want the quickest possible response from parcel-loading commands. This can be especially important when parcels are loaded across a distributed network.

If you program your application such that it will reuse already-loaded parcels, you may want to create an additional mechanism that will force reloading of parcels on demand. Such a mechanism could reload parcels based on user request, amount of time since last reload, or any other criteria you establish in your application.

Reload

You probably want to reload the parcel if:

- n You are in the process of testing and debugging your application and want to ensure that you are always using the most recent version of parcels.
- n Your deployed application requires frequent updating.
- n Your users routinely leave the application running for long periods of time without restarting and reloading.

Note that loading a parcel will not allow you to overwrite an existing class definition. Thus, you must use the **Parcel>>unload** method to unload parcels before loading their replacements. Unloading does not return the image to its

state before the parcel was loaded. In particular, it does not reconstruct methods that were overwritten by extension methods when the parcel was loaded. As with loading parcels, order may be important when unloading.

Behavior at Load Time

Loading classes from a parcel causes `postLoad` to be sent to those classes. The default behavior for `postLoad` is to call the class's `initialize` method if it has one. (The superclass's `initialize` method is not checked.) You can override that behavior by creating a `postLoad` method on the class side of your class.

The `postLoad` method can be used in combination with a `preSave` method to store named objects in a parcel and retrieve them as load time.

Load Order

It is up to you to ensure that parcels with dependencies are loaded in the correct order:

- n Superclasses must be loaded before their subclasses.
- n A class must be loaded before its methods.
- n Resources such as subcanvases, embedded data forms, bitmaps must be loaded before the classes that use them.

Load Errors

Loading of the basic contents of the parcel is uninterruptable as far as processes executing within the environment are concerned: basic loading either succeeds entirely or fails completely.

Loading a parcel will display an error message if you attempt to overwrite an existing class definition.

Note that parcel files do *not* contain source code. Without source code, you can execute the parcel's contents, but you may not be able to browse them. If you try to browse its contents, VisualWorks will notify you that there is a problem with the source and, if you have a decompiler, decompile the parcel's contents and display the results in the browser.

Filing Parcel Contents In and Out

The Parcel List and Parcel Browser each allow you to file out a parcel. Filing out a parcel writes all of the class and method definitions in the parcel to a file

as source code. Filing out a parcel does not file out the parcel itself or the association between the parcel and its classes and methods. Parcel contents that have been filed out can be filed in using the standard file-in mechanisms.

To file in a set of definitions and associate them with a parcel, first create and select the parcel in the Parcel List. Then choose **Utilities?File Into Parcel** to file the definitions into the parcel that you created. All of the definitions within it will be added to the system and also to the parcel.

Tips for Working with Parcels

Keeping Source Code and Parcels in Sync

Always keep in mind that parcels do not contain source code definitions. If you load the parcel into an image from which it was not created, the definitions are not present for editing, browsing, etc. If you have a decompiler, VisualWorks will display decompiled code, but variable names and comments will not be preserved.

Also keep in mind that source files do not preserve parcel information. You can assign sources to a parcel if you file them in using the Parcel List.

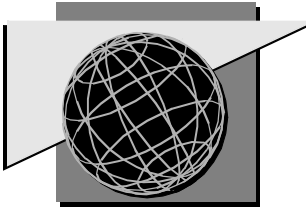
If you are using parcels during development, you may want to maintain both source code and parcel information and keep them in sync with each other. For example, you might:

- n Save parcels to both parcel files and source files.
- n Give parcel files and source files corresponding names, with different extensions.
- n Keep a record of which parcels are in which parcel files. The file names do not necessarily match the parcel names.

Warning: *If the definitions are not saved in the image or in a source file, you may not be able to reclaim them.*

Testing Parcel Files and Source Files for Matches

One way to test if your parcel file and source files match is to load the parcel file first and then load the source file. If any of the parcel's methods are still decompiled, then the files do not match. The source file should overwrite all of the parcel's methods.



Chapter 26

Creating a Deployment Image

A *deployment image* is an image that has been configured to run programs written in whole or in part in ParcPlace Smalltalk and whose purpose is to support the activities of an end user. In a deployment image, the development capabilities of the software have been stripped, blocked, or disabled and the remaining portions have been integrated with the VisualWorks user application into a cohesive whole.

The deployment image can be minimal with almost nothing in it. It serves as a vanilla environment. Your end users can then specify parcels to load at run time or your application can load parcels as they are needed.

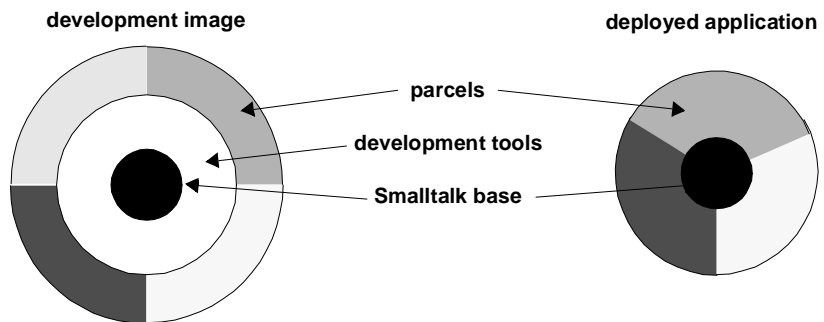


Figure 26-1 Development Image and Deployed Application

Setting Up a Deployment Image

Before deploying your application, you should consider a few aspects of image-making and the resulting deployed image.

Handling Errors

Your application is expected to catch most error signals and handle them in a way that is consistent with your user interface. The **DeploymentNotifier** is provided for errors that are not otherwise handled—and as a placeholder for your own notifier.

For more information about trapping error signals, see the chapter “Processes and exception handling.”

The Transcript

The **Transcript** as an object is preserved in a deployment image. The **Transcript** as part of the **VisualWorks** main window, however, is *not* in the deployment image. Messages sent to **Transcript** process without errors but do not display themselves unless you define something to show the state of the **Transcript**.

Undeclared Variables

The system maintains a dictionary of undeclared variables, which you can access via the global name **Undeclared**. An entry is appended to **Undeclared** when:

- n A reference to a nonexistent variable is compiled during file-in (or inter-actively, if you override the compiler’s warning).
- n A variable is removed while references still exist.
- n A class is removed (regardless of whether outside references to it exist). This assures that any outside references that may exist will be properly reconnected if the class is recreated.

Ideally, the **Undeclared** dictionary is empty in a deployed image. Entries are removed automatically when the missing variable is declared. A specialized Inspector has been created to help you manage the **Undeclared** dictionary manually. To open an Inspector on **Undeclared**:

1. Open a Workspace and type **Undeclared**.
2. Select **Undeclared** and choose **inspect** from the <Operate> menu.

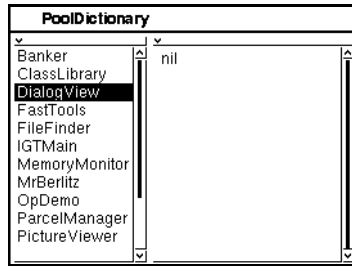


Figure 26-2 An Inspector on the Undeclared

The Undeclared Inspector provides a **references** command for finding methods that refer to the selected variable. Note that hidden references will not be reported—if you suspect such references may exist, you will have to track them down using other means.

When you are satisfied that no references to the selected variable exist, use the Inspector's **remove** command to delete the entry.

Creating a Deployment Image

To create a deployment image, you use a tool called Image Maker. Image Maker enables you to remove development tools and other unwanted classes from an image. The resulting image is more appropriate for distribution to end users of your application. It also occupies significantly less disk space. Just how much depends on which functions and classes you mark for removal.

To use Image Maker:

1. Set up your application as you want it to be delivered:
 - n To save your entire application as part of the deployment image, make sure it is in the current development image and that the appropriate windows are open.
 - n To save only your application's initial window as part of the deployment image, make sure it is open and in the current development image. Make sure application code that will be delivered in parcels (or some other form) is removed from the image. You can remove classes manually or by specifying them as additional classes for Image Maker to remove in step 4, below.

- n To save only a bare deployment image that will load your entire application from parcels at (or after) startup, remove all of your application code from the image.
2. File in **imagemkr.st** from the **utils** directory.
3. In a Workspace, execute the following:

ImageMaker open

Image Maker starts and displays a window that allows you to choose what you want to remove from your development image before saving it as a deployment image.

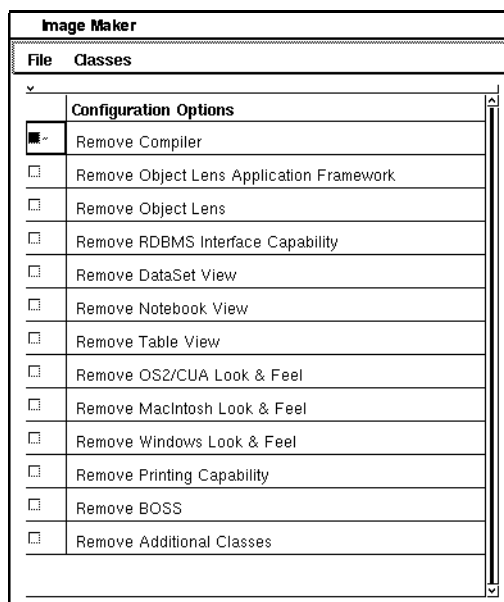


Figure 26-3 Image Maker

4. Choose what you want to remove from the image. For more information about removal options, see “Optional Removal of Other Facilities” on page 366.
5. Run Image Maker by choosing **File?Make Deployment Image**. Image Maker asks you for the name of the release directory. The release directory is the directory in which VisualWorks was installed.
6. Enter the name of the release directory and click **OK**.

Image Maker then:

- a. Removes the specified classes from the image.
 - b. Closes all of the VisualWorks development tools.
 - c. Leaves your application in the image and, if it was open, leaves it open.
 - d. Asks you for the name of the file to which you want the new image saved.
7. Enter the name for the new image and click **OK**.
 8. Image Maker displays a dialog box providing instructions for the next step. Click **OK** to dismiss the dialog box.

Image Maker then:

- a. Saves the image.
 - b. Closes your application, if it was running.
 - c. Shuts down the VisualWorks object engine and image.
9. Start the new image. To do so, use the same object engine you used previously and indicate the name of the new image.
The new image appears exactly as it did before Image Maker shut down in step 8. If your application was running, it is restarted.
 10. Image Maker displays a final dialog box indicating that it will now save and quit once again. Click **OK**.

Image Maker again:

- a. Saves the image.
- b. Closes your application, if it was running.
- c. Shuts down the VisualWorks object engine and image.

Your new image is now ready.

Operations Performed by Image Maker

Image Maker uses your current development image as the base from which to create the deployment image. It does not change the file in which your development image resides. The new deployment image is saved in a separate file.

***Note:** Image Maker has been optimized to meet the needs of the current VisualWorks release. Image Maker code for one release may not work correctly for another release.*

Removal of Development Facilities

When Image Maker creates a deployment image, it always removes:

- n The VisualWorks main window.
- n All of the VisualWorks development tools, including the browsers, inspectors, debugger, change list, project tools, painting tools, online documentation tools, and database application-generation tools.
- n All of the development tools for add-on products and ObjectKits, including the Advanced, Business Graphics, and DLL and C Connect tools.
- n System organization (categories and protocols).
- n Pool dictionaries that are no longer needed. The pool dictionary `OpcodesPool` is treated specially. It is emptied and removed from Smalltalk; its keys are hidden, but it keeps its associations around in case they are used.
- n All source files.

Optional Removal of Other Facilities

Image Maker provides options for removing a variety of facilities from the deployment image. The Image Maker window presents the options as check boxes. Options are expressed in terms of system functionality, not in terms of class names or categories.

To find out exactly which classes will be removed when a particular option is chosen, look at the corresponding method in the `options` protocol on the class side of `ImageMaker` and the associated remove script in the `utils/removals` subdirectory of the release directory. When a remove script indicates to remove a class, that class's subclasses are also removed.

Display Capability

If you have filed `utils/headless.st` into your image, Image Maker presents an additional option for creating an image that does not have access to a display system (console or window system). Such an image is called *headless*. By default, the **Generate Headless Image** option is chosen, causing Image Maker to create a deployment image that is in the headless state. (The intermediate image during processing is *headful* so it can present image-making status and instructions; the final saved image is headless.)

If **Generate Headless Image** is not chosen, Image Maker creates a deployment image that operates, by default, in a headful state. Note that

Image Maker does *not* remove the headless capability or supporting code; it simply turns the display features on or off based on your selection.

Compiler Classes

When you select the Image Maker option labeled **Remove Compiler**, Image Maker removes all compiler-support classes.

Removing compiler classes produces what is known as a *closed system*, because the system is closed to compilation activities. When compiler support is left in the image, it is called an *open system*.

A closed system provides a greater degree of protection against user tampering, and further reduces the space required by the image file. While no new code can be evaluated in such an image, all existing tools and applications that do not rely on this capability will continue to function normally.

An open system is appropriate when your application needs to evaluate Smalltalk code. For example, a development environment for Smalltalk programmers would need to be open.

Note: If you do not choose to remove the compiler classes, Image Maker displays a notifier reminding you to verify that your license agreement allows you to deploy systems that include the compiler.

Database Support

If your application does not access a database, you may choose for Image Maker to remove RDBMS Interface (EXDI) capability, the ObjectLens, or the ObjectLens application framework (including LensMainApplication and LensDataManager).

Note that there are dependencies between these three facilities:

- n The RDBMS Interface capability is used by the ObjectLens and the ObjectLens application framework. You cannot remove it without removing them.
- n The ObjectLens is used by the ObjectLens application framework. You cannot remove the ObjectLens without removing its application framework.

Unused User-Interface Widgets

Certain widgets have a great deal of supporting code. If you do not use those widgets in your application, you should choose for Image Maker to remove their code. In particular, Image Maker provides the ability to remove the:

- n Dataset
- n Notebook
- n Table

***Note:** Use caution when removing dataset widgets from database applications. Data forms that were generated by the Canvas Composer and that use a tabular format generally require the dataset view.*

Unused User-Interface Looks

Image Maker provides options to remove user-interface look policies and supporting code for the following platforms:

- n OS2/CUA
- n Windows
- n Macintosh

Image Maker does not provide options for removing the default look or Motif user-interface look policies. DefaultLook is required as a last resort of AutoSelect when no other look policy is appropriate. Default look policy also provides constants that are used by all the other look policies. In turn, the Default Look widgets are built on top of and require the Motif look.

Note that Image Maker will not remove a user-interface look that is currently in use. You may have to change your **UI Look** settings before creating a deployment image.

BOSS

Image Maker provides an option for removing the Binary Object Streaming Service (BOSS). If your application does not depend on objects delivered in BOSS files, choose **Remove BOSS**.

Printing Capability

If your application does not need to write PostScript files or send information to a printer, you can remove printing support by choosing **Remove Printing Capability**.

Additional Classes

In addition to removing predefined sets of classes, Image Maker provides a way for you to identify other classes you would like to have removed. To remove additional classes:

1. In the Image Maker window, choose **Remove Additional Classes**.
2. Choose **Classes?Set Additional Removals** from Image Maker's menu bar.

Image Maker shows all of the classes in the system, including those already being removed. From this list, choose additional classes that you want removed. You may want to consider classes such as:

- n Classes that support your own development tools or extensions to the VisualWorks development environment
- n Operating-system support classes that your application does not require

The additional classes that you specify are stored in Image Maker's `additionalClasses` instance variable.

Preservation of Certain Facilities

Unless you specify them as additional classes to be removed, Image Maker does not remove:

- n Frameworks for add-on products or ObjectKits.
- n The headless capability. If you have filed in `utils/headless.st`, Image Maker does not remove that code from the deployment image. It does, however, allow you the option of creating a deployment image that operates in either headful (with a graphical user interface or console) or headless mode. See "Display Capability" on page 366.
- n Any development tools you have created.
- n Your application classes.

Optimization of Memory Usage

Image Maker automatically performs a series of operations to optimize memory usage. In particular, it:

- n Performs a **perm save as**, which moves all of the objects that were formerly in OldSpace into PermSpace, including your application code.

Moving your application code to PermSpace enable the system's memory reclamation facilities to run faster.

- n Performs a global garbage collection, which removes transient objects from PermSpace.
- n Asks you to load the new image back into memory so that it can perform one last snapshot. The image resulting from this snapshot obtains the benefits of the global garbage collection and is significantly smaller. Performing a snapshot also makes subsequent loads on the same platform even faster because the global garbage collector compacts the objects in PermSpace, which forces the load code to relocate these objects at start-up time. By performing one extra snapshot, these objects will not need to be relocated on subsequent loads when it is possible for the object engine to load them into their former locations.

For more information about kinds of memory and optimizing memory usage, see Chapter 23, "Memory Management."

Other Changes

In addition to removing unneeded functionality and optimizing memory usage, Image Maker performs a variety of operations to set up the image for deployment. Image Maker:

- n Replaces the `NotifierView` class with `DeploymentNotifier`. Whenever an error occurs in the deployed image, `DeploymentNotifier` writes the entire stack trace to a file (by default, `visual.err`).
- n Writes a complete account of all patches loaded before and during image making, including removal scripts, the list of additional classes removed, and all parcel names. The account is written to the file `image-name.rpt`, where `imagename` is the name of the deployment image.
- n Sets `noWindowBlock` on `ControlManager` to be a block that will shut down the image whenever the last window is closed by the end user.
- n Creates an instance of `AutoLoader` and sets it to be a dependent of `ObjectMemory`. `AutoLoader` is responsible for loading parcels during startup (see "Starting Up a Deployed Image," below). `AutoLoader` is set before `HeadlessImage` in the dependents list so that, if present, parcels will be loaded before source files.

Saving the State of Image Maker

You can save the state of Image Maker, including all of the options chosen and additional classes specified. To do so, choose **File?File Out Choices** from the Image Maker menu bar. Image Maker prompts you for the name of the file in which to save its state.

To load pre-saved state back into Image Maker, choose **File?File In Choices** from Image Maker's menu bar.

Starting Up a Deployed Image

You start a deployment image the same way you start a development image, specifying the object engine and name of the deployment image.

When a deployed image starts up, it looks in the working directory for a parcel configuration file with the filename *imagename.cnf*, where *imagename* is the same as the image file's prefix. If such a file exists, the image loads the parcel files named in the file. Parcel file names should be listed one per line and are resolved with respect to the working directory.

Additionally, you can use command-line arguments to specify parcels to be loaded at start-up:

- n **-pcl *filename*** loads the specified parcel file.
- n **-cnf *filename*** loads all of the parcels listed in the specified configuration file.

If the image is headless, it then looks for a start-up file.

Debugging a Deployed Image

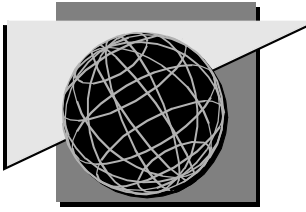
A deployed image does not contain development tools. There are, however, a few mechanisms to help you obtain information about errors in a deployed image.

When it creates an image, Image Maker writes a complete account of all patches loaded before and during image making, including removal scripts, the list of additional classes removed, and all parcel names. The account is written to the file *imagename.rpt*, where *imagename* is the name of the deployment image. That report can be used to recreate or duplicate the deployed image.

Image Maker replaces the `NotifierView` class with `DeploymentNotifier`. Whenever an error occurs in the deployed image, `DeploymentNotifier` displays a dialog in which you can click **Write Report** and specify a filename for the report. The report contains a description of the image (release number, patches, and so on) and the complete stack trace. Within your application, you can send the message `logging: true` to the `DeploymentNotifier` class to arrange for a stack trace to be appended to the file `visual.err` whenever an unhandled exception occurs.

Exiting a Deployed Image

To supplement any shut-down facilities provided by your application, Image Maker establishes `noWindowBlock` on `ControlManager`. When the last window is closed, the deployment image shuts down.



Chapter 27

Creating Applications without Graphical User Interfaces

Applications that rely on direct user interaction typically provide graphical user interfaces for collecting input and displaying output. You can also write batch or server applications that, by their nature, do not rely on direct user interaction, and may run on computers that have no console or windowing system. Such applications execute in *headless* VisualWorks images—that is, images that run with the display system deactivated (in *headless mode*).

This chapter describes the general steps for creating a headless image and executing an application in it.

Key Concepts

The headlessness of an image is controlled by the sole instance of the class **HeadlessImage**. This instance (**HeadlessImage default**) enables you to create new images by saving them either in headless mode (with the display system deactivated) or in “headful” mode (with an activated display system). You typically develop your application in a headful image, test it in a headless image, and then debug it in a headful image that is created from the headless image. The **HeadlessImage** instance records the image’s mode and can be queried for it.

The basic way to provide input to a headless image is through a startup file. A startup file is a file that contains Smalltalk expressions in file-in format. When a headless image is started, it reads the file and evaluates the expressions. You typically use a startup file to start your application in the headless image. Applications can also accept input through sockets, file I/O, tty interaction, and so on.

By default, output that would normally be displayed in the System Transcript is saved to disk in a transcript file.

Setting Up a Headless Image

To prepare to execute an application in headless mode, you start with a standard VisualWorks image, configure it, and then create a headless image from it, as described in the following steps:

1. In a standard VisualWorks image, file in **headless.st**, located in the **utils** subdirectory of the release directory. This introduces the **HeadlessImage** class plus several other classes in the category **Headless-Support**.
2. Write your application so that it can run in headless mode (see “Tips for Programming a Headless Application” on page 379). Note that the application can send messages to the **HeadlessImage** instance (for example, to test whether it is running in a headless or headful image).
3. Decide how you will want to start your application and prepare accordingly (see “Techniques for Starting a Headless Application” on page 379). You may want to file out your application into a startup file, or make certain modifications to the system. A basic technique is to leave the application in the image and create a startup file that contains a line such as **MyApplication open!**.
4. Decide whether you want the headless image to file in a startup file, and if so, whether to use the default startup filename (**hlstrc.st**).
 - n If you do not want to use a startup file, evaluate the following expression:

```
HeadlessImage default startupFilename: nil
```

- n If you want to use a startup file with a nondefault name (for example, **myStartUp.st**), evaluate an expression such as the following:

```
HeadlessImage default startupFilename: 'myStartUp.st'
```

5. Decide whether you want the headless image to append transcript messages to the file **hlst.tr**:
 - n If you do not want to use any transcript file, evaluate the following expression:

```
HeadlessImage default transcriptFilename: nil
```

- n If you want to use a transcript file with a nondefault name (for example, **myTranscript.tr**), evaluate an expression such as the following:

```
HeadlessImage default startupFilename: 'myTranscript.tr'
```

- 6. Create a headless image by evaluating an expression such as the following:

```
HeadlessImage default saveHeadless: 'headlessImageName'
```

This creates a new image named **headlessImageName.im** in which **HeadlessImage**'s state is set to headless. Creating a headless image has no effect on the current image.

Running an Application in Headless Mode

To run an application in headless mode:

- %o Start the headless image as you would normally start a standard VisualWorks image.

When an Image Starts

When a standard VisualWorks image starts, **ObjectMemory** installs objects that are fundamental to the display, thereby hooking the image up to the host windowing system. **ObjectMemory** also broadcasts **#returnFromSnapshot** to its various dependents, which respond with their own startup actions.

When a headless image starts, the **ObjectMemory** refrains from hooking it up to the underlying windowing system and does not install the objects that are associated with the display. Consequently, those objects are not available for referencing later.

The **HeadlessImage** is registered as a dependent of **ObjectMemory**. Upon receiving **#returnFromSnapshot**, the **HeadlessImage** instance checks its state to verify that the image is headless and replaces the normal **Transcript** (a display-oriented object of class **TextCollector**) with a file-based surrogate of class **FileTextCollector** or **NullTextCollector** if a nil transcript filename is provided. The normal **Transcript** is retained (so it can be reinstalled in the image if it is saved as headful), but not accessible while headless.

Finally, the `HeadlessImage` instance checks whether a startup file has been specified; if so, the start-up file is filed in and the Smalltalk expressions in it are evaluated. Typically, these expressions start up the headless application.

If an Application Attempts to Access a Display

If an application that is running in a headless image attempts to access the non-existent display, the attempt is trapped, and an exception, `HeadlessImage headlessErrorSignal`, is raised. If the exception is not caught, the offending process is suspended and saved by the `HeadlessImage` instance for debugging.

More specifically, the message `#checkHeadless` is sent to the `HeadlessImage` instance by methods that attempt to create instances of `DisplaySurface` or its subclasses (for example, `ScheduledWindow`, `PopUpMenu`, `NotifierView`, and so on). In a standard, headful image, `#checkHeadless` returns without any side effect. In a headless image, the `HeadlessImage` instance responds to `#checkHeadless` by sending itself `#cannotSend`. This, in turn, causes the `HeadlessImage` instance to raise the exception, suspend and save the process, write a context trace to the transcript file, save the image as a headful image, and then terminate the image.

Debugging a Suspended Process

When a process has been suspended as the result of an attempt to display something, you can use the saved, headful image to debug a suspended process:

1. Start the headful image that was saved by the headless image before it terminated. By default, the image is called `h1st_dbg.im`.
2. Inspect the suspended processes by evaluating the following expression:

```
HeadlessImage default suspendedProcesses inspect
```

3. In the Inspector, select a process and then invoke **debug** from the `<Operate>` menu. VisualWorks brings up a debugger on the selected process.

Creating a Headful Copy of a Headless Image

In general, you can create a headful copy of a headless image by including an expression such as the following in your application code or by providing the expression in file-in format in a startup file:

```
HeadlessImage default saveHeadfull: 'name'
```

In the resulting image, the `HeadlessImage` instance's state is set to `headful`, which enables the display at startup. Saving a headful image from a headless image is useful if you need to debug a failure (see “Debugging a Suspended Process” on page 378).

When a headful image is created from a headless one, the normal Transcript is restored.

Tips for Programming a Headless Application

Your headless application may do whatever you wish, as long as it does not access the display. When programming your application, you need to consider how to start it, how users can communicate with it, how to terminate it, and how to prevent it from accessing the display.

Techniques for Starting a Headless Application

A simple technique for starting an application is to write it in a start-up file (in file-in format). The start-up file is read and evaluated (filed in) when the image starts. By writing your application in the start-up file, you have the flexibility to make changes and re-execute relatively quickly. That is, you can change your application without having to start up and save a headful image; you can simply change the startup file and restart the headless image.

Alternatively, you can write your application in the headful image from which you will create the headless image. When your application resides in the headless image, you have three options for starting it:

- n Use the start-up file—for example, `MyApplication open!`.
- n Modify `HeadlessImage>returnFromSnapshot` to fork off a process with your application—for example, `(MyApplication open) fork`.
- n Register your application as a dependent of `ObjectMemory` and wait for `#returnFromSnapshot` to be broadcast. Look at `HeadlessImage` for an example, particularly `#initialize` and `#update`: If you do this, make

sure that `HeadlessImage` appears before your application in the dependents collection.

Techniques for Communicating with a Headless Application

Your application must provide some means other than a window system for users to interact with a headless image. This can be addressed with sockets, file I/O, or some other manner.

Terminating a Headless Application

Your application should make provisions for shutting down gracefully, under both normal and exceptional circumstances. The last message send should be `ObjectMemory quit`, which causes the image to terminate. Failure to do so will leave the image running, but with nothing to do. Your only recourse then is to terminate the image from the operating system (for example, by using `kill` in UNIX).

Preventing Access to the Display

If a headless application attempts to access the non-existent display, the attempt is trapped, and an exception, `HeadlessImage headlessErrorSignal`, is raised.

Your application can ignore this exception and rely on default behavior. Alternatively your application can handle the exception as appropriate. Note that you can still execute the default behavior if you `#proceed` rather than `#return` in the exception handler. For example:

```
HeadlessImage headlessErrorSignal
  handle: [ :exception |
    . . . special stuff to do when a headless violation occurs . . .
    exception proceed]
  do: [ . . . whatever your application does normally . . . ]
```

If your application is to have different behavior depending on the kind of image it is running in, you can use the following expression to determine whether it is currently running in a headless image:

```
HeadlessImage default isHeadless
```

Similarly, you may send `#checkHeadless` from your application code when you have code that should be executed only in a headful image:

`HeadlessImage default checkHeadless`

Note that you may modify the `HeadlessImage>>cannotSend` method to tailor it for your specific needs.

Delivering a Headless Application

You deliver a headless application much the same way you delivery any other application. You can separate your application into parcels for easy distribution and you can use Image Maker to create a minimal deployment image that loads your application's parcels.

When you use Image Maker in a headless image, Image Maker includes a special **Generate Headless Image** option (shown in Figure 27-1). When the option is:

- n Chosen, Image Maker creates a deployment image that is in headless mode.
- n Not chosen, Image Maker creates a deployment image that is in headful mode.

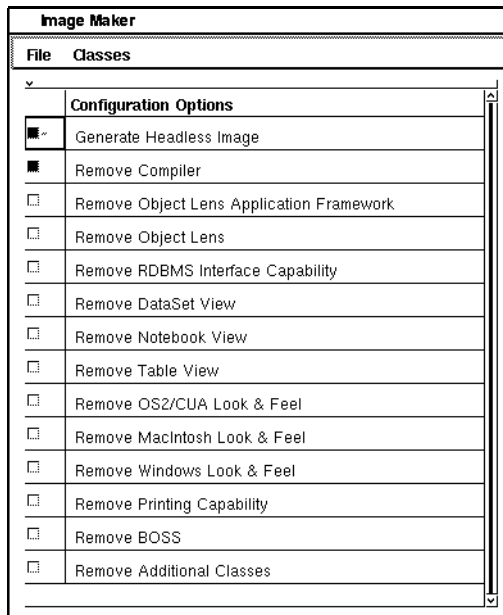
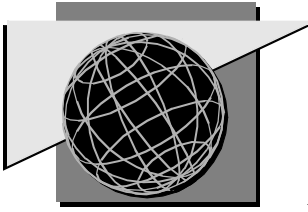


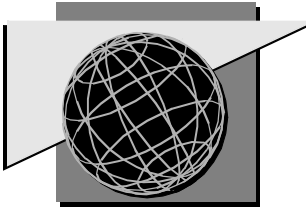
Figure 27-1 Image Maker with Headless Option

Note that in either case, Image Maker does *not* remove the headless capability. The classes added by `headless.st` are included in the deployment image unless you name them as additional classes to be removed.



Part V

Appendixes



Appendix A

Protocol Reference

The following table is provided as an aid in choosing a protocol for a common method type. It is by no means a complete listing of protocols in the system, nor should it have the effect of limiting your invention of new protocols for uncommon situations.

Common Class Protocols

Common protocols of methods addressed to classes are listed in the first section, followed by instance protocols. Within each section, protocols are listed in descending order of their frequency within the class library.

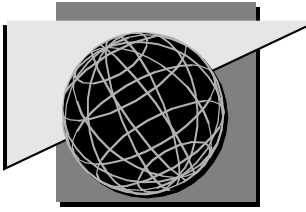
Table A-1 Class Protocols

Use this Protocol	for Methods that...
instance creation	Return an instance of the class
class initialization	Set the values of class variables
constants access	Return the value of a class variable
private	Perform an operation that is only likely to be needed by another method within the class or a subclass
examples	Provide sample code showing how to use an instance of the class
documentation	Provide comments about some aspect of the class

Common Instance Protocols

Table A-2 Instance Protocols

Use this Protocol	for Methods that...
accessing	Set or retrieve the value of an instance variable
private	Perform an operation that is only likely to be needed by another method within the class
initialize-release	Set the initial values of the instance's variables, or release a dependency connection
testing	Test whether the object has a particular attribute, such as <code>aNumber isZero</code>
printing	Provide a printable representation
copying	Make a copy
converting	Transform to another class of object, as when a <code>Dictionary</code> is converted to a <code>SortedCollection</code>
comparing	Compare the receiver with an argument object
displaying	Render an object (text or other graphics) on a displayable medium
enumerating	Perform an element-by-element operation on a collection
updating	For a view, receive an update notice from a model
adding	Add an element to a collection
menu messages	Perform a menu operation
controller access	For a view, get or set the controller
error handling	Raise an exception
model access	Get or set the model on which a view or controller is dependent



Appendix B

Syntax Descriptions

In the sections that follow, the syntax of the Smalltalk language is formally defined with the aid of Backus-Naur form. The following characters have special meanings unless they are enclosed in quotation marks.

Table B-1 Special Characters

Character	Description
=	expands to
' '	terminal (single quotes surround an atomic literal)
" "	comment (double quotes surround a comment)
	or
+	one or more
*	zero or more
[]	zero or one
...	through
()	grouping
< >	keyboard key

Lexical Primitives

The lexical syntax is formally ambiguous, in that, for example, the string `abc:` can be parsed either as an identifier followed by a non-quote-character, or as a keyword. We resolve this ambiguity in all cases in favor of the longest token that can be formed starting at a given point in the source text. Thus `abc:` is always considered to be a keyword, if the 'a' is the beginning of the token.

Character Classes

The definition of token is not used anywhere else in the syntax; it is supplied only for exposition.

```
token = number | identifier | special-character | keyword |  
       block-argument | assignment-operator |  
       binary-selector | character-constant | string
```

```
digit = '0' | ... | '9'
```

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

```
binary-character = '+' | '/' | '\' | '*' | '~' | '<' | '>' |  
                 '=' | '@' | '%' | '|' | '&' | '?' | '!' | ','
```

```
whitespace-character = <tab> | <space> | <newline>
```

```
non-quote-character =  
  digit | letter | binary-character |  
  whitespace-character |  
  '[' | ']' | '{' | '}' | '(' | ')' | '_' | '^' | ';' |  
  '$' | '#' | ':' | '.' | '-' | '`'
```

Numbers

digits = digit+

big-digits = (digit | letter)+ “as appropriate for radix”

number = digits
('r' ['-'] big-digits optional-big-fraction-and-exponent |
optional-fraction-and-exponent)

optional-fraction-and-exponent =
['.' digits] [('e' | 'E' | 'd' | 'D') ['-'] digits]

optional-big-fraction-and-exponent =
['.' big-digits] [('e' | 'E' | 'd' | 'D') ['-'] digits]

Other Lexical Constructs

identifier = letter (letter | digit)*

block-argument = ':' identifier

assignment-operator = '_' | ':' '='

keyword = identifier ':'

binary-selector = ('-' | binary-character) [binary-character]

unary-selector = identifier

character-constant = '\$' (non-quote-character | ''' | '''')

symbol = identifier | binary-selector | keyword+

string = ''' (non-quote-character | ''' | '''')* '''

comment = ''' non-quote-character* '''

separators = (whitespace-character | comment)+

Note: Numbers in radices 14 or 15 and higher can't specify an exponent, because the 'd' or 'e' respectively will be interpreted as a big-digit.

Atomic Terms

```
literal = ['-'] number | named-literal | symbol-literal |
         character-literal | string | array-literal |
         byte-array-literal
```

```
named-literal = 'nil' | 'true' | 'false'
```

```
symbol-literal = '#' (symbol | string)
```

```
array-literal = '#' array-literal-body
```

```
array-literal-body = '(' (literal | symbol | array-literal-body |
                        byte-array-literal-body)* ')'
```

```
byte-array-literal = '#' byte-array-literal-body
```

```
byte-array-literal-body =
  '[' number* "integer between 0 and 255" ']'
```

```
variable-name = identifier
               "other than named-literal, pseudo-variable-name or
               'super'"
```

We originally intended that the definition of array-literal be the following:

```
array-literal = '#' '(' literal* ')'
```

This would have simplified the syntax, eliminating the need for array-literal-body and byte-array-literal-body as separate constructs. However, this definition is not backward-compatible with previous versions of the Smalltalk-80 language: Specifically, it requires symbols and arrays appearing within an array literal to be quoted with #. Because of this, we adopted the more complex definition.

Expressions and Statements

primary = variable-name | pseudo-variable-name | literal |
block-constructor | '(' expression ')'

pseudo-variable-name = 'self' | 'thisContext'

unary-message = unary-selector

binary-message = binary-selector primary unary-message*

keyword-message =
(keyword primary unary-message* binary-message*)+

cascaded-messages =
(';' (unary-message | binary-message | keyword-message))*

messages =
unary-message+ binary-message* [keyword-message] |
binary-message+ [keyword-message] |
keyword-message

rest-of-expression = [messages cascaded-messages]

expression =
variable-name
(assignment-operator expression | rest-of-expression) |
keyword '=' expression "see below" |

```
primary rest-of-expression |
'super' messages cascaded-messages
```

```
expression-list = expression ('.' expression)* ['.']
```

```
temporaries = '|' temporary-list '|' '|'
```

```
temporary-list = declared-variable-name*
```

```
declared-variable-name = variable-name
```

```
statements = ['^' expression ['.'] | expression ['.'] statements]
```

```
block-constructor = '[' [block-declarations] statements ']'
```

```
block-declarations = temporaries |
  block-argument+
  ('|' [temporaries] | '|' temporary-list '|' '|')
```

In order to keep lexical analysis and parsing separate, but still allow constructs like `x:=3` (without a space, making it look like a keyword, `x:`), we have had to introduce the alternative

```
keyword '=' expression
```

for assignment. This should really be read as though it were

```
variable-name ':=' assignment
```

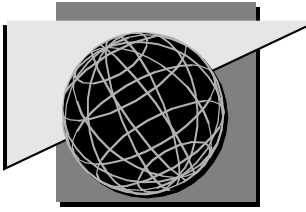
Methods

method = message-pattern [primitive] [temporaries] statements

message-pattern =
unary-selector |
binary-selector declared-variable-name |
(keyword declared-variable-name)+

primitive = '<' 'primitive:' primitive-number '>'

primitive-number = number “an integer between 0 and 65535”



Appendix C

Special Characters

A variety of special characters, such as the yen sign (¥), can be typed into VisualWorks text views by using a special key sequence. A prefix known as the *compose key* is the first element in the key sequence, followed by two characters that define the desired special character. On some keyboards, a single key has been defined to send the required sequence, such as the dollar sign on American keyboards. If the font in use does not contain a character, it is displayed as a black square.

For example, <Control>-q = Y is the sequence for composing the yen sign.

The default compose key is <Control>-q. To change the default key, execute the expression `CharacterComposer setComposeKey`. The new compose key will affect newly created views but not existing views.

Composed Characters

The following table lists the special characters in the left column. The two characters that make up the body of the compose sequence are shown in the second column. The hexadecimal equivalents of these two columns are displayed in the right-hand columns. A description is shown in the middle column.

Table C-1 Special Characters

Special char	Compose chars	Description	Special hex code	Compose hex codes
#	+ +	number sign	0023	2B 2B
\$	S	dollar sign	0024	7C 53
@	A A	at	0040	41 41
[((left bracket	005B	28 28

Table C-1 *Special Characters*

Special char	Compose chars	Description	Special hex code	Compose hex codes
\	//	backslash	005C	2F 2F
]))	right bracket	005D	29 29
{	(-	left brace	007B	28 2D
	/ ^	vertical bar	007C	2F 5E
}) -	right brace	007D	29 2D
~	^ ^	tilde	007E	5E 5E
¡	! !	inverted exclamation	00A1	21 21
¢	c	cent sign	00A2	7C 63
£	= L	pound sign	00A3	3D 4C
¤	x o	currency	00A4	78 6F
¥	= Y	yen sign	00A5	3D 59
§	! s	section	00A7	21 73
©	O C	copyright	00A9	4F 43
ª	_ a	ordfeminine	00AA	5F 61
<<	< <	<<	00AB	3C 3C
—	- -	horizontal bar	00AD	2D 2D
®	O R	registered	00AE	4F 52
°	^ 0	degree sign	00B0	5E 30
±	+ -	plus or minus	00B1	2B 2D
²	^ 2	superscript 2	00B2	5E 32
³	^ 3	superscript 3	00B3	5E 33
µ	/ u	micro, mu	00B5	2F 75
¶	! p	paragraph sign	00B6	21 70
.	. ^	middle dot	00B7	2E 5E

Table C-1 *Special Characters*

Special char	Compose chars	Description	Special hex code	Compose hex codes
¹	^ 1	superscript 1	00B9	5E 31
º	_ o	ordmasculine	00BA	5F 6F
>>	> >	>>	00BB	3E 3E
¼	1 4	one fourth	00BC	31 34
½	1 2	one half	00BD	31 32
¾	3 4	three fourths	00BE	33 34
¿	? ?	inverted ?	00BF	3F 3F
Æ	A E	AE diphthong	00C6	41 45
Ð	+ D	capital eth	00D0	2B 44
×	x x	cross	00D7	78 78
Ø	/ O	O slash	00D8	2F 4F
Þ	O	capital thorn	00DE	7C 4F
ß	s s	German double-s	00DF	73 73
æ	a e	ae diphthong	00E6	61 65
ð	+ d	small eth	00F0	2B 64
÷	- :	divide	00F7	2D 3A
ø	/ o	o slash	00F8	2F 6F
þ	o	small thorn	00FE	7C 6F
Ð	- D	D with stroke	0110	2D 44
ḏ	- d	d with stroke	0111	2D 64
Ḣ	- H	H with stroke	0126	2D 48
ḥ	- h	h with stroke	0127	2D 68
ı	. i	dotless i	0131	2E 69
IJ	I J	IJ ligature	0132	49 4A

Table C-1 *Special Characters*

Special char	Compose chars	Description	Special hex code	Compose hex codes
ij	i j	ij ligature	0133	69 6A
κ	k k	kra	0138	6B 6B
Ł	. L	L with dot	013F	2E 4C
ł	. l	l with dot	0140	2E 6C
Ł	- L	L with stroke	0141	2D 4C
ł	- l	l with stroke	0142	2D 6C
’n	n ’	n apostrophe	0149	6E 27
Ŋ	N)	capital eng	014A	4E 29
ŋ	n)	small eng	014B	6E 29
Œ	O E	OE diphthong	0152	4F 45
œ	o e	oe diphthong	0153	6F 65
Ƨ	- T	T with stroke	0166	2D 54
Ƨ	- t	t with stroke	0167	2D 74
‘	‘ 1	single quote left	2018	60 31
’	’ 1	single quote right	2019	27 31
“	‘ ‘	double quote left	201C	60 60
”	’ ’	double quote right	201D	27 27
™	T M	trademark	2122	54 4D
Ω	o m	ohm	2126	6F 6D
1/8	1 8	one eighth	215B	31 38
3/8	3 8	three eighths	215C	33 38
5/8	5 8	five eighths	215D	35 38
7/8	7 8	seven eighths	215E	37 38
←	- <	arrow left	2190	2D 3C

Table C-1 *Special Characters*

Special char	Compose chars	Description	Special hex code	Compose hex codes
↑	^	arrow up	2191	7C 5E
→	- >	arrow right	2192	2D 3E
↓	v	arrow down	2193	7C 76
♪	n o	musical note	266A	6E 6F
·j	. j	dotless j	FC10	2E 6A

Diacritical Marks

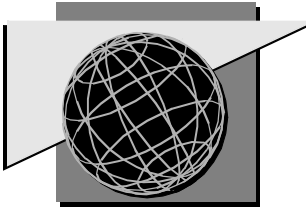
A diacritical mark, such as a circumflex (^), is combined with a character in a similar fashion. The compose key (<Control>-q by default) comes first, then a character representing the diacritical mark (taken from the table below) and finally the base character. For example, to get ñ, you would type <Control>-q, followed by a tilde (~) and the letter 'n'.

Table C-2 *Diacritical Marks*

Diacritical mark	Compose char	Description	Diacritical hex code	Compose hex code
`	`	grave	0300	60
´	´	acute	0301	27
^	^	circumflex	0302	5E
~	~	tilde	0303	7E
-	-	macron	0304	2D
˘	u	breve	0306	75
·	.	dot above	0307	2E
¨	"	dieresis	0308	22
◌◌	*	ring above	030A	2A
ˆˆ	:	double acute	030B	3A

Table C-2 (Continued) Diacritical Marks

Diacritical mark	Compose char	Description	Diacritical hex code	Compose hex code
ˇ	v	hacek (caron)	030C	76
¸	,	cedilla	0327	2C
	;	ogonek	0328	3B
—	—	underline	0332	5F



Appendix D

Implementation Limits

This appendix describes aspects of the system's inner workings that impose restrictions on certain implementations.

Size Limitations

The following table gives the size limitations for various aspects of the system. A limit of "None" implies that no hard limit exists, though available address space is an upper bound in every case.

Table D-1 Size Limitations

Unit	Limit	Comment
Number of objects	None	Object memory grows dynamically
Object size	None	
Named instance variables	256 per class	Includes inherited instance variables
Method variables	255	Includes arguments, named temporary variables, unnamed temporary variables (needed to implement to:do: loops, etc.). Also includes pushes and pops, so the effective limit may be a little less.

Table D-1 Size Limitations

Unit	Limit	Comment
Block variables	255	Includes block arguments and temporaries; in some circumstances, it also includes arguments and temporaries from outer scopes to which the block refers. Also includes pushes and pops (see above).
Method literals	256	Includes ordinary literals (strings, numbers, etc.), message selectors (other than about 200 of the most common selectors), static variables (global, pool and class) that are referenced, and one for each block.
Block nesting	256 levels	
Method branches	1023 bytes, forward or backward	This does not limit the length of regular code. In practice, it means that the body of an open-compiled loop or conditional cannot be longer than 1023 bytes.

Open-coded Blocks

All control constructs in Smalltalk are defined as operations involving blocks, but, in the interest of performance, certain of these messages are optimized by the compiler—that is, the blocks are in-lined in the method. Ordinarily, literal blocks (code inside square brackets) create **BlockClosure** instances when compiled. These **BlockClosures** reference separate methods, and create separate contexts when invoked. By contrast, in-line optimized or "open-coded" blocks have their code generated inside the defining method and do not have associated **BlockClosures**; their arguments and temporaries are merged into the enclosing method's context as "compiler-generated temporaries."

The following constructs are subject to block open-code optimization by the compiler:

```
BlockClosure>>whileTrue
BlockClosure>>whileTrue:
BlockClosure>>whileFalse
BlockClosure>>whileFalse:
```

```
Boolean>>and:
Boolean>>or:
Boolean>>ifTrue:
Boolean>>ifFalse:
Boolean>>ifTrue:ifFalse:
Boolean>>ifFalse:ifTrue:
```

```
Number>>to:do:
Number>>to:do:by:
```

For the most part, blocks that are open-coded by the compiler have the same semantics as ordinary blocks. But there are some distinctions that should be noted with respect to context materialization and visibility.

Shared Context

True closed blocks are invoked in a separate **Context**. Therefore, the following code evaluates to **false**:

```
| outerContext answer |
outerContext := thisContext.
(1 to: 1) do: [:i | answer := thisContext == outerContext].
answer
```

On the other hand, the following evaluates to **true**:

```
| outerContext answer |
outerContext := thisContext.
1 to: 1 do: [:i | answer := thisContext == outerContext].
answer
```

Of course, one typically doesn't need to know the current context (and the compiler *could* refuse to open-code blocks that referred to **thisContext** directly). But code that inspects the context stack (such as the debugger, the

profiler or the exception-handling facility) can see the difference. This invariably is not a problem in practice.

Browser Visibility

Because the compiler rewrites the methods for open-coded blocks, the methods that use these messages do not register as senders of the message. For example, the expression `1 to: 5 do: [:i |]` is compiled as if it were:

```
| t1 |  
t1 := 1.  
[t1 <= 5] whileTrue: [ti := t1 + 1]
```

Hence, it isn't thought to be a sender of `to:do:` at all, (or even `whileTrue:` because that is also open-coded), but it *is* considered to be a sender of `+`. All this means in practice is that you can't use the browser to find the senders of the 12 messages listed in "Open-coded Blocks."

Block Optimization

There are three common patterns of usage for blocks in Smalltalk programs:

- n Arguments to `ifTrue:`, `ifFalse:`, `or:`, etc. These blocks are in-lined by the Compiler's open-coded block optimization facility. These are not blocks as far as the execution machinery is concerned and are not further discussed here.
- n Exception blocks (for example, passed as an argument to `ifAbsent:`). These blocks are created fairly frequently and almost never executed.
- n Iteration blocks (for example, passed as an argument to `do:`). These blocks are executed many times.

Because close-coded blocks are so common, both block creation and block invocation must be supported efficiently.

The `[]` language construct creates a `BlockClosure` object (except for open-coded blocks as mentioned above), which contains only the starting point and a reference to the "home" (the enclosing method); invoking a block (sending a `value` message to a `BlockClosure`) creates a `BlockContext`, which is almost exactly the same as a `MethodContext`—it has local arguments and temporaries of its own, and a receiver which is the `BlockClosure`. To reference variables in enclosing scopes (outer blocks, the enclosing method, or the

method's receiver or instance variables), the code in a block must chain back through the outer scope pointers stored in the block objects.

Because chaining through the outer scope pointers is relatively expensive, and because a straightforward implementation would always cause the home of a block to be materialized as a `Context`, we adopt a slightly more sophisticated implementation based on experience from the lexically scoped Lisp world. If the code in a block refers to variables from outer scopes, but it can be determined at compile time that the variables can't change their values after the block has been created, then the values can be copied into the `BlockClosure`, and the `BlockClosure` doesn't need an outer scope pointer, and the home doesn't have to be made hybrid.

Of course, a block with an explicit return needs a reference to its home whether it refers to outer variables or not. We refer to blocks with a home pointer as *full blocks*, and blocks without a home pointer as *clean blocks*. Blocks that refer to outer scope variables whose values do not change after block creation copy their values into the closure, and are called *copying blocks*; they are otherwise identical to clean blocks. If there is only a single copied value, it is stored in the closure; otherwise, the closure references an `Array` of copied values. The compiler generates different code for creating the block, depending on whether it is to be full, copying or clean. Clean `BlockClosures`, in fact, are created at compile-time.

Although Smalltalk's definition requires that procedure activations be accessible as first-class objects, our implementation only incurs the overhead of materializing an activation as a `Context` if some reference is made to it, which is not the common case.

Apart from this optimization, Smalltalk supports the semantics of blocks in a straightforward way. It has a `BlockClosure` class with three instance variables:

<code>outerScope</code>	a <code>MethodContext</code> or <code>BlockContext</code> for dirty blocks, nil for clean
<code>blocks</code>	
<code>method</code>	a <code>CompiledBlock</code>
<code>copiedValues</code>	nil for full or clean blocks, a single value or an <code>Array</code> for copying
<code>blocks</code>	

Note that creating a full block involves not only allocating a `BlockClosure` object, but also materializing the outer activation as a `Context`. This intro-

duces a substantial delayed cost, because the outer activation must be converted to an object when control returns from it, regardless of whether the block outlives it.

The Debugger

Because only full blocks have a reference to their outer-scope, the Debugger prints clean/copying block contexts as:

```
[ ] optimized
```

...rather than the following, which is used for full blocks:

```
[ ] in ReceiverClass>>someMethod
```

Performance

Try to make clean blocks if you care about performance. Blocks with ^ returns are full blocks. Blocks that reference outer-scope variables (even `self`) will be at least copying blocks. Some examples:

```
| t |  
[:x | t := x frobnicate] value: 1.  
t := t * 2.  
^t
```

The reference to `t` inside the block makes it at least a copying block, and worse, the change in `t`'s value after the block is created makes it full. Instead, you might write the following, which leaves the block clean:

```
| t |  
t := [:x | x frobnicate] value: 1.  
t := t * 2.  
^t
```

Non-overrideable Methods

Some methods are treated specially by the execution machinery and cannot be overridden. These optimizations are done for performance reasons. For

example, the `==` method in `Object` is hard-wired because it cannot be permitted to fail for obvious reasons. Any `==` method you create will not take effect.

Special Treatment Only at Compile Time

The following messages are treated specially by the compiler only at compile time:

```

anObject and: aBlock0
anObject or: aBlock0
anObject ifTrue: aBlock0
anObject ifFalse: aBlock0
anObject ifTrue: aBlock0 ifFalse: anotherBlock0
anObject ifFalse: aBlock0 ifTrue: anotherBlock0
aBlock0 whileTrue: anotherBlock0
aBlock0 whileTrue
aBlock0 whileFalse: anotherBlock0
aBlock0 whileFalse
aBlock0 repeat
anObject to: anotherObject do: aBlock1
anObject to: anotherObject by: aNumber do: aBlock1

```

If the receiver and/or argument(s) meet certain syntactic requirements, these messages are compiled open; otherwise, they are compiled as ordinary message sends. The following example is compiled as an ordinary message (and will cause a `messageNotUnderstood` error when executed).

```
1 and: 2
```

The requirements for open compilation are indicated in the message descriptions above. “aBlock0” or “anotherBlock0” means the receiver or argument must be a literal 0-argument block. “aBlock1” means the argument must be a literal 1-argument block. “aNumber” means the argument must be a literal number.

The effect of open compilation is that adding, removing or changing definitions of these messages in certain classes will have no effect on the execution of expressions that meet the open compilation requirements, specifically:

Any class

and:
or:
ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:

BlockClosure

whileTrue:
whileTrue
whileFalse:
whileFalse
repeat

Any class

to:do:
to:by:do:

Note that if the receiver and arguments do *not* meet the open compilation requirements, the expression is compiled as an ordinary message send. For example, the user can define a method in `Integer` for `and:`, and the method will be invoked as expected for:

```
1 and: 2.
```

Note that the requirements are syntactic, not semantic. For example,

```
something and: [some other thing]
```

compiles open, but

```
something and: someOtherThing
```

compiles as an ordinary message send, which is executed even if the value of `someOtherThing` turns out to be a `BlockClosure` at execution time. This is why correct definitions for the above-mentioned selectors must exist in classes `True` and `False` (for the first list above), `BlockClosure` (for the second list), and `Number` (for the third).

The user can change this state of affairs in a fairly straightforward way by modifying the compiler. `MessageNode class>>initialize` constructs the dictionary that determines what messages should be compiled open; each message has a corresponding transformation method defined in class `MessageNode`. Note that one must recompile the entire system for the changes to take full effect. Removing the conditionals (`ifTrue/False`) from the list is likely to produce unacceptable system performance.

*Note: Senders of these messages cannot be found using the **Browse?References To...** command.*

Special Treatment at Compile Time and Translation Time

Many messages are compiled using “special selector” opcodes. These opcodes have two different functions: They eliminate the need to store copies of the selector in the sender’s literals, and the translator knows how to compile some of them specially.

The translator treats the following selectors specially by generating machine code that performs certain explicit class checks before (or instead of) sending a message. As long as these selectors are compiled as “special selectors,” their definitions for the given classes are fixed and cannot be modified by the user.

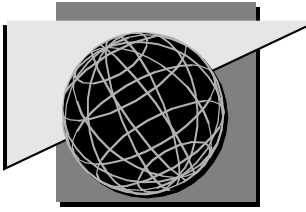
(SmallInteger) + (SmallInteger)
(SmallInteger) - (SmallInteger)
(SmallInteger) < (SmallInteger)
(SmallInteger) > (SmallInteger)
(SmallInteger) <= (SmallInteger)
(SmallInteger) >= (SmallInteger)
(SmallInteger) = (SmallInteger) literal
(SmallInteger) ~= (SmallInteger) literal
(Object) == (Object)

Note that `=` and `~=` are only translated specially if the argument is a literal number. Note also that if an addition or subtraction overflows, the expression is handled as a normal message send.

If the receiver or argument doesn’t meet the listed criterion, the expression is executed as a normal message send. Note that this is a *semantic* check carried out at runtime, not a syntactic one.

The special selectors are defined in `DefineOpcodePool class>>initializeSpecialSelectors` and `extendedSpecialSelectors`. The user can (carefully) modify the set of special selectors, subject to two constraints:

- n Either the special selectors that are handled specially by the translator must remain unchanged, or those indices in the table must not be used at all.
- n Changing the set of special selectors requires removing the old ones from the table, recompiling the entire system, adding the new ones, and recompiling the entire system again. The second recompilation is unnecessary if special selectors are only being removed, not replaced or added.



Appendix E

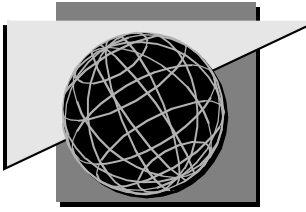
Keyboard Shortcuts

Table E-1 Canvas-painting Shortcuts

Task	Shortcut
Editing Text and Components	
Undo previous text edit	<L4> (Sun), <Command>-<z> (Mac)
Copy text or component	<L6> (Sun), <Command>-<x> (Mac)
Paste text or component	<L8> (Sun), <Command>-<v> (Mac)
Cut text or component	<Delete>, <L10> (Sun), <Com- mand>-<c> (Mac)
Displaying Tools and Dialogs	
Open Properties Tool	<Escape>-<p>
Display alignment dialog	<Escape>-<a>
Display distribution dialog	<Escape>-<d>
Display constrained layout dialog	<Escape>-<l>
Display equalize dialog	<Escape>-<e>
Display install dialog	<Escape>-<i>
Selecting Components	
Select next component	<Tab>

Table E-1 Canvas-painting Shortcuts

Task	Shortcut
Moving Components	
Move component one step toward front	<Control>-<f>
Move component one step toward back	<Control> -
Move component by one grid unit; if grid is off, move by one pixel	Arrow keys
Snap to grid	<Escape>-<s>
Aligning Components	
Align tops of selected components	<Escape>-<up-arrow>
Align bottoms	<Escape>-<down-arrow>
Align right edges	<Escape>-<right-arrow>
Align left edges	<Escape>-<left-arrow>
Grouping Components	
Group selected components	<Escape>-<g>
Ungroup	<Escape>-<u>
Changing Layouts	
Make component's position relative	<Escape>-<r>
Make position fixed	<Escape>-<f>
Changing Tool Focus	
In Properties Tool, shift focus to previous component	<Control>-<p>
Shift focus to next component	<Control>-<n>



Appendix F

User-Defined Primitives

The user-defined primitive (UDP) interface provides a means for interfacing software written in C or C++ to VisualWorks applications.

Note: The VisualWorks DLL and C Connect interface now supplants the user-defined capabilities described in this appendix. You are encouraged to migrate to the improved interface supplied by the VisualWorks DLL and C Connect product.

To create and use UDPs, you should be knowledgeable in both the C language and your computer's host operating system. In addition, you may need a particular compiler to link UDPs to VisualWorks—if this is the case for your platform, the *VisualWorks Installation Guide* has details.

Theory of Operation

VisualWorks allows you to extend the software development environment by accessing external C and C++ (referred to collectively as C from here on) functions and routines. To do this, you create user-defined primitives (UDPs). Briefly, the steps to create a UDP are:

1. Compile your C code module with a header file provided in the UDP interface. The header file includes a set of interface subroutines so that your C code and Smalltalk can share information. For example, the header file defines a call that does the equivalent of the `basicAt:` message in Smalltalk to get a particular field from an object.
2. Link the resulting object module with the Smalltalk library or object module. This results in the creation of a functional Smalltalk executable object engine file.
3. Start VisualWorks, using your new OE. One piece of the interface between your “C” code and Smalltalk is an entry point into your code for initialization, a routine called `UPinstall()`. When it starts up, the OE calls the `UPinstall()` routine you supply.

4. In addition to any particular initialization you may want to do, you must register the primitive in a table of UDPs, again through one of the interface routines.
5. In Smalltalk, write a method and declare your primitive. When that method is invoked, the primitive code will be called.

Basic Capabilities

The UDP interface provides your code with a variety of capabilities, including:

- n Access to the receiver and arguments of the message. This is automatic; they are the arguments to the primitive procedure.
- n An extensive set of routines for accessing these and other objects, following field references, converting between C and Smalltalk representations, and creating new objects.
- n Ability to exit the procedure, returning an object as the method's value or indicating to Smalltalk that the primitive did not finish successfully. A set of routines is provided for object return. Also, an interface is provided for so-called "primitive failure" (some primitive failures are invoked automatically by some of the interface routines).

Defining a New Primitive

To write a UDP, follow these steps (the rest of this appendix provides information in greater depth on these procedures):

1. Create a C routine called `UPinstall()` that adds your primitives by invoking the `UPaddPrimitive()` routine. The `primNumber` parameter is typically a number between 10,000 and 19,999, `primFunc` is the name of a C function, and `numArgs` is the number of parameters it takes.

```
UPaddPrimitive(primNumber, primFunc, numArgs);
```

2. Define your user primitive function in a `.c` file that includes `userprim.h`. The header file `userprim.h` contains `#define` macros and definitions common to all UDPs. A user-primitive handle (dubbed `aUpHandle` in examples later in this appendix) is an encoded identifier. The `upHandle` line in the code sample below declares receiver and the arguments of the primitive to have a data type of `upHandle`. This sets up

the data structure for receiving the encoded identifiers that will be passed from the UDP interface.

```
void primFunc(receiver, argument1, argument2,...)
    upHandle receiver, argument1, argument2, ...;
{
}
```

3. UDPs must either succeed (and return a value) or fail. The last executed line should be a return function such as either of the following examples:

```
UPreturnNil();
```

or

```
UPfail (failCode)
```

All together, this might look like:

```
#include "userprim.h"
void primFunc();
char *UPinstall()
{
    UPaddPrimitive(10003, primFunc, 2);
    return "Message to identify primitive(s)";
}
void primFunc(receiver, arg1, arg2)
    upHandle receiver, arg1, arg2;
{
    /* your code here */
    UPreturnNil();
}
```

4. Compile and link the new object engine.
5. In VisualWorks, create a method like the following:

```
prim: arg1 with: arg2
<primitive: 10003>
"failure code here"
```

Installation and Access

At startup, the OE calls the routine `UPinstall()` that was described previously. You must supply this routine.

1. To install your primitives, invoke the following routine for each primitive.:

```
UPaddPrimitive(primNumber, primFunc, numArgs);
```

2. Declare the arguments as follows:

```
int primNumber, numArgs;  
void (*primFunc)();
```

Calls to `UPaddPrimitive()` can be in any order.

Access to UDPs is through a Smalltalk method of the following form:

```
primName: args  
"comment"  
  
< primitive: primNumber >  
failure code
```

Here, `primNumber` is the number that was used to register the primitive with the `UPaddPrimitive()` call mentioned above. The “failure code” is a set of Smalltalk expressions for handling failure situations.

Any primitive that is called before it is installed will return a “primitive has failed” error.

Primitive Numbers

Each primitive in VisualWorks—whether predefined or defined by a user—is identified with a number. Numbers from 1 to 9,999 are reserved for current and future primitives. Numbers 20,000 and above are also reserved.

That leaves numbers from 10,000 to 19,999 open for user-defined primitives. You can create a maximum of 10,000 UDPs.

Arguments

Arguments can be passed from VisualWorks to the UDP code. The number of arguments must agree with the number of arguments supplied to the `UPaddPrimitive()` installation mechanism.

The UDP is always passed the receiver of the message followed by the arguments as they appear in the Smalltalk method (in left-to-right order). They appear to the C function correctly placed on the stack as if they were passed from a C calling function.

Data Types

Seven C data types are predefined for the UDP interface. They are used to convert between Smalltalk objects and standard C data types according to Table F-1.

Table F-1 UDP Data Types

“C” type	Convertible to objects of this type	“C” equivalent	Comment
<code>upHandle</code>			reference to a Smalltalk object
<code>upBool</code>	<code>true/false</code>	<code>TRUE/FALSE</code>	convention
<code>upByte</code>	<code>byte-type</code>	<code>unsigned char</code>	
<code>upInt</code>	<code>SmallInteger</code>	<code>int</code>	
<code>upFloat</code>	<code>Float</code>	<code>float</code>	
<code>upDouble</code>	<code>Double</code>	<code>double</code>	
<code>upChar</code>	<code>Character</code>	<code>char</code>	

Any object can be passed to or received from a user-defined primitive. For a composite object such as a nested array, build a struct with a matching architecture.

Failure Codes

If a UDP succeeds, it returns a value to the invoking Smalltalk routine; if it fails, it does not. In case of failure, a system primitive is available to return a failure code, namely, the argument to `UPfail()`.

The argument to `UPfail()` must be a C integer, an `upInt`. The UDP interface defines the failure codes listed in Table F-2 and in `userprim.h`. To return this failure code for examination, send class `UserPrimitives` the `failCode` message defined in `userprim.st`

Table F-2 Failure Codes

Number.	Name	Description
0		No failure code available.
-1	UPECrange	“C” argument out of range; probably not large enough data.
-2	UPESrange	Smalltalk argument out of range.
-3	UPEnonHandle	Handle argument not a handle.
-4	UPEwrongClass	Handle argument incorrect type.
-5	UPEargCheckFailed	Declared argument wrong type.
-6	UPEintNotSmallInteger	C int not a SmallInteger.
-7	UPEobjectTooSmall	C data structure too small.
-8	UPEconversion-TooSmall	Smalltalk datum too small to be represented in C.
-9	UPEconversionToo-Big	Smalltalk datum too big to be represented in C.
-10	UPEallocationFailed	Smalltalk object memory allocation failed.
-11	UPEargCountMatch	Argument count for this primitive doesn’t match registered count.
-12	UPEprimitiveNotInstalled	The primitive is not installed.

Table F-2 (Continued)*Failure Codes*

Number.	Name	Description
-13	UPEtooManyArgu- ments	Too many arguments.
-14	UPEnoReturnValue	The primitive returned with no value.
-15	UPEassertFail	The primitive invoked an assertion fail- ure.

The standard C global error number (errno) is not affected by failure code assignments.

General Advice

- n The UDP is passed opaque handles to Smalltalk objects, not actual object pointers (OOPs). The only meaningful role of these handles is to pass them as object references to the UDP interface. UDPs cannot obtain valid pointers into Smalltalk object memory. Do not try to create direct C pointers to object memory.
- n UDPs are capable of corrupting Smalltalk object space.
- n The UDP interface expects the same number of arguments as are defined for each routine.
- n The maximum number of arguments that a UDP can have is 32. You cannot alter this limit.
- n Many operating system calls, such as `read()`, can cause VisualWorks to block until the call is completed. Your end user may find it annoying if this system call cannot be satisfied within a reasonably short amount of time.
- n There is no way to interrupt a UDP or a system call from VisualWorks. `<Control>-c` does not cause anything to happen until the UDP returns control to VisualWorks.
- n The call stack above the current call frame does not conform to C calling conventions.
- n Since any allocation can fail, perform all allocations before provoking side effects in any data structure.
- n A `upHandle` (user primitive handle) is only valid during the current call to a UDP. If you try to save it across primitive calls you may cause primitive failure or a program crash. If an object is needed from call to call,

make it an instance variable of the receiver, or pass it as an argument on each call, or use the registry discussed in “Registering Long-lived Objects” on page 435.

- n C uses 0-based indexes, while VisualWorks uses 1-based indexes.
- n It is poor style to install the same primitive in radically different classes. It’s true that you can pass different types of arguments to the same function, because a number of type-checking routines are provided. However, for each UDP there is a fixed number of arguments, as specified at UDP installation.
- n If a Smalltalk datum is too big to be represented in C, the routine fails with either `UPEconversionTooSmall` or `UPEconversionTooBig` as the error message. (See “Data Types” on page 417 for valid source and destination data types.)

C Conversion

In each of the following routines, the copying begins at the `startingAt` index into the 1-based Smalltalk object (`aUpHandle`). It ends after either the length of `aUpHandle` has been reached or `aCount` elements have been copied. The result is not null-terminated. These routines cause the primitive to fail if the conversion is not possible. (See “Failure Codes” on page 418 for more information about primitive failure.)

String to String

Copy a C string (`theString`) to the Smalltalk string referenced by `aUpHandle`. Return the number of characters copied.

```
upInt UPcopyCtoSTstring
(aUpHandle, theString, aCount, startingAt)
upHandle aUpHandle;
upChar *theString;
upInt aCount, startingAt;
```

Byte Array to Byte Object

Copy a C array (`theBytes`) to the Smalltalk byte object referenced by `aUpHandle`. Return the number of elements copied.

```

upInt UPcopyCtoSTbytes
(aUpHandle, theBytes, aCount, startingAt)
upHandle aUpHandle;
upByte *theBytes;
upInt aCount, startingAt;

```

Integer Array to Array

Copy a C array (theInts) to the Smalltalk array referenced by aUpHandle. Return the number of elements copied.

```

upInt UPcopyCtoSTintArray
(aUpHandle, theInts, aCount, startingAt)
upHandle aUpHandle;
upInt *theInts;
upInt aCount, startingAt;

```

Float Array to Array

Copy a C array (theFloats) to the Smalltalk array referenced by aUpHandle. Return the number of elements copied.

```

upInt UPcopyCtoSTfloatArray
(aUpHandle, theFloats, aCount, startingAt)
upHandle aUpHandle;
upFloat *theFloats;
upInt aCount, startingAt;

```

Integer to Integer

Copy a C integer (aUpInt) into a Smalltalk integer. Only integers from $-(2^{29})$ to $(2^{29} - 1)$ can be represented.

```

upHandle UPctoSTint(aUpInt)
upInt aUpInt;

```

Float to Float

Copy a C floating-point number (`aUpFloat`) to a Smalltalk float.

```
upHandle UPCtoSTfloat(aUpFloat)
  upFloat aUpFloat;
```

Double Float to Double

Copy a C double-precision floating-point number (`aUpDouble`) to a Smalltalk double.

```
upHandle UPCtoSTdouble(aUpDouble)
  upDouble aUpDouble;
```

Boolean to Boolean

Copy a C boolean (`aUpBool`) to a Smalltalk boolean (`true` or `false`).

```
upHandle UPCtoSTbool(aUpBool)
  upBool aUpBool;
```

Character to Character

Copy a C character (`aUpChar`) to a Smalltalk character.

```
upHandle UPCtoSTchar(aUpChar)
  upChar aUpChar;
```

Return nil

Return the Smalltalk `upHandle nil`.

```
upHandle UPnil()
```

Smalltalk Conversion

In each of the following routines, the copy begins at the **startingAt** index into the one-based Smalltalk object to the C data type beginning with index zero. It ends after **aCount** number of elements or the remaining size object have been copied. These routines cause the primitive to fail if the conversion is not possible. (See also the section on Failure Codes.)

String to String

Copy a Smalltalk string (**aUpHandle**) to a C string (**aUpString**). Return the number of characters copied. Note that the resulting C string is *not* null-terminated.

```
upInt UPcopySTtoCString
(aUpHandle, aUpString, aCount, startingAt)
upHandle aUpHandle;
upChar* aUpString;
upInt aCount, startingAt;
```

Byte Array to Byte Array

Copy a Smalltalk byte object (**aUpHandle**) to a C byte array (**aUpBytes**). Return the number of elements copied.

```
upInt UPcopySTtoCbytes
(aUpHandle, aUpBytes, aCount, startingAt)
upHandle aUpHandle;
upByte* aUpBytes;
upInt aCount, startingAt;
```

Integer Array to Array

Copy a Smalltalk integer array (**aUpHandle**) to a C integer array (**aUpInt**). Return the number of elements copied.

```
upInt UPcopySTtoCintArray
(aUpHandle, aUpInt, aCount, startingAt)
upHandle aUpHandle;
```

```
upInt *aUpInt;  
upInt aCount, startingAt;
```

Float Array to Array

Copy a Smalltalk array (`aUpHandle`) to a C array of floating point numbers (`aUpfloatArray`). Return the number of elements copied.

```
upInt UPcopySTtoCfloatArray  
  (aUpHandle, aUpfloatArray, aCount, startingAt)  
upHandle aUpHandle;  
upFloat *aUpfloatArray;  
upInt aCount, startingAt;
```

Integer to Integer

Copy a Smalltalk integer (`aUpHandle`) to a C integer (`upInt`).

```
upInt UPSTtoCint(aUpHandle)  
upHandle aUpHandle;
```

Float to Float

Copy a Smalltalk floating point number (`aUpHandle`) to a C float (`upFloat`).

```
upFloat UPSTtoCfloat(aUpHandle)  
upHandle aUpHandle;
```

Double Float to Double

Copy a Smalltalk double-precision floating-point number (`aUpHandle`) to a C double.

```
upDouble UPSTtoCdouble(aUpHandle)  
upHandle aUpHandle;
```

Character to Character

Copy a Smalltalk character (`aUpHandle`) to a C character (`upChar`).

```
upChar UPSTtoCchar(aUpHandle)
upHandle aUpHandle;
```

Boolean to Boolean

Copy a Smalltalk boolean (`aUpHandle`) to a C boolean (`upBool`).

```
upBool UPSTtoCbool(aUpHandle)
upHandle aUpHandle;
```

Success Return

Use one of the following constructs to return a value to the invoking Smalltalk routine.

Any Value

Return a value (`aUpHandle`) to the invoking Smalltalk code.

```
void UPreturnHandle(aUpHandle)
upHandle aUpHandle;
```

Nil

Return nil to the invoking Smalltalk code.

```
void UPreturnNil()
```

True

Return true to the invoking Smalltalk code.

```
void UPreturnTrue()
```

False

Return `false` to the invoking Smalltalk code.

```
void UPreturnFalse()
```

Failure Return

If a UDP fails, and if the invoking method contains Smalltalk expressions after the `<primitive #>` statement, the Smalltalk expressions are executed. Common causes for failure include the type checking and type conversion routines.

Also see “Failure Codes” on page 418.

Coded Failure

Unaffected by `UPinstallErrorHandler()`; `UPfail()` always fails. The `failcode` is an `upInt`, and is accessible via primitive 9999. Do not use 0.

```
void UPfail(failCode)
    upInt failCode;
```

Type Checking

Use one of the following routines to check the data type of an argument that has been passed from a Smalltalk program.

Character

Return `TRUE` if `aUpHandle` is a character, otherwise `FALSE`.

```
upBool UPisCharacter(aUpHandle)
    upHandle aUpHandle;
```

Fail (that is, abort execution of the primitive) if `aUpHandle` is not a character.

```
void UPshouldBeCharacter(aUpHandle)
    upHandle aUpHandle;
```

String

Return TRUE if aUpHandle is a string, otherwise FALSE.

```
upBool UPisString(aUpHandle)
```

Fail if aUpHandle is not a string.

```
void UPshouldBeString(aUpHandle)
    upHandle aUpHandle;
```

Integer

Return TRUE if aUpHandle is an integer, otherwise FALSE.

```
upBool UPisInteger(aUpHandle)
    upHandle aUpHandle;
```

Fail if aUpHandle is not an integer.

```
void UPshouldBeInteger(aUpHandle)
    upHandle aUpHandle;
```

Float

Return TRUE if aUpHandle is a floating point number, otherwise FALSE.

```
upBool UPisFloat(aUpHandle)
    upHandle aUpHandle;
```

Fail if aUpHandle is not a floating point number.

```
void UPshouldBeFloat(aUpHandle)
    upHandle aUpHandle;
```

Double

Return TRUE if aUpHandle is a double-precision floating-point number, otherwise FALSE.

```
upBool UPisDouble(aUpHandle)
    upHandle aUpHandle;
```

Array of Integers

Return TRUE if aUpHandle is an array of integers, otherwise FALSE.

```
upBool UPisArrayOfInteger(aUpHandle)
    upHandle aUpHandle;
```

Fail if aUpHandle is not an array of integers.

```
void UPshouldBeArrayOfInteger (aUpHandle)
    upHandle aUpHandle;
```

Array of Floats

Return TRUE if aUpHandle is an array of floating point numbers, otherwise FALSE.

```
upBool UPisArrayOfFloat(aUpHandle)
    upHandle aUpHandle;
```

Fail if aUpHandle is not an array of floating point numbers.

```
void UPshouldBeArrayOfFloat (aUpHandle)
    upHandle aUpHandle;
```

Byte Array

Return TRUE if aUpHandle is a byte array, otherwise FALSE.

```
upBool UPisByteArray(aUpHandle)
    upHandle aUpHandle;
```

Fail if aUpHandle is not a byte array.

```
void UPmustBeByteArray(aUpHandle)
    upHandle aUpHandle;
```

Byte-like

Return TRUE if obj contains only bytes (that is, no OOPs), otherwise FALSE.

```
upBool UPisByteLike(obj)
    upHandle obj;
```

Fail if aUpHandle is not byte-like.

```
void UPmustBeByteLike(aUpHandle)
    upHandle aUpHandle;
```

Boolean

Return TRUE if aUpHandle is a boolean, otherwise FALSE.

```
upBool UPisBoolean(aUpHandle)
    upHandle aUpHandle;
```

Fail if aUpHandle is not a boolean.

```
void UPmustBeBoolean(aUpHandle)
    upHandle aUpHandle;
```

Immediate

Return TRUE if aUpHandle is an immediate object, otherwise FALSE.

```
upBool UPisImmediate(aUpHandle)
    upHandle aUpHandle;
```

Class Check

Given handles for an object and a class, return TRUE if the object belongs to that class or its superclass, otherwise return FALSE.

```
upBool UPisKindOf(objUpHandle, classUpHandle)
    upHandle objUpHandle classUpHandle;
```

Object Allocation

If an allocation fails, the primitive fails. Aggregate initialization operations terminate as soon as numElements is reached.

String

Allocate an instance of String, numElements in size, all elements of which are initialized to cvalue.

```
upHandle UPallocString (cvalue, numElements)
    upChar cvalue;
    long numElements;
```

Byte Array

Allocate an instance of ByteArray, numElements in size, all elements of which are initialized to bvalue.

```
upHandle UPallocByteArray (bvalue, numElements)
    upByte bvalue;
    long numElements;
```

Array

Allocate an instance of `Array`, `numElements` in size, all elements of which are initialized to `ovalue`.

```
upHandle UPAllocArray (ovalue, numElements)
    upHandle ovalue;
    long numElements;
```

Other Object Types

Allocate a fixed-size object of the given class (initialized to nil), and return a handle for the instance.

```
upHandle UPAllocFsObject(aUpHandle)
    upHandle aUpHandle;
```

Allocate a variable-size object of the given class (initialized to nil if it's a pointer, otherwise 0), and return a handle for the instance.

```
upHandle UPAllocVsObject(classHandle, size)
    upHandle classHandle;
    upInt size;
```

Indexed Access

These routines cause the primitive to fail if the operation is not possible or if `index` is out of bounds. The lowest legal value for `index` is 1 (not 0).

Indexed Variable

Return the `index`'th element in `arrayUpHandle`.

```
upHandle UPbasicAt(arrayUpHandle, index)
    upHandle arrayUpHandle;
    upInt index;
```

Replace the `index`'th element of `arrayUpHandle` with `upHandleToBePut`.

```
void UPbasicAtPut(arrayUpHandle, index, upHandleToBePut)
    upHandle arrayUpHandle, upHandleToBePut;
    upInt index;
```

Instance Variable

Return the value of the `index`'th instance variable.

```
upHandle UPinstVarAt(aUpHandle, index)
    upHandle aUpHandle;
    upInt index;
```

Replace the `index`'th instance variable in `aUpHandle` with `upHandleToBePut`.

```
void UPinstVarAtPut(aUpHandle, index, upHandleToBePut)
    upHandle aUpHandle, upHandleToBePut;
    upInt index;
```

Indexed Byte

Return the `index`'th byte from `aUpHandle`, which must be byte-like.

```
upInt UPbyteAt(aUpHandle, index)
    upHandle aUpHandle;
    upInt index;
```

Replace the `index`'th byte in `aUpHandle`, which must be byte-like.

```
void UPbyteAtPut(aUpHandle, index, aUpByte)
    upHandle aUpHandle;
    upInt index;
    upByte aUpByte;
```

Indexed Float

Return the `index`'th float in `aUpHandle`, which must be an array.

```
upFloat UPfloatAt(aUpHandle, index)
    upHandle aUpHandle;
    upInt index;
```

Replace the `index`'th element of the array called `aUpHandle` with `aUpFloat`. This routine does not return any value. It fails if `index` is out of bounds.

```
void UPfloatAtPut(aUpHandle, index, aUpFloat)
    upHandle aUpHandle;
    upInt index;
    upFloat aUpFloat;
```

Sizing

Return the number of named instance variables in `obj`.

```
upInt UPinstVarSize(obj)
    upHandle obj;
```

Return the size of the variable portion of this object in number of `upHandles` (for pointer objects) or number of bytes (for nonpointer objects).

```
upInt UPindexVarSize(obj)
    upHandle obj;
```

Initializing

Install an error handler (a user-supplied function returning `void`), which is called when a support routine detects an error condition. The error handler is called with one argument, the C `int` failure code. This error handler can fail the primitive (using `UPfail()`, etc.), exit the primitive successfully (using `UPreturnHandle()`, for example) or transfer control elsewhere (using `setjmp/longjmp`).

```
void UPinstallErrorHandler (errorHandler)
    void (*errorHandler)();
```

Initialize the user primitive interface and return a herald string. This must be defined in the user primitive code module.

```
char *UPinstall()
```

Install `primFunc` as primitive # `primNumber` with `numArgs` number of arguments. Return `TRUE` if successful.

```
upBool UPaddPrimitive (primNumber, primFunc, numArgs)
    upHandle (*primFunc)();
    upInt primNumber, numArgs;
```

Other Support Routines

Return a handle for the class of the `aUpHandle` object.

```
upHandle UPclass(aUpHandle)
    upHandle aUpHandle;
```

Return a handle from the registry.

```
upHandle UPregisteredHandleAt(aUpInt)
    upInt aUpInt;
```

Put a handle in the registry at a specified slot.

```
void UPregisteredHandleAtPut(index, handle)
    upInt index;
    upHandle handle;
```

Allocate a slot in the registry and return the slot index.

```
upInt UPallocRegistrySlot();
```

Given a handle for a class, return an integer indicating whether the class has fixed-size instances, variable-size instances, or is not a class.

```
#define UPnotAClass 0
#define UPfixedSizeClass 1
#define UPvariableSizeClass 2
upInt UPclassType(aUpHandle)
    upHandle aUpHandle;
```

Given a handle to a semaphore, signal the semaphore.

```
void UPsignalSemaphore(aUpHandle)
    upHandle aUpHandle;
```

Registering Long-lived Objects

The object engine (OE) maintains a system registry of objects that must be referenced by the object engine code. This is needed because the OE relocates objects during memory management operations, so direct references to object memory (and `upHandles` are such references) cannot persist across primitive calls.

To refer to objects over time, the OE provides a facility to register indirect references to objects. These indirect references are indices in a table that the OE memory manager keeps current.

Indices for special objects such as `nil`, `true` and `false` are defined in `userprim.h`, so you can call:

```
UPregisteredHandleAt(nilOopX)
```

to get a handle on `nil`, giving the same result as the `UPnil()` call. More interestingly, you can call:

```
UPregisteredHandleAt(byteArrayClassX)
```

to get a handle on class `ByteArray`.

Another important reason to use the system registry is to reference an object that was given to a primitive in a prior call. For example, imagine you have a

primitive that was passed a `Semaphore` so it could be signalled later (by using `UPsignalSemaphore()`). Your primitive can ask the registry for a permanent slot (it returns a table index if there is enough room) and store the semaphore into the registry at that slot. You would presumably record the slot index in a static variable in the C code. Later, to reference the object, you can read the registry at the slot you recorded.

Registry slots are a finite resource, and cannot be recycled (while the OE is running), so use them sparingly. To reserve a registry slot, call:

```
static upInt slot;  
slot = UPAllocRegistrySlot();
```

Any slots you allocate (and the references you store in them) are discarded in a snapshot file. So you can not use the registry to keep objects from being garbage-collected across snapshots. To do that, use the normal Smalltalk techniques (such as keeping the object in a class variable) and re-register your slots when the snapshot starts up.

An object that has a reference in the registry, however, will not be garbage-collected while the OE is running—so it's a good idea to store nil in a registry slot when you are done with it.

Interrupts and Poll Handlers

In some situations, a Smalltalk program needs to respond to a condition that can only be detected asynchronously (such as in a UNIX signal-handler, or a PC interrupt-handler). In such a case, it is not possible to access object memory, because the event might occur while the OE is doing some operation that involves relocating objects. You could create a Smalltalk process that periodically calls a primitive to see whether that event has occurred (by examining a C static variable, perhaps, or by making an operating system call), but this constant polling can be inefficient or inconvenient.

As an alternative, call the support routine `UPpostInterrupt()` from your asynchronous signal/interrupt-handler. This routine arranges to have a poll-handler called soon thereafter—before the next backward branch (in any loop) or frame-building send (all sends build frames except those that simply return a variable or call a primitive). This is the only UDP support routine you can call asynchronously.

The poll-handler must be registered prior to the interrupt, by inserting the following call in a primitive or in your `UPinstall()` routine:

```
upFunc myHandler;  
UPinstallPollHandler(myHandler);
```

Your handler will only be called once, no matter how many times `UPpostInterrupt()` is invoked in the interim. Posting an interrupt just sets a flag, which is cleared only when your poll-handler is called.

The handler will usually determine what event has occurred (again, by examining a C static variable or by making an OS call) and possibly signal a semaphore that was stored previously in the registry. The only UDP support routines that a handler can safely call are:

```
UPsignalSemaphore()  
UPregisteredHandleAt()  
UPregisteredHandleAtPut()
```

On UNIX platforms, the OE requires unimpeded access to certain signals. Specifically, your user primitives should not establish signal handlers for the following signals:

```
SIGIO or SIGPOLL  
SIGALRM  
SIGVTALRM  
SIGCHLD
```

In addition, user primitives should not make system calls that generate the last three signals listed above. Operations that generate `SIGIO` or `SIGPOLL` will not cause problems.

Unsafe Primitives

The standard UDP interface is very defensive about bugs in primitive code (and in the Smalltalk code that calls it). Bounds-checking is performed on indices, class-checking is performed during coercion, etc. However, such runtime checks have a negative performance impact.

For situations in which critical performance needs motivate you to bypass the safety mechanisms, you can compile your code defining the symbol `UNSAFE`. You can usually just pass the switch `-DUNSAFE` to the compiler when you compile your program.

We don't recommend that you do this in ordinary practice.

Even in “safe” mode, your primitives should perform consistency checks on passed arguments—at least checking for the correct class. When you use the `UNSAFE` implementation, doing these checks is absolutely essential for correct operation of the system. Test your primitive code *very* carefully before compiling with `-DUNSAFE` and using the resulting OE on a useful image.

When a primitive is compiled `UNSAFE`, it must be installed `UNSAFE`. In other words, the invocation of `UPaddPrimitive` that installs a given primitive must be compiled with the same safety level as the primitive being installed. This is because the safe and unsafe implementations have different argument-passing mechanisms.



Warning: *If you compile your primitives with `-DUNSAFE`, you will get an equivalent user-primitive support interface. It will run somewhat faster, but bugs in your primitives can crash the resulting OE (rather than fail). Worse yet, some bugs can silently corrupt the virtual image (VI) without crashing. Finally, because hard-to-trace errors can be introduced so easily in unsafe mode, an OE built with `-DUNSAFE` and its derived VI's will not be handled by ParcPlace-Digitalk technical support.*

Specific OE implementation details that are exposed by the code in the files `unsafe_oops.h` and `up_unsafe.h` do *not* represent a supported OE interface. They will change from release to release, and have hidden dependencies and restrictions, which may render them useless outside this context. In other words, pretend that you can't read those files.

Example

The following example illustrates some of the basic elements of the UDP interface. In this example, the C library regular expression parser is linked into VisualWorks as UDP # 10000.

C Code

```
#include "userprim.h"
/* Install user-defined primitives */
char *UPinstall()
{
    void RegEx();
```



```

UPaddPrimitive(10000, RegEx, 2);
return "Regular expression parser";
}

typedef upChar *upString;

/* Match the string against the regular expression string. The
 * regular expression language is compatible with the one
 * defined for the UNIX ed(1) * editor. Returns TRUE if it
 * matches, FALSE otherwise. Fail if the expression is badly
 * formed, or we can't allocate memory for the buffers */

void RegEx(recv, expressionHandle, testStringHandle)
    upHandle recv, expressionHandle, testStringHandle;
{
    upInt expressionSize, testStringSize, UPindexVarSize();
    upString errorMessage, re_comp(), UPtoString(), malloc(),
        expressionString, testString;
    int result;

    expressionSize = UPindexVarSize (expressionHandle);
    if ((expressionString = malloc (expressionSize+1)) == NULL)
        UPfail(1); /* FAIL "expression too long" */

    (void)UPcopySTtoCString (expressionHandle,
        expressionString, expressionSize, 1);
    expressionString[expressionSize] = '\0';

    testStringSize = UPindexVarSize(testStringHandle);
    if ((testString = malloc(testStringSize+1)) == NULL)
        {
            (void)free(expressionString);
            UPfail(2); /* FAIL "test string too long" */
        }

    (void)UPcopySTtoCString(testStringHandle,testString,
        testStringSize, 1);
    testString[testStringSize] = '\0';

    /* Compile the regular expression, re_comp() indicates a bad
     * regular expression by returning a C string */

```

```
errorMessage = re_comp(expressionString);
if (errorMessage)
    (void) free (expressionString);
    (void) free (testString);
    UPfail(4); /* FAIL*/

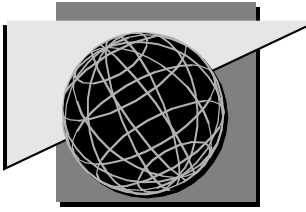
/* execute the regular expression */
result = re_exec (testString);

(void) free (expressionString);
(void) free (testString);
switch (result)
{
    case -1: UPfail(5); /* FAIL "Badly formed expression" */
    case 0: UPreturnFalse(); /* NO MATCH */
    case 1: UPreturnTrue(); /* MATCH */
}
/* NOT REACHED */
}
```

Smalltalk Code

In the testing protocol of a `Testing` class:

```
match: this with: that
<primitive:10000>
^self primitiveFailed
```



Index

Symbols

<Control>-click xxii
<Meta>-click xxii
<Operate> button xxi
<Select> button xxi
<Shift>-click xxii
<Window> button xxi

A

A 96
abstract class, defined 19
abstract superclass 67
animation 299

- double buffering 300
- flashing 299

answer set 197

- cancelling 205
- describing 200
- handling multiple 198
- using an output template 202

answer stream 201
application

- building 1

application building tools 131–155
arc, defined 291
arguments

- in user-defined primitives 417

ArithmeticValue class 67
array 27

Array class

creating an instance 73
defined 70
finding elements 77
methods 75
replacing elements 75
See also collection
ArrayedCollection class 79, 85
Aspect property, defined 141
assignment 29
atomic term, syntax 391

B

Bag class

adding elements 75
counting elements 77
defined 70
sorting elements 77
See also collection
behavior, defined 11
bezier curve, defined 291
binary message 37
bitmap, defined 274
block

- clean, defined 405
- copying, defined 405
- full, defined 405
- limitations 403–407
- optimized 402

block expression 42
block optimization 404–410
BlockClosure class 47, 402
boolean 27
boolean objects 47
branching 47–48
brightness, defined 307
buffers and adaptors 200
Builder 1, 142
bulletin boards xxv
buttons, mouse. *See* mouse buttons
ByteArray class, defined 71
ByteString class, defined 86

C

CachedImage class, defined 297

canvas

- alignment grid 132
- creating 157
- fence 132
- UI look-and-feel 132
- See also* Canvas Tool

Canvas Tool, defined 132

capitalization 23

cascade 42

case statement 48

catalog query 206

category

- adding 118
- class 15
- editing 119
- filing a class description 118
- opening a browser 118
- printing description of each class 118
- removing 118
- renaming 118
- updating 119

Change List 179, 182, 215

adding file contents 112
browsing 182
checking for conflicts 113
commands 112–113
condensing 185
defined 109–113
deleting entries 112
displaying entries 111, 112, 183
executing changes 112
filing change entries 112
filter switches 110–111
opening 112
removing entries 112
sharing code between images 184
unmarking entries 112
views 110
See also changes

Change Set 179

clearing 182
defined 180
displaying changes 183
inspector 181
opening 181
sharing code between images 184
types of changes 181
updating 182
See also changes

changes

displaying 183
managing 109–114
moving between images 180
removing during fileout 182
See also Change List, Change Set

character

categorizing an instance 82
comparing 83
creating 81–83
defined 81
extended set 81

- operations 82–83
- test messages 82
- wildcard symbols 85
- working with 83–85
- See also* string
- Character class 81
 - syntax 388
- character literal 26
- CharacterArray class 85
 - defined 86
- checkForEvents message 262
- circle, defined 292
- class
 - abstract, defined 19
 - defined 13
 - displaying hierarchy 119
 - editing 120
 - filing a description 119
 - filtering in Finder 131
 - listing instance variables 120
 - mapping to database tables 157
 - mapping to relational data type 189
 - method 117
 - moving to other category 120
 - name 34
 - opening a browser 119
 - printing a description 119
 - removing 120
 - renaming 120
 - searching for 119
 - updating in Finder 132
 - variable 32
 - viewing methods 120
- class category 15
- class category view 116
 - commands 118–119, 166
- class inheritance 16–17
- class library 15
 - organization 16
- class method 14
- class variable, defined 13
- class view 116
 - commands 119–120, 166
- clean block, defined 405
- click xxii
- clipping, using GraphicsContext 285
- closed system 367
- code
 - compiling 123
 - executing 123
 - formatting 44, 123
 - inserting explanation 124
 - organizing 179
 - parsing and compiling 323–325
 - printing 124
 - putting in PermSpace memory 332
 - reverting to prior version 184
 - searching for a class 132
 - sharing between applications 2
 - sharing between images 184
 - supporting a component 154
 - testing 124
- code view 116, 127
 - commands 123–124, 168
 - defined 115
- collection
 - adding elements 74–76
 - choosing a class 69
 - comparing 76
 - converting 78
 - copying 78
 - counting elements 77
 - creating an instance 73
 - finding elements 77
 - hierarchy 78–79
 - looping methods 50–52
 - operations 69–79
 - printing 78

- removing elements 74–76
- replacing elements 74–76
- types of classes 69–72
- Collection class 78
 - categories 79
 - types of 69–72
- color 303–313
 - constants 305
 - dithering 313
 - hue-saturation-brightness (HSB) values 305
 - map 310
 - red-green-blue (RGB) values 305
 - types of 303
 - See also* palette, pattern
- color properties
 - defined 147
- color rendering
 - policies 311–313
 - types of 312
- ColorValue class
 - creating an instance 305
 - defined 304
- combination rules, defined 297
- commands
 - change list 112–113
 - change list switches 111
 - class category view 118–119, 166
 - class view 119–120, 166
 - code view 123–124, 168
 - contents view 109
 - Menu Editor (enhanced) 137–140
 - method view 122, 168
 - names view 108
 - Parcel menu 162
 - pattern view 107
 - protocol view 121, 165, 167
 - stack view 129
 - Utility menu 163
- Compiler class 325
 - defined 323
- component
 - defined 301
 - generating supporting code 154
- component, *see* visual component
- composite object 9
- composite views 263
- CompositePart, as window component 263
- CompositeView 263
- conditional looping, defined 48
- conditional selection 47
- connect string 191
- connection coordinator 207
- constants 24
- container, window as a 258
- contents view 106
- context stack 171
- control structure 47–52
 - methods 52–53
- controlActivity 269
- controlInitialize 268
- Controller 266, 271
- controller
 - activity loop 269
 - defined 264
 - event methods 271
- ControlManager 266, 267
- conventions
 - naming 12, 13
 - screen xx
 - typographic xix–xx
- copying block, defined 405
- coverage, in a Mask 283
- crash recovery 215
- creating
 - array instance 73
 - canvas 157
 - character 81
 - collection instance 73

- ColorValue instance 305
 - date instance 60–61
 - exception 96
 - file 105
 - float instance 57
 - fraction instance 59
 - illustrations 135
 - image instance 295
 - integer instance 55
 - interval instance 73
 - menu 135
 - parser instance 324
 - point instance 276
 - process 87
 - project 179
 - random instance 59–60
 - rectangle instance 277
 - scanner instance 323
 - signal instance 94
 - string 83
 - time instance 64
 - cursor 298
 - changing the current 299
 - creating 298
 - displaying 299
 - hot spot 298
 - Cursor class
 - defined 298
 - curves 291
- D**
- data compactor, defined 339
 - data form
 - embedded, defined 158
 - linked, defined 158
 - data heap 334
 - Data Modeler 157
 - data type
 - for user-defined primitive 417
 - database
 - accessing 187–??
 - connecting to 188, 190
 - controlling transactions 207
 - default connection 192
 - disconnecting from 193
 - interaction with Smalltalk 189
 - mapping data type to Smalltalk class 189
 - mapping tables to classes 157
 - modifying canvas 157
 - reconnecting a restarted image 214
 - relational data types 189
 - saving connected image 213
 - signal hierarchy 212
 - types of errors 210
 - See also* transaction
 - database tools 157–159
 - Canvas Composer 157
 - data forms 158
 - Data Modeler 157
 - Mapping Tool 158
 - Query Editor 158
 - Date class 60
 - creating an instance 60–61
 - methods 62
 - dates 60–63
 - accessing information 62
 - arithmetic functions supported 61
 - comparing 61
 - printing 62–63
 - Debugger 126–130
 - continuing program execution 129
 - copying stack 129
 - debugging techniques. *See* debugging techniques
 - displaying message selectors 129
 - displaying more stack elements 129
 - opening 173

- opening a browser 129
- printing block contexts 406
- restarting program execution 129, 175, 177
- stepping through the code 129, 178
- views 126–129
- debugging techniques
 - inserting status messages 176
 - inspecting and changing variables 175
 - interrupting a program 177
 - reading the execution stack 171–172
 - restarting a program 177
 - tracing the flow of messages 173–174
- define dialog 140
 - defined 154
 - opening 154
- Definer 1–2
- Delay class, defined 92
- DependentComposite 263
- DependentPart
 - role in a view hierarchy 261
- diacritical marks 399
- Dictionary class
 - adding element from other dictionary 76
 - defined 72
 - finding elements 77–78
 - See also* collection
- directory
 - working with 105–108
 - See also* File List
- display surface
 - and snapshots 283
 - types of 280
- display, organizing 179
- DisplaySurface class, defined 280
- dithered color 313
- documentation. *See* VisualWorks documentation
- double buffering, in animation 300
- double-click xxii
- drop source properties 150

- drop target properties 151–153

E

- Eden, defined 330
- electronic bulletin boards xxv
- electronic mail xxv
- ellipse, defined 292
- Emergency Evaluator 219
- Emergency exit 219
- enumeration methods 77
- environment string 191
- environment, setting default 191
- error handling. *See* exception handling
- error notifier. *See* notifier
- error, start-up 216
- ErrorDiffusion 313
- event
 - methods 271
- event-driven controller
 - See* controller
- exception
 - cleaning up 100
 - creating 96
 - flow of control 98
 - handling 92–100, 210–213
 - setting parameters 97
- Exception class 93, 96
- execution
 - error 210
 - stack 171
 - tracing the flow 208
- executor, defined 319
- exiting the system 4
 - emergency 219
 - without Launcher 217
- expression 36
 - executing 109, 123
 - syntax 392

External Database Interface

classes, defined 188

defined 187

F

failure code, for user-defined primitive 418

false 28

fax support xxv

file

working with 105–108

See also File List

File Editor, defined 113

File List

commands 107–109

contents view commands 109

default pathname 107

defined 105

display options 106

displaying disk volumes 107

names view commands 108

pattern view commands 107

views 106

FileBrowser class 107

FillingWrapper class, defined 288

finalization 316–322

example 320

Finder

adding resource to a class 132

browsing code 132

defined

editing a resource 132

filtering classes 131

removing a resource 132

starting the main interface 132

updating classes 132

flashing, in animation 299

Float class 57

creating an instance 57

floating point numbers 57–59

arithmetic functions supported 57–58

comparing 58

converting 58

printing 58

test messages 58

fonts xix–xx

for loop. *See* number looping

formatting conventions 44

Fraction class 59

creating an instance 59

fractions 59

full block, defined 405

G

garbage collectors 335–341

compacting, defined 338

global 338

incremental

defined 336

phases of operation 337

generation scavenger, defined 336

geometrics

arcs, circles, and wedges 291

displaying 288

lines and polygons 290

splines and bezier curves 291

See also graphics

global variable 34

graphic object

defined 289

integrating into application 300–302

static vs. dynamic 300

See also graphics

graphics

animation 299

attributes, storing 288

coordinate system 274

- display surface, types of 280
- displaying geometrics 288
- image 293
- operations 273–302

GraphicsContext class

- as transient entity 284
- clipping 285
- default font 287
- default paint 286
- defined 284
- line characteristics 286
- translating displayed objects 285

H

halt message 177

HandleRegistry class

- defined 319
- hierarchy 319

hierarchy

- collection classes 78–79
- displaying 119
- numeric classes 67
- of images 294
- of paints 303
- of palettes 307
- signals 94–95, 212
- string classes 85–86
- weak arrays 319

hierarchy of objects 10

HSB color 305

hue, defined 306

I

icon, defined 299

ID property, defined 142

IdentityDictionary class, defined 72

IdentitySet class, defined 70

if statements 47

illustrations, creating 135

image

- as graphic object 293
- bit processing 297
- capturing 295
- combining with a mask 297
- creating 332
- magnifying and shrinking 297
- packed rows 296
- palette 296
- palette vs. performance 309
- processing 296
- restarting and reconnecting to database 214
- saving 3, 215
- saving when connected to database 213

Image class

- creating an instance 295
- defined 293
- hierarchy 294

Image Editor, defined 135

Image Maker 169–170

implementation limits 401–410

informational message, displaying 125

inheritance 16–17, 18

inherited method, overriding 18

inspector 127, 175, 181

- defined 130
- opening 123, 130

instance

- defined 13
- method 117

instance method 14

instance variable 31

- mapping to database table 158

Integer class

- creating an instance 55
- defined 55

integers 55–57

arithmetic functions supported 55–57
 comparing 57
 converting 57
 printing 57
 test messages 56
 Interval class 73
 creating an instance 73
 defined 71
 See also collection
 isControlActive 269
 isControlWanted 267
 iterative operations 48
 See also looping

K

keyboard shortcuts 411–412
 keyword message 39

L

Launcher, restarting 217
 lexical constructs, syntax 390
 limitations
 blocks 403–407
 non-overrideable methods 407
 size 401
 line
 displaying 290
 setting characteristics 286
 LinkedList class 73
 adding elements 76
 defined 71
 removing elements 76
 See also collection
 literal 24
 array 27
 character 26
 number 24

 string 26
 symbol 27
 lookup 17
 looping 47–52, 77
 types of 48
 low-space notifier 217

M

Magnitude class 67
 defined 85
 mail
 electronic xxv
 main interface, starting 132
 main window 103
 opening 217
 Mask class
 coverage 283
 defined 282
 mask value, defined 309
 memory layout 327–335
 object memory 333
 OldSpace, defined 333
 OE memory 328–333
 CompiledCodeCache, defined 329
 LargeSpace, defined 331
 NewSpace, defined 330
 PermSpace, defined 332
 StackSpace, defined 329
 old remembered table 335
 remembered table 334
 memory management 327–341
 memory policy 339
 memory reclamation
 compacting garbage collector 338
 data compactor, defined 339
 facilities 335–341
 generation scavenger, defined 336
 global garbage collector 338

- incremental garbage collector, defined 336
- MemoryPolicy class, defined 340
- menu
 - commands 137–140
 - defining in terms of a query 158
- Menu Editor 135
 - clearing text area 137
 - defined 135
 - enhanced 135, 136
 - commands 137–140
 - properties 139
 - exiting 137
 - standard 135
 - See also* menu
- message 12
 - binary 37
 - cascade 42
 - in sequence 40
 - keyword 39
 - types 37
 - unary 37
- message category
 - defined 12
- message expression 36
- message, defined
 - See also* informational messages, method
- method 11–18
 - class method 14
 - defined 9, 11–15
 - displaying method selectors 122
 - filing description 122
 - grouping 12
 - instance method 14
 - moving 122
 - non-overrideable 407
 - opening a browser 122, 124
 - overriding 18
 - printing description 122
 - removing 122

- syntax 394
- method lookup 14
 - defined 17
- method view 116
 - commands 122, 168
 - defined 116
- mouse buttons xx
 - <Operate> button xxi
 - <Select> button xxi
 - <Window> button xxi
 - one-button mouse xxi
 - three-button mouse xxi
 - two-button mouse xxi
- mouse operations xxii
 - <Control>-click xxii
 - <Meta>-click xxii
 - <Shift>-click xxii
 - click xxii
 - double-click xxii
- MVC 20, 257

N

- named input binding 196
- names view 106
- naming conventions 12, 13, 23
- nil 28
- notational conventions xix–xx
- notification properties 146–147
 - defined 146
- notifier 125, 171, 177
 - defined 126
 - See also* low-space notifier
- Number class, defined 67
- number literal 24
- number looping 49
- number syntax 389
- numeric classes 55–67
 - categories 67

- hierarchy 67
- O**
- object
 - behavior, defined 11
 - composite 9
 - examining variable values 130
 - hierarchy 10
 - state, defined 11
- Object class 19
- object table 334
- ObjectMemory class, defined 339
- object-oriented programming 7
 - overview 7–21
- OE, registering long-lived objects 436
- online documentation. *See* VisualWorks documentation
- open system 367
- open-coded block, defined 402
- operation
 - canceling 123
 - undoing 123
- OrderedCollection class
 - adding elements 75–76
 - defined 71
 - finding elements 77
 - removing elements 75–76
 - See also* collection
- OrderedDither 313
- output template 202
 - defined 202
 - reusing 204
 - skipping a variable 202
- P**
- packed row, in an image 296
- paint
 - color 304
 - coverage 304
 - defined 286
- Paint class 303
 - hierarchy 303
- paint policy
 - default 312
 - defined 311–313
- painter 1–2
- palette
 - adding multiple components of one type 134
 - color 308
 - color, 8-bit 309
 - conversion 309
 - coverage 307
 - creating 308
 - defined 133, 307
 - effect on performance 309
 - fixed, creating 308
 - mapped, creating 308
 - opening 133
- Palette class
 - defined 307
 - hierarchy 307
- parameter 195
 - binding NULL 196
 - binding to a name 196
 - defined 194
- Parcel Browser 163–169
 - structure 164
 - views 164
- Parcel List 161–163
- Parcel menu
 - commands 162
- Parser class
 - creating an instance 324
 - defined 323
- password, securing 190
- pattern
 - shifting the tile phase 304

- view 106
 - See also* tile
- Pattern class, defined 303
- performance tuning 204, 406–410
- PermSpace memory, putting code in 332
- pixel, defined 274
- Pixmap class
 - defined 281
 - depth and palette 282
- placeholder. *See* parameter
- Point class
 - arithmetic functions supported 276
 - creating an instance 276
 - specifying polar coordinates 276
- pointer 315
- polling controller
 - See* controller
- polygon, displaying 290
- pool variable 33
- preferences, setting 104
- primitive
 - arguments 417
 - C to Smalltalk conversion 421–423
 - coded failure 427
 - defining 414
 - failure code 418
 - failure return 426
 - general advice 419
 - numbering 417
 - object allocation 431
 - registering long-lived objects 436
 - Smalltalk to C conversion 423–425
 - success return 426
 - support routines 435
 - type checking 427
 - unsafe 439
 - user-defined 413–442
 - See also* user-defined primitive
- primitive numbers 417
- priority level
 - defined 89
 - setting 89–90
- proceedability attribute 96
- process 87–100
 - coordinating 90–92
 - creating 87
 - postponing 92
 - running multiple 87–88
 - scheduling 88
 - setting the priority level 89–90
 - sharing data 92
 - states of 91
 - terminating 88
- Processor, defined 88
- project
 - closing 180
 - creating 179
 - entering 179
 - exiting 179
 - managing 179–185
 - moving changes between images 180
 - nesting 180
 - summarizing changes 180
 - See also* Change List, Change Set
- Project tool 179
 - defined 113
 - See also* project
- properties
 - applying 141
 - editing 141
 - See also* widget, Properties Tool
- Properties Tool
 - defined 140
 - See also* widget, properties
- protocol
 - adding 121
 - defined 12
 - editing 121

- filing method description 121
- finding methods of a class 121
- list of 385–386
- opening a browser 121
- printing description of methods 121
- removing 121
- renaming 121

protocol view

- commands 165, 167

Protocols 116

Q

query

- allocating adaptors 200
- allocating buffers 200
- asynchronous execution 199
- cancelling answer set 205
- catalog 206
- checking execution status 199
- defining 158
- describing an answer set 200
- executing 195
- getting an answer 197, 198
- handling multiple answer sets 198
- parameters 194
- processing an answer stream 201
- raising an exception 197
- testing row count 199
- using an output template 202
- viewing results 193

Query Editor

- menu queries 158
- SQL Editor 159

query variable. *See* parameter

R

raising an exception 96

Random class 59

- creating an instance 59–60

random numbers 59–60

- generating 59

RasterOp class, use in image processing 297

rectangle

- creating 277–279

- displaying 290

- inquiries and transformations 279

Rectangle class

- creating an instance 277

- defined 277

redisplaying a view ??–262

remembered table, defined 334

resource

- adding to a class 132

- editing 132

- removing 132

Resource Finder. *See* Finder

resource, releasing 207

retained medium, defined 281

reuse methodologies 2

reverting to a prior version 184

RGB color 305

RunArray class, defined 70

S

saturation, defined 307

saving an image 3

Scanner class

- creating an instance 323

- defined 323

scavenge threshold, defined 330

ScheduledControllers 267

ScheduledWindow class, defined 281

screen conventions xx

seed, defined 59

selector 12

- self 34
- Semaphore class, defined 90–91
- SequenceableCollection class 85
 - defined 79
- session
 - defined 193
 - disconnecting 205, 208
 - reconnecting 208
 - See also* query
- Set class, defined 70
- Settings Tool, defined 104
- SharedQueue class, defined 92
- shift value, defined 309
- signal
 - choosing 94
 - creating 94
 - global 94
 - hierarchy 94–95
 - nested 99–100
 - public 95–96
 - restricted in primitives 439
- Signal class 93
- signal constants protocol 210, 212
- size limitations 401
- snapshot, making 3
- SortedCollection class
 - adding elements 76
 - creating an instance 73
 - defined 71
 - finding elements 77
 - removing elements 76
 - See also* collection
- sources file 217
- special characters 395–400
- special symbols xix–xx
- SPIM graphic model 273
- spline, defined 291
- SQL Editor 159
- SQL query, executing 188, 193, 207
- stack 171
- stack view 127
- starting the system 2
- start-up errors 216
- state error 210
- state, defined 11
- string
 - as graphic object 289
 - creating 83
 - evaluating as Smalltalk expression 325
 - operations 81–86
 - working with 83–85
 - See also* character
- String class 81
 - defined 86
 - hierarchy 85–86
- string literal 26
- StrokingWrapper class, defined 288
- strong pointer, defined 315
- subcategory 15
- subviewWantingControl message 267
- super 34
- superclass 19
- support, technical
 - electronic bulletin boards xxv
 - electronic mail xxv
 - fax xxv
 - telephone xxv
 - World Wide Web xxv
- switches 117
 - types of 111
- symbol 27
- symbols used in documentation xix–xx
- syntax 23
 - atomic terms 391
 - Character class 388
 - expressions and statements 392
 - fixed-point numbers 25
 - floating-point numbers 25

- formal description 387–394
 - integers 25
 - lexical constructs 390
 - methods 394
 - nondecimal numbers 25
 - numbers 24, 389
 - scientific notation 26
 - special characters 387
 - System Browser 13, 16, 115–124
 - class category view 116
 - class view 116
 - class-instance switch 117
 - code view 116
 - defined 115
 - method view 116
 - protocol view 116
 - structure 115
 - views 115
 - system constant, defined 13
 - system failure, recovering from 215
 - System Transcript 125–126
 - clearing 176
 - displaying debug messages 176
 - sending output to 176
- T**
- technical support xxiv, 219
 - electronic mail xxv
 - electronic bulletin boards xxv
 - fax support xxv
 - telephone support xxv
 - World Wide Web xxv
 - telephone support xxv
 - temporary variable 29
 - tenure threshold, defined 331
 - text
 - as graphic object 289
 - copying 123
 - deleting 123
 - editing 113
 - pasting 123
 - replacing 109
 - restoring previous 123
 - saving 109
 - text object, types of 289
 - thisContext 34
 - tile phase, defined 304
 - tile, in a pattern 303
 - time 63–66
 - accessing information 65
 - arithmetic functions supported 64
 - comparing 64
 - converting 65
 - printing 65
 - Time class 63
 - creating an instance 64
 - methods 65
 - TimeZone class 66
 - tools
 - application building 131–155
 - Canvas Tool 132
 - Image Editor 135
 - Menu Editor 135
 - Palette 133
 - Resource Finder 131
 - application delivery
 - Image Maker 169–170
 - Parcel Browser 163–??
 - Parcel List 161–163
 - database 157–159
 - Canvas Composer 157
 - Canvas Tool 157
 - data forms 158
 - Data Modeler 157
 - Mapping Tool 158
 - Palette 157
 - Query Editor 158

- environment 103–114
 - Change List 109–113, 179
 - Change Set 179
 - File Editor 113
 - File List 105–109
 - Project 113, 179
 - Settings Tool 104
- programming 115–130
 - Debugger 126–129
 - Parcel Browser ??–169
 - System Browser 115–124
 - System Transcript 125–126
 - Workspace 124
- tracing 208
 - adding information 209
 - defined 208
 - disabling 209
 - setting trace level 209
 - specifying output location 208
 - using for troubleshooting 219
- tracing protocol 210
- transaction
 - controlling 207
 - coordinated 207
- Transcript class 126
- translation protocol 291
- translation, using GraphicsContext 285
- troubleshooting 215–220
- true 27
- type checking 427
- typographic conventions xix–xx

U

- UDP 413–442
- UDP interface
 - basic capabilities 414
 - example 440
- unary message 37

- unwind protection 100
- user interrupt 177
- user-defined primitive 413–442
 - C to Smalltalk conversion 421–423
 - datatypes 417
 - general advice 419
 - indexed access 432
 - initializing 434
 - installing and accessing 416
 - interrupts 437
 - poll handlers 437
 - signal restrictions 439
 - sizing 434
 - Smalltalk to C conversion 423–425
 - See also* primitive
- Utility menu
 - commands 163

V

- validation callback, defined 144
- validation properties, defined 144–145
- variable
 - assignment 29
 - class 32
 - defined 11–14
 - global 34
 - instance 31
 - pool 33
 - temporary 29
 - types 28
- version control 179–185

VI

- and memory manager 329
- view
 - as visual component 259
 - composite 263
 - redisplaying ??–262
 - without a controller 261

- views
 - Change List window 110
 - code view 127
 - Debugger window 126–129
 - File List browser 106
 - organizing 179
 - Parcel Browser 164
 - stack 127
 - System Browser 115
 - visual component
 - active vs. passive 258
 - autonomous vs. dependent 259
 - visual reuse, defined 2
 - VisualPart
 - role as window component 259
 - VisualWorks
 - application building 1–2, 131–155
 - connecting to database. *See* database
 - exiting 4
 - exiting problems 217–219
 - main window 103
 - starting 2
 - See also* tools
 - VisualWorks documentation
 - online xxiii
 - Database Cookbook* xxiii
 - Database Quick Start Guides* xxiii
 - International User's Guide* xxiii
 - VisualWorks Cookbook* xxiii
 - VisualWorks DLL and C Connect Reference* xxiii
 - printed xxii
 - Cookbook* xxii
 - Database Connect User's Guide* xxiii
 - Database Tools Tutorial and Cookbook* xxiii
 - Installation Guide* xxii
 - International User's Guide* xxiii
 - Object Reference* xxiii
 - Release Notes* xxii
 - Tutorial* xxii
- ## W
- weak array 315–322
 - finalization 316–322
 - finalization example 320
 - WeakArray class, defined 315
 - WeakDictionary class
 - defined 319
 - hierarchy 319
 - wedge, defined 292
 - while loop 48
 - widget
 - changing focus 143
 - color properties, defined 147
 - confirming an action has completed 146
 - creating a menu 135
 - details properties 143
 - disabling 143
 - hiding 144
 - position properties 149–150
 - properties
 - basic 142
 - color 147
 - details 143–144
 - drop source 150
 - drop target 151–153
 - notification 146
 - position (bounded widgets) 149, 150
 - validation 144
 - specifying appearance 143, 147
 - validating actions 144
 - wildcard symbols 85
 - window
 - as container 258
 - Window class, defined 280–281
 - WordArray class, defined 71

Index

Workspace, defined 124
World Wide Web xxv
wrapper
 defined 264