Apple Dylan Extensions and Framework Reference

Preliminary

Developer Press
Apple Computer, Inc.

Apple Computer, Inc.
© 1993–1995 Apple Computer, Inc.
All rights reserved.

No part of this publication or the software described in it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

Printed in the United States of America.

The Apple logo is a trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, APDA, HyperCard, LaserWriter, MacApp, Macintosh, and MPW are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Geneva and Monaco are trademarks of Apple Computer, Inc.

America Online is a registered service mark of America Online, Inc.

Adobe Illustrator, Adobe Photoshop, and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

CompuServe is a registered service mark of CompuServe, Inc.

Docutek is a trademark of Xerox Corporation.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

Internet is a registered trademark of Digital Equipment Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

PowerPC is a trademark of International Business Machines Corporation, used under license therefrom.

Mercutio MDEF from Digital Alchemy
Copyright ©Ramon M. Felciano 1992–1995, All Rights Reserved

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings xxi

Preface About This Book xxxiii

Chapter Organization xxxiii
Conventions Used in This Book xxxiv
Special Fonts xxxiv
Types of Notes xxxiv
Numerical Formats xxxiv
For More Information xxxv

Part 1 Language Extensions Reference 1

Chapter 1 Introduction 3

Apple Dylan and C-Compatible Libraries 5
Interface Importation 6
Cross-Language Calls 7

Chapter 2 Interface Definitions 9

About Interface Definitions 11
Interface Importation 12
The Interface Importation Process 13
Importing C Definitions 14
Importing C Containers 16
Name Mapping 16
Type Mapping 17
Interface Definitions 18
File Clauses 19

Other Clauses 21
Using Interface Definitions 23
Using File Clauses 23
Importing C Header Files 23
Controlling Conditional Processing 24
Selectively Importing or Excluding C Definitions 26
Controlling Dynamism 27
Using Other Types of Clauses 31
Using Structure and Union Clauses 31
Using Function Clauses 34
Using Variable Clauses 35
Using Constant Clauses 37
Using Callout and Callback Clauses 38
Interface Definition Reference 39
Interface Definition Syntax 39
Interface Definition Clauses 39
File Options 42
Container Options 43
Constant Options 46
Variable Options 47
Function Options 49
Callback Options 51

Chapter 3 Name Mapping 53

About Name Mapping 55
Using Name Mapping 56
Specifying Explicit Name Mapping 56
Using the Import Option of a Container Clause 56
Using the Rename Option of a Container Clause 57
Using the Arrow Option of a Clause 58
Using the Getter and Setter Options of a Variable Clause 60
Adding Prefixes to Imported Names 60
Using the Name-Mapping Functions 62
Name Mapping Reference 64

Chapter 4 Cross-Language Calls 69

About Cross-Language Calls 71
Types of Cross-Language Calls 71
Overview of a Cross-Language Call 72
Using Cross-Language Calls 74
Calling Imported Functions 75
Calling C Library Functions Using Function Pointers 77
Calling Apple Dylan Functions From C Libraries 79

Chapter 5 Type Mapping 85

About Type Mapping 87
About Importing Type Definitions 87
About Type Mappings 89
Implicit Type Mapping 90
By-Value Types 91
Untyped-Pointer Types 92
Statically-Typed-Pointer Types 93
Explicit Type Mappings 93
About Value Translation 94
Using Type Mapping 95
Importing Structures 95
Arrays as Structure Members 97
Importing Pointer-To-Structure Types 98
Importing Pointer-To-Function Types 99
Using Explicit Type Mapping 100
Type Mapping Strings 100
Importing and Exporting Values 101
Type Mapping Reference 105
Pointer Classes 105
Structure-Related Functions 111
Functions for Importing and Exporting Values 113

Chapter 6 Access Paths 119

About Access Paths 121
Using Access Paths 122
Using File Options to Specify Information About Access Paths 122
Using the Inline Machine Code Access Path 124
Using The Code Fragment Manager Path 126
Creating a CFM Shared Library 126
Using CFM Shared Libraries 128
Using The Apple Shared Library Manager Access Path 129
Creating an ASLM Shared Library 131
Using ASLM Shared Libraries 132
Using The External Module Access Path 133
Creating a Custom Application Nub 134
Using Statically Linked External Modules 135
Using The Direct Pointer Access Path 135
Preparing a Code Resource 136
Loading and Unloading a Code Resource 138
Calling Functions Imported from a Code Resource 140

Chapter 7 Low-Level Facilities 143

About Low-Level Facilities 145
Using Low-Level Facilities 145
Using Machine Pointers Directly 146
Allocating Stack Storage 148
Making Low-Level Function Calls 150
Calling Dylan Methods From Other Languages 151
Using Preemption and Event Checking 151
Low-Level Facilities Reference 152
Reading and Writing Data Using Machine Pointers 152
Reading and Writing A Single Byte 153
Reading and Writing Pairs of Bytes 158
Reading and Writing Long Words 162
Reading and Writing Strings 168
Low-Level Macros for Allocating Stack Storage 171
Low-Level Functions for Cross-Language Function Calls 174

Preemption Facilities Reference 178
Preemption Constants 178
Preemption Functions 178
Preemption Macros 181
Event-Checking Facilities Reference 183
Break-Request Variables and Functions 183
Event-Checking Functions 185
Microsecond Functions 187

Chapter 8 Garbage Collection 191

About Garbage Collection 193
Overview of Apple Dylan Garbage Collection 193
Ephemeral Garbage Collection 194
Garbage-Collecting Algorithms 195
Large-Object Garbage Collection 196
Using Garbage Collection 197
Using Spontaneous Garbage Collection 197
Requesting Garbage Collection 197
Controlling Garbage Collection Settings 198
Debugging 199
Error Handling 200
Garbage Collection Reference 201
Constants and Variables 201
Functions 202

Chapter 9 Termination 205

About Termination 207
Termination and Garbage Collection 207
Termination Processing 208
Using Termination 209
Enabling Termination For an Object 209
Termination Processing 209
Specifying Termination Behavior 210
Resurrecting an Object 210

Termination Reference 211
Variables 211
Functions 211

Part 2 Framework Reference 215

Chapter 10 Introduction 217

Conceptual Model 219
Framework Foundation 220
Event Handling 223
User Interface Support 228
Documents and Data Interchange 236

Chapter 11 Event Handling 239

About Event Handling 242
Kinds of Events 242
Event Handlers 246
The Target Chain 248
Responding to Events 250
Event Handling Using the Target Chain 251
Mouse Event Handling 252
Changing the Target 254
Behaviors and Event Handling 254
Using Event Handling Objects 259
Adding an Event Handler to the Target Chain 259
Defining a do-event Method 260
Implementing a Behavior 260
Specifying Actions When the Target Changes 263
Event Handling Objects Reference 263
The <event-handler> Class 263
The <main-handler> Class 265
The <behavior> Class 266

The <event> Class 267
Toolbox Event Classes 268
The <disk-event> Class 270
Key Event Classes 270
System Event Classes 271
Window Event Classes 271
Mouse Event Classes 273
The <become-target-event> Class 274
The <resign-target-event> Class 275
Functions for Event Handling 276

Chapter 12 Menus 281

About Menus 283
Menus and Menu Items 283
Menu Event Handling 284
Menu Classes 285
Using Menu-Related Objects 287
Creating Menus and Menu Items 287
Setting Up Menus 288
Responding to Menu Events 290
Setting up the About Menu Item 292
Menu Class Reference 292
The <menu-element> Class 292
The <menu-item> Class 294
The <separator-item> Class 297
The <menu> Class 297
The <apple-menu> Class 300
The <menu-event> Class 301
Functions for Menus 302

Chapter 13 Graphics 305

About Graphics 307
About Graphics Ports and Buffers 309
About Graphics Caches 310

About Frames 310
Frame Coordinate Systems 313
About Regions 314
About Text Styles 315
About Color 316
Using Graphics Objects 317
Changing an Object's Location and Extent 317
Manipulating Regions and Rectangles 317
Defining Text Styles 318
Defining Colors 318
Graphics Objects Reference 319
The <frame> Class 319
Graphics Ports Classes 320
Text Style Classes 322
The QuickDraw Region Class 324
RGB Color Class 324
Graphics-Related Functions 325

Chapter 14 Views 333

About Views and Supporting Classes 335
View Hierarchies 337
Adorners 339
View Determiners 341
Drawing in Views 342
Highlighting 343
Cursor Manipulation 344
Using View Objects 345
Defining a View Class 345
Setting Up Views 346
Setting the Target View 347
Drawing in Views 348
Setting Up a View for a Graphics World 348
Drawing With Buffered Views 349
Setting Up Adorners 350
Implementing a View Determiner 352

View Objects Reference 354
The <view> Class 355
Adorner Classes 360
View Determiner Classes 362
View-Related Functions 364
Functions for Graphics Support 367
Functions for Adorners 368
Functions for View Determiners 369

Chapter 15 Text Views 371

About Text View Classes 373
Using Text View Objects 374
Creating a Text View 374
Getting and Setting Font Styles 374
Growing a Text View 375
Text View Objects Reference 375
The <text-view> Class 375
The <edit-text> Class 378
The <number-text> Class 379
Text View Functions 381

Chapter 16 Grid and List Views 385

About Grid and List Views 387
Grid and List View Classes 387
Grid View Behaviors 388
Using Grid and List View Objects 388
Setting Up Grid and List Views 389
Drawing in Views 389
Selecting Cells 390
Highlighting Cells in a View 390
Handling Variable Column Widths 390
Handling Variable Row Heights 391
Grid and List View Objects Reference 391
The <grid-view> Class 391

The <text-grid-view> Class 394
The <list-view> Class 394
The <text-list-view> Class 395
The <grid-select-behavior> Class 396
The <grid-arrow-key-behavior> Class 396
Grid and List View Related Functions 397

Chapter 17 Windows 399

About Windows 401
Using Window Objects 402
Creating a Window 402
Creating a Window for a Dialog Box 404
Creating a Floating Window 404
Window Objects Reference 406
The <window-context> Class 406
The <window> Class 407
The Grow Icon Classes 413
The <grow-icon> Class 414
The <grow-icon-determiner> Class 414
Functions for Windows 415

Chapter 18 Dialogs and Controls 417

About Dialogs and Controls 419
Dialog Behaviors 419
Control View Classes 421
Control Events 423
Highlighting in Controls 424
Using Dialogs and Controls 424
Creating a Modal Dialog Box 425
Creating a Window for a Modal Dialog Box 429
Setting up Dialog Box Dismissers 429
Creating Static Text 430
Creating Number Text 430
Implementing a Popup Menu 431

Adding a Tabbing Behavior 432
Opening the Modal Dialog Box 432
Dialog and Control Objects Reference 433
The <dialog-behavior> Class 433
The <tabber> Class 435
The <edit-text-dialog-filter> Class 436
The <cluster> Class 437
The <simple-icon> Class 438
The <static-text> Class 439
The <control> Class 440
Subclasses of the <control> Class 442
The <button> Class 443
The <check-box> Class 443
The <radio> Class 443
The <scroll-bar> Class 444
The <popup> Class 445
Control Manager Control Classes 446
The <ctl-mgr-control> Class 446
The <ctl-mgr-button> Class 448
The <ctl-mgr-check-box> Class 449
The <ctl-mgr-radio> Class 450
The <ctl-mgr-scroll-bar> Class 451
The <ctl-mgr-popup> Class 452
Control Event Classes 454
The <default-button-adorner> Class 455
Functions for Dialogs and Controls 456

Chapter 19 Scrolling and Tracking 459

About Scrolling 461
About Tracking 463
Using Scrolling and Tracking 465
Setting up a Scrolling View 465
Setting up a Scroller View 466
Setting up Scroll Bars 467
Implementing Mouse Tracking 467

Scrolling and Tracking Objects Reference 471
The <scroller> Class 471
The <tracker> Class 473
The <tracker-adorner> Class 474
Functions for Scrolling and Tracking 475

Chapter 20 Documents 477

About Documents 479
The Open Protocol 479
Commands and Change Counts 480
Document Menu Items 480
Using Documents 481
Defining a Document Subclass 481
Registering a Document 482
Creating and Opening a Document 482
Creating a Document's Windows 483
Specifying a Document's File Type 484
Document Objects Reference 484
The <document> Class 485
Functions for Documents 487

Chapter 21 Files 489

About Files 491
Manipulating Files 491
Manipulating Resources 492
Documents and Files 492
Using Files 493
Reading From a File 493
Writing To a File 494
Reading From a Resource Fork 495
Updating a Resource Fork 495
File Objects Reference 496
The <file> Class 496
File Functions 500

Chapter 22 Streams 503

About Streams 505
About Class and Slot Specifications 507
Using Class and Slot Specifications 507
Setting up the Class and Slot Specifications 508
Using Slot Specification Getters and Setters 509
Cloning Objects 511
Using Shared and Default Objects 512
Using Streams 513
Using Handle Streams 513
Using Object Streams 514
Reading and Writing Objects 515
Reading and Writing Primitive Data Types 516
Stream Objects Reference 517
The <stream> Class 517
The <file-stream> Class 518
The <handle-stream> Class 519
Object Stream Classes 521
The <class-spec> Class 522
The <slot-spec> Class 523
Functions for Streams 525

Chapter 23 Commands 529

About Commands 531
Using Commands 534
Defining a Command Class 534
Doing a Command 535
Undoing and Redoing a Command 535
Committing a Command 536
Completing a Command 537
Performing a Command 537
Command Objects Reference 539
The <command-context> Class 539
The <command> Class 540
Text Command Classes 541

The <typing-command> Class 542
The <text-style-command> Class 544
The Font Command Classes 545
Functions for Commands and Command Contexts 547

Chapter 24 Data Interchange 549

About Data Interchange 551
Using Data Interchange Classes 554
Using Typed Data 554
Using Typed Handles 554
Using Typed Pointers 555
Using Data Items 555
Using Designators 556
Creating Designator Objects 557
Defining Designator Subclasses 557
Data Interchange Objects Reference 558
The <typed-data> Class 558
The <typed-pointer> Class 559
The <typed-handle> Class 560
The <data-item> Class 560
Designator Classes 561
The <designator> class 561
The <selection-designator> class 562
The <text-designator> class 563
The <character-designator> class 563
The <range-designator> Class 564
The <offset-designator> Class 564
Functions for Data Interchange 566

Chapter 25 Drag and Drop 569

About Drag and Drop 571
Using Drag and Drop Classes 572
Setting up a Drag 572
Setting up a Promise Function 574

Accepting a Drag 575
Receiving a Drop 576
Drag and Drop Objects Reference 576
The <drag> Class 577
The <drag-item> Class 578
The <drag-flavor> Class 579
Drag Event Classes 580
Functions for Drag and Drop 582

Chapter 26 Clipboard 585

About the Clipboard 587
Using the Clipboard 588
Setting Up the Clipboard Behavior 589
Specifying the Kind of Data to Paste 589
Setting up a Designator for Selected Data 590
Retrieving Data Specified by a Designator 591
Associating Clipboard Data with a View 591
Specifying the Selection 593
Specifying How to Delete Specified Data 593
Clipboard Objects Reference 594
The <clipboard-behavior> Class 594
The <clipboard> Class 594
The <clipboard-flavor> Class 595
Clipboard Command Classes 596
Functions for Clipboard Support 598

Chapter 27 Scripting Support 599

About Scripting Support 601
Apple Events 602
Data Structures 602
Properties, Object Specifiers, and Descriptors 603
Using Scripting Support Classes 604
Getting and Setting Property Values 604
Responding to Apple Events 605

Scripting Support Objects Reference 607
The Apple Event Classes 607
The <ae-desc> class 607
The <ae-list> class 608
The <ae-record> class 609
The <apple-event> class 610
The <object-specifier> class 610
Descriptor Classes 611
The <token> class 612
Scripting Event Classes 612
Property Designator Classes 615
The <property> Class 616
The <document-property> Class 616
The <file-property> Class 617
The <text-view-property> Class 617
The <text-view-selection-property> Class 618
The <window-property> Class 618
Scripting Support Functions 619

Chapter 28 Environment and Utilities 625

About Environment and Utility Classes 627
About Region Classes 628
About String Classes 628
About Condition Classes 628
About the Busy Cursor Class 629
About the Framework Library Class 629
About External Reference Tables 630
Using Environment and Utility Classes 630
Using Rectangles 630
Using String Classes 631
Using Handle Strings 631
Using Resource Strings 631
Using Condition Classes 632
Using Framework Error Conditions 632
Using OS Error Conditions 632

Using the Busy Cursor Class 633
Using the Framework Library Class 633
Using the External Reference Table Class 634
Geometry Objects Reference 635
The <region> Class 635
The <rect> Class 635
Geometry Functions 637
String Objects Reference 639
The <handle-string> Class 639
The <resource-string> Class 640
String Resource Functions 641
Condition Objects Reference 642
The <framework-error> Class 642
The <os-error> Class 643
Error Condition Functions 644
Debugger Behavior Reference 645
The <debugger-behavior> Class 645
Busy Cursor Reference 646
The <busy-cursor> Class 646
Busy Cursor Functions 648
Library Objects Reference 649
The <framework-library> Class 649
Framework Library Functions 651
External Reference Table Reference 651
The <external-reference-table> Class 652
External Reference Functions 653
Miscellaneous Function Reference 654

Index 657

Figures, Tables, and Listings

Preface About This Book xxxv

Chapter 1 Introduction 3

Chapter 2 Interface Definitions 9

Chapter 3 Name Mapping 53

Chapter 4 Cross-Language Calls 69

Listing 4-1 Header file with function pointer type and functions 77
Listing 4-2 A header file using a callback function 79

Chapter 5 Type Mapping 85

Chapter 6 Access Paths 119

Listing 6-1 A sample rez file for a CFM shared library 127
Listing 6-2 A sample ASLM header file 130
Listing 6-3 A sample rez file for an ASLM shared library 132
Listing 6-4 An assembly language transfer vector 137
Listing 6-5 Header file for functions imported from a code resource 140

Chapter 7 Low-Level Facilities 143

Listing 7-1 Using the `with-stack-block` macro to allocate stack storage 149

Chapter 8 Garbage Collection 191

Chapter 9 Termination 205

Chapter 10 Introduction 217

Figure 10-1 Framework subsystems 219
Table 10-1 Files, resources, and streams 221
Table 10-2 Error reporting 222
Table 10-3 Geometry 223
Table 10-4 General event-handling and dispatching classes 223
Table 10-5 Toolbox events 224
Table 10-6 Object model support 226
Table 10-7 Designators 228
Table 10-8 Graphics 229
Table 10-9 Windows and views 230
Table 10-10 Text views 231
Table 10-11 Grid views 232
Table 10-12 Dialogs and controls 232
Table 10-13 Menus 234
Table 10-14 Adorners 235
Table 10-15 Context undo and redo 235
Table 10-16 Documents 236
Table 10-17 Data interchange 237
Table 10-18 Clipboard 237
Table 10-19 Drag and drop 238

Chapter 11 Event Handling 239

Figure 11-1 Event class hierarchy 243
Figure 11-2 Mouse event class hierarchy 245
Figure 11-3 Event handler class hierarchy 246
Figure 11-4 Target chain for a document with one window and content view 248
Figure 11-5 An application with several event-handling chains 250
Figure 11-6 A mouse down event in a view hierarchy 253
Figure 11-7 Target chain with behaviors 256

Figure 11-8 Behavior class hierarchy 257
Listing 11-1 Specifying the next handler for a window 259
Listing 11-2 Defining a `do-event` method 260
Listing 11-3 Defining a behavior subclass 261
Listing 11-4 Defining a behavior's response to an event 261
Listing 11-5 Handling the cursor from a behavior 262
Listing 11-6 Adding and removing a behavior 262
Listing 11-7 The `<event-handler>` class definition 263
Listing 11-8 The `<main-handler>` class definition 265
Listing 11-9 The `<behavior>` class definition 267
Listing 11-10 The `<event>` class definition 267
Listing 11-11 The `<toolbox-event>` class definition 268
Listing 11-12 The `<disk-event>` class definition 270
Listing 11-13 Class definitions for key events 270
Listing 11-14 Class definitions for system events 271
Listing 11-15 The `<window-event>` class definition 272
Listing 11-16 The `<activate-event>` class definition 272
Listing 11-17 The `<update-event>` class definition 273
Listing 11-18 Class definitions for mouse events 273
Listing 11-19 The `<become-target-event>` class definition 274
Listing 11-20 The `<resign-target-event>` class definition 275
Table 11-1 General event-handling functions 276
Table 11-2 Event-handling functions for behaviors 278
Table 11-3 Event-handling functions for dispatching 279
Table 11-4 Event-handling functions for streaming 279

Chapter 12 Menus 281

Figure 12-1 Menu-related classes 285
Listing 12-1 Making a menu and menu items 287
Listing 12-2 Specifying an event object for a menu item 288
Listing 12-3 Setting up menus for a behavior 289
Listing 12-4 Setting up menus for an event handler 289
Listing 12-5 Specializing the `do-marking` method's parameters 290
Listing 12-6 An event handler's response to a menu event 291
Listing 12-7 An event handler's response to a subclassed menu event 291
Listing 12-8 A behavior's response to a menu event 291
Listing 12-9 The `<menu-element>` class definition 293
Listing 12-10 The `<menu-item>` class definition 295
Listing 12-11 The `<separator-item>` class definition 297

Listing 12-12 The <menu> class definition 298
Listing 12-13 The <apple-menu> class definition 300
Listing 12-14 The <menu-event> class definition 301
Table 12-1 Functions for manipulating entire menus 302
Table 12-2 Functions for manipulating menu items 303
Table 12-3 Functions for handling menu events 303

Chapter 13 Graphics 305

Figure 13-1 Graphics mechanism 308
Figure 13-2 Graphic port and buffer classes 309
Figure 13-3 Frame classes 310
Figure 13-4 Location and extent for windows and root views 311
Figure 13-5 Translation between scroller views and its subviews. 312
Figure 13-6 Framework coordinate systems 314
Figure 13-7 Region classes 315
Figure 13-8 Text style classes 315
Table 13-1 QuickDraw text style constants 316
Table 13-2 RGB color constants 316
Listing 13-1 Changing a view's location and extent 317
Listing 13-2 Manipulating regions and rectangles 318
Listing 13-3 Defining a text style 318
Listing 13-4 Defining a color constant 319
Listing 13-5 The <frame> class 319
Listing 13-6 The <graphics-port> class definition 320
Listing 13-7 The <qd-graphics-port> class definition 320
Listing 13-8 The <qd-graphics-buffer> class definition 321
Listing 13-9 The <text-style> and <qd-text-style> classes 322
Listing 13-10 The <qd-region> class 324
Listing 13-11 The <rgb-color> class 325
Table 13-3 Functions for frames 326
Table 13-4 Functions for graphics ports 327
Table 13-5 Functions for graphics buffers 327
Table 13-6 Functions for QuickDraw regions 328
Table 13-7 Functions for QuickDraw text styles 330
Table 13-8 Functions for colors 331

Chapter 14 Views 333

Figure 14-1 View classes 336
Figure 14-2 Views for a simple dialog 338
Figure 14-3 The view hierarchy for a simple dialog 338
Figure 14-4 Views that support scrolling 339
Figure 14-5 The view hierarchy for a scrolling window 339
Figure 14-6 Adorner classes 340
Figure 14-7 View determiner classes 341
Table 14-1 Highlight state transitions and iteration orders 344
Listing 14-1 Defining a view class 345
Table 14-2 Cursors 345
Listing 14-2 Setting up view for a window 346
Listing 14-3 Defining a `draw` method 348
Listing 14-4 Defining a view class with a graphics buffer 348
Listing 14-5 Creating the buffer and view 349
Listing 14-6 Drawing into an offscreen view buffer 349
Listing 14-7 Drawing the contents of a buffer onscreen 350
Listing 14-8 Defining an adorner subclass 351
Listing 14-9 Defining an `adorner-draw` method 351
Listing 14-10 Defining `adorner-hilite` methods 351
Listing 14-11 Adding an adorner to a view 352
Listing 14-12 Setting up view determiners for scrolling 353
Listing 14-13 Defining a view determiner subclass 354
Listing 14-14 Defining a `super-view-extent-changed` method 354
Listing 14-15 The `<view>` class definition 355
Listing 14-16 Adorner class definitions 361
Listing 14-17 View determiner class definitions 362
Table 14-3 Functions for views 364
Table 14-4 Functions for graphics support 367
Table 14-5 Functions for adorners 368
Table 14-6 Functions for view determiners 369

Chapter 15 Text Views 371

Figure 15-1 Text View classes 373
Listing 15-1 Creating a `<text-view>` object 374
Listing 15-2 Determining and setting the style of selected text. 375
Listing 15-3 The `<text-view>` class definition 376
Listing 15-4 The `<edit-text>` class definition 378

Listing 15-5 The <number-text> class definition 380
Table 15-1 Functions for text views 381
Table 15-2 Functions for text styles 382

Chapter 16 Grid and List Views 385

Figure 16-1 Grid and list view classes 387
Figure 16-2 Setting up a grid view 389
Listing 16-1 The <grid-view> class definition 391
Listing 16-2 The <text-grid-view> class definition 394
Listing 16-3 The <list-view> class definition 395
Listing 16-4 The <text-list-view> class definition 395
Listing 16-5 The <grid-select-behavior> class definition 396
Listing 16-6 The <grid-arrow-key-behavior> class definition 397
Table 16-1 Functions for grid views 397
Table 16-2 Functions for list views 398

Chapter 17 Windows 399

Figure 17-1 The <window> and <window-context> classes 401
Listing 17-1 Creating a scrolling window 403
Listing 17-2 Creating a window 404
Listing 17-3 Creating a dialog box window 404
Listing 17-4 Creating a Floating Window 405
Listing 17-5 The <window-context> class definition 406
Listing 17-6 The <window> class definition 407
Listing 17-7 The <grow-icon> class definition 414
Listing 17-8 The <grow-icon-determiner> class definition 414
Table 17-1 Functions for window contexts 415
Table 17-2 Functions for windows 415

Chapter 18 Dialogs and Controls 417

Figure 18-1 The dialog behavior classes 420
Figure 18-2 Dialog-related view classes 421
Figure 18-3 The <control> class and its subclasses 422
Figure 18-4 The <control-event> class and its subclasses 423
Table 18-1 Control modes 424

Figure 18-5 Moveable modal dialog box 425
Listing 18-1 Creating a dialog box 425
Listing 18-2 Creating a moveable modal dialog box 429
Listing 18-3 Setting up dialog box dismissers 429
Listing 18-4 Creating static text 430
Listing 18-5 Creating number text 430
Listing 18-6 Implementing a popup menu 431
Listing 18-7 Determining the current choice ID 432
Listing 18-8 Adding tabbing behavior 432
Listing 18-9 Posing a Dialog Box 433
Listing 18-10 The `<dialog-behavior>` class definition 434
Listing 18-11 The `<tabber>` class definition 435
Listing 18-12 The `<edit-text-dialog-filter>` class definition 436
Listing 18-13 The `<cluster>` class definition 437
Listing 18-14 The `<simple-icon>` class definition 438
Listing 18-15 The `<static-text>` class definition 439
Listing 18-16 The `<control>` class definition 441
Listing 18-17 The `<button>` class definition 443
Listing 18-18 The `<check-box>` class definition 443
Listing 18-19 The `<radio>` class definition 444
Listing 18-20 The `<scroll-bar>` class definition 444
Listing 18-21 The `<popup>` class definition 445
Listing 18-22 The `<ctl-mgr-control>` class definition 446
Listing 18-23 The `<ctl-mgr-button>` class definition 449
Listing 18-24 The `<ctl-mgr-check-box>` class definition 450
Listing 18-25 The `<ctl-mgr-radio>` class definition 450
Listing 18-26 The `<ctl-mgr-scroll-bar>` class definition 451
Listing 18-27 The `<ctl-mgr-popup>` class definition 452
Listing 18-28 The `<control-event>` class and subclass definitions 454
Listing 18-29 The `<default-button-adorner>` class definition 455
Table 18-2 Functions for Control Manager controls 456
Table 18-3 Functions for Control Manager scroll bar controls 456
Table 18-4 Functions for Control Manager popup menu controls 457

Chapter 19 Scrolling and Tracking 459

Figure 19-1 The relationship between a scroller view, scroll bars, and a content view 461
Figure 19-2 Scrolling view hierarchy 462
Figure 19-3 The translation between a content view and a scroller 462

Figure 19-4 Mouse tracking feedback 464
Listing 19-1 Setting up a scrolling view 465
Listing 19-2 Setting up a scroller view 467
Listing 19-3 Initiating mouse tracking 468
Listing 19-4 Defining a `<tracker>` subclass 468
Listing 19-5 Beginning mouse tracking 469
Listing 19-6 Ending mouse tracking 470
Listing 19-7 Constraining tracking 470
Listing 19-8 Implementing feedback 471
Listing 19-9 The `<scroller>` class definition 472
Listing 19-10 The `<tracker>` class definition 473
Listing 19-11 The `<tracker-adorner>` class definition 474
Table 19-1 Functions for scrolling 475
Table 19-2 Functions for tracking 475

Chapter 20 Documents 477

Listing 20-1 Defining a document subclass 481
Listing 20-2 Specifying a document's data type 482
Listing 20-3 Creating a document 482
Listing 20-4 Creating a document's windows 483
Listing 20-5 Specifying a document's file type 484
Listing 20-6 The `<document>` class definition 485
Table 20-1 Functions for documents 487
Table 20-2 Functions for files associated with documents 488

Chapter 21 Files 489

Listing 21-1 Reading from a file 494
Listing 21-2 Writing to a file 494
Listing 21-3 Reading from a resource fork 495
Listing 21-4 Updating a resource fork 496
Listing 21-5 The `<file>` class definition 496
Table 21-1 Functions for files associated with documents 500
Table 21-2 Functions for resources 501
Table 21-3 Functions for files associated with documents 502

Chapter 22 Streams 503

Figure 22-1 Stream classes 506
Listing 22-1 The `make-class-spec` and `make-slot-spec` forms 508
Listing 22-2 Using non-default getters and setters 509
Listing 22-3 Specifying a slot getter function for streaming 510
Listing 22-4 Specifying a slot getter function for cloning 510
Listing 22-5 Preventing cloning 511
Listing 22-6 A `clone` function 511
Listing 22-7 Specifying a class for shareable objects without slots 512
Listing 22-8 Using a default object 512
Listing 22-9 Retrieving the contents of a handle 513
Listing 22-10 Reading a handle as a stream 514
Listing 22-11 Writing an object to a stream 515
Listing 22-12 Reading an object from a steam 515
Listing 22-13 A `read-from-stream` function 516
Listing 22-14 The `write-to-stream` function 517
Listing 22-15 The `<stream>` class definition 517
Listing 22-16 The `<file-stream>` class definition 518
Listing 22-17 The `<handle-stream>` class definition 519
Listing 22-18 Object stream class definition 521
Listing 22-19 The `<class-spec>` class definition 522
Listing 22-20 The `<slot-spec>` class definition 523
Table 22-1 Functions for streams 525
Table 22-2 Functions for file streams 526
Table 22-3 Functions for handle streams 527
Table 22-4 Functions for object streams 527
Table 22-5 Functions for class specifications 528

Chapter 23 Commands 529

Figure 23-1 Command classes 532
Figure 23-2 Command context classes 533
Listing 23-1 Defining a `<command>` subclass 535
Listing 23-2 Specializing the `doit` method 535
Listing 23-3 Specializing the `undoit` method 536
Listing 23-4 Specializing the `redoit` method 536
Listing 23-5 Specializing the `commit` method 536
Listing 23-6 Specializing the `completed` method 537
Listing 23-7 Creating and posting a command 538

Listing 23-8 The <command-context> class 539
Listing 23-9 The <command> class 540
Listing 23-10 The <typing-command> class 542
Listing 23-11 The <text-style-command> class 544
Listing 23-12 The <font-command>, <font-size-command>, and <font-style-command> classes 546
Table 23-1 Functions for command contexts 547
Table 23-2 Functions for commands 548
Table 23-3 Functions for typing commands 548

Chapter 24 Data Interchange 549

Figure 24-1 Typed data classes 552
Figure 24-2 Data item classes 552
Figure 24-3 Designator classes 553
Listing 24-1 Creating a typed handle 554
Listing 24-2 Creating a typed pointer 555
Listing 24-3 Creating a data item of allowable flavors in the clipboard 556
Listing 24-4 Creating a <range-designator> object 557
Listing 24-5 Using a <range-designator> object 557
Listing 24-6 Defining a <designator> subclass 558
Listing 24-7 Specializing the is-empty? method 558
Listing 24-8 The <typed-data> class 559
Listing 24-9 The <typed-pointer> class 559
Listing 24-10 The <typed-handle> class 560
Listing 24-11 The <data-item> class 561
Listing 24-12 The <designator> class 562
Listing 24-13 The <selection-designator> class 562
Listing 24-14 The <text-designator> class 563
Listing 24-15 The <character-designator> class 563
Listing 24-16 The <range-designator> class 564
Listing 24-17 The <offset-designator> class 565
Table 24-1 Functions for typed data 566
Table 24-2 Functions for typed handles 566
Table 24-3 Functions for data items 567
Table 24-4 Functions for designators 567
Table 24-5 Functions for designators 568

Chapter 25 Drag and Drop 569

Figure 25-1 Drag classes 571
Listing 25-1 Setting up a drag operation 573
Listing 25-2 A promise function 574
Listing 25-3 A `behavior-accept-drag?` method 575
Listing 25-4 A `behavior-receive-drop` method 576
Listing 25-5 The `<drag>` class 577
Listing 25-6 The `<drag-item>` class 578
Listing 25-7 The `<drag-flavor>` class 579
Listing 25-8 The `<drag-event>`, `<track-drag-event>`, and `<receive-drag-event>` classes 581
Table 25-1 Functions for drag and drop 582
Table 25-2 Functions that support drag and drop for views and windows 583

Chapter 26 Clipboard 585

Figure 26-1 Clipboard classes 587
Figure 26-2 Clipboard command classes 588
Listing 26-1 Implementing a clipboard behavior 589
Listing 26-2 The `can-receive-data?` method 590
Listing 26-3 Designating the current selection 590
Listing 26-4 Retrieving data associated with a designator 591
Listing 26-5 Associating data with a view 592
Listing 26-6 Setting the selection 593
Listing 26-7 Deleting data 593
Listing 26-8 The `<clipboard-behavior>` class 594
Listing 26-9 The `<clipboard>` class 595
Listing 26-10 The `<clipboard-flavor>` class 595
Listing 26-11 The `<clipboard-command>` class and its subclasses 596
Table 26-1 Functions for clipboard support 598

Chapter 27 Scripting Support 599

Table 27-1 Apple event suites and events 602
Table 27-2 Apple event data structures represented by Framework classes 603
Figure 27-1 An Apple script 604
Listing 27-1 Converting between Dylan objects and Apple event descriptor records 605

Listing 27-2 Responding to an Apple event 606
Listing 27-3 Responding to an Apple event 606
Listing 27-4 Associating a scripting object with a window 606
Listing 27-5 The `<ae-desc>` class 608
Listing 27-6 The `<ae-list>` class definition 609
Listing 27-7 The `<ae-record>` class definition 609
Listing 27-8 The `<apple-event>` class definition 610
Listing 27-9 The `<object-specifier>` class definition 610
Listing 27-10 Apple event descriptor class definitions 611
Listing 27-11 The `<token>` subclass 612
Listing 27-12 The `<scripting-event>` subclass 613
Listing 27-13 Scripting event subclasses 614
Listing 27-14 The `<property>` class 616
Listing 27-15 The `<document-property>` class 617
Listing 27-16 The `<file-property>` class 617
Listing 27-17 The `<text-view-property>` class 618
Listing 27-18 The `<text-view-selection-property>` class 618
Listing 27-19 The `<window-property>` class 619
Table 27-3 Object model functions 619
Table 27-4 Scripting event functions 620
Table 27-5 Apple event descriptor functions 620
Table 27-6 Apple event descriptor list and record functions 621
Table 27-7 Apple event object specifier and descriptor functions 622
Table 27-8 Apple event descriptor list and record functions 623

Chapter 28 Environment and Utilities 625

Listing 28-1 Using a handle string 631
Listing 28-2 Using the `without-busy-cursor` macro 633
Listing 28-3 Initializing a library 634
Listing 28-4 The `<region>` class definition 635
Listing 28-5 The <rect> class definition 636
Table 28-1 Point functions 637
Table 28-2 Rectangle functions 637
Table 28-3 Rectangle or region functions 638
Listing 28-6 The `<handle-string>` class definition 639
Listing 28-7 The `<resource-string>` class definition 640
Table 28-4 String functions 641
Listing 28-8 The `<framework-error>` class definition 642
Listing 28-9 The `<os-error>` class definition 643

Table 28-5 Error condition functions 644
Table 28-6 Error string functions 644
Table 28-7 Functions for `<os-error>` objects 645
Listing 28-10 The `<debugger-behavior>` class definition 646
Listing 28-11 The `<busy-cursor>` class definition 646
Table 28-8 Busy cursor functions 648
Listing 28-12 The `<framework-library>` class definition 649
Table 28-9 Framework library functions 651
Listing 28-13 The `<external-reference-table>` class definition 652
Table 28-10 External reference functions 653
Table 28-11 Environment functions 654
Table 28-12 Miscellaneous utility functions 654
Table 28-13 Miscellaneous goodie functions 655

About This Book

This book lists the language extensions that Apple has made to the Dylan language, including the Apple Dylan framework extensions.

This book is not designed explicitly to teach object-oriented programming nor the Dylan language. Familiarity with Macintosh application development, object-oriented programming and the Dylan language is assumed. For information about how to write Apple Dylan programs, see the book *Programming in Apple Dylan*. For information about the Dylan language, see the book *Dylan Reference Manual*. For information about Macintosh programming, see *Inside Macintosh*.

Chapter Organization

This book is divided into two parts; Part I covers the Apple Dylan language extensions, and Part II covers the Apple Dylan framework. Most chapters in this book follow a standard general structure. For example, the chapter "Event Handling" on page 241 contains these sections:

- "About Event Handling." This section provides an overview of the event-handling mechanism in the Framework.
- "Using Event Handling Objects." This section describes the tasks you can accomplish by handling events. It describes how to use the most common functions and provides code samples.
- "Event Handling Objects Reference." This section provides a complete reference for general purpose event-handling classes. (Classes related to menu event handling, for example, are described in the Menus chapter.) This section also identifies the functions that you can use with event handling.

Conventions Used in This Book

This book uses various conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and the names of data structures, constants, fields, parameters, and functions are shown in a monospaced font (`this is monospaced`).

When new terms are introduced, they are in **boldface.**

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is especially important. ▲

Numerical Formats

Hexadecimal numbers are shown in this format: #x0008.

The numerical values of constants are shown in decimal, unless the constants are flag or mask elements that can be summed, in which case they are shown in hexadecimal.

For More Information

APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

PART ONE

Language Extensions Reference

PART ONE

CHAPTER 1

Introduction

Contents

Apple Dylan and C-Compatible Libraries 5
Interface Importation 6
Cross-Language Calls 7

This chapter provides a brief overview of the Apple Dylan language extensions used to communicate with C-compatible libraries. The chapter begins by examining the motivation of these language extensions, and then it surveys the features provided by them.

Neither this chapter nor any part of this book is intended to provide a tutorial introduction to these facilities. You can find tutorial material in the accompanying book, *Programming In Apple Dylan*.

Apple Dylan and C-Compatible Libraries

Although Apple Dylan is a powerful and versatile language, there are still a number of reasons why you might want to call functions written in a language like C:

- You may want to use code you've already written, without rewriting it in Apple Dylan.
- You may want to use a commercially-available library of subroutines for which you don't even have the source code.
- You may want to take advantage of the strong points of another language—such as the fast floating-point computation available in C.
- You may want to continue to spend time working in a familiar programming language while you learn how to program with Apple Dylan.
- You may want access to the Macintosh Toolbox routines from your Apple Dylan program.

To address these issues, Apple Dylan provides a set of extensions to the standard Dylan language. These language extensions allow Apple Dylan programs to call functions written in C or in any **C-compatible language**—that is, any language that can be called from C.

You can also use these language extensions, called the C-compatible library language extensions, or simply the **C library extensions,** to call Apple Dylan functions from functions in C libraries.

These two features—calling C code from Apple Dylan and calling Apple Dylan code from C—are called **cross-language calls.** Providing cross-language calls is the primary motivation behind the C library extensions. However important,

the ability to make cross-language calls is only one of the many features offered by the Apple Dylan language extensions.

Interface Importation

The C library extensions of Apple Dylan offer a number of features to facilitate communication between Apple Dylan and other programming languages.

The first of these is **interface importation.** Before your Apple Dylan program can make calls to C library functions, it must have some internal representation of the interface to those functions—for example, your Apple Dylan program needs to know what functions are available to it, and how many and what type of values those expect as input and provide as output.

C-compatible languages use C header files to specify this type of information. Interface importation is the process Apple Dylan uses to parse C header files and produce a set of Apple Dylan objects that act as mediators between your Apple Dylan program and C library code. In general, the goal of interface importation is to create an Apple Dylan interface to C library functions—a set of Apple Dylan objects that look and act like other Apple Dylan objects, but that are actually conduits to C libraries.

Here are some parts of a C interface you can import into Apple Dylan:

- **Constants.** C header files can define constants in a number of ways: `const` variables, `#define` macros, `enum` constants. You can import constants defined in a C header file into Apple Dylan. The resulting objects are Apple Dylan constants—read-only variables you can reference in your program.
- **Types.** C header files also define data types. These type definitions control the shape of the data passed to and from the functions. Of particular importance in C is the structure type definition. Apple Dylan imports structure types as classes. You use these classes like other Dylan classes—making instances, accessing slots, passing to functions. The difference is that these classes' objects are implemented using standard C structures. Therefore, they can be accessed by C functions as easily as by Dylan ones.
- **Functions.** Function definitions, of course, are a crucial part of C header files. When importing functions, Apple Dylan creates function objects. These function objects can be called from your program like other Apple Dylan functions, but the purpose of one of these functions is to translate

arguments, adjust the calling sequence, call a C function, and translate return values. These functions perform the overhead necessary to transform normal Apple Dylan function calls into a cross-language communication.

Apple Dylan provides an extensive amount of default behavior that governs the interface importation process for you. However, there are always mechanisms provided for you to override the default behavior when you want more explicit control. Most of this part of the book explains Apple Dylan's default behavior or describes how and why you might override it.

For example, one of the processes that Apple Dylan handles for you when importing C header files is **name mapping**—choosing the names of the Apple Dylan objects created to represent imported C definitions. Apple Dylan provides some default name mapping for you. For example, when you import a structure type definition, Apple Dylan creates a class by the same name—except angle brackets are added around the name, in keeping with the Apple Dylan naming conventions. By default, then, importing the `Line` structure type, creates a class named `<Line>`.

Another process that occurs automatically during interface importation is **type mapping.** Whenever a C definition is imported, the C types used by that definition must have Apple Dylan counterparts. For example, importing a C function that takes two arguments creates an Apple Dylan function that takes two arguments. In the C function definition, the arguments have C types. Apple Dylan chooses (or creates) Apple Dylan classes to associate with the original types. For example, imagine that the original argument types are `long` and `Boolean`. Apple Dylan maps those C types to the built-in Apple Dylan classes `<integer>` and `<Boolean>`. The imported Apple Dylan function, then, takes an `<integer>` argument and a `<Boolean>` argument. The original C types of the function arguments are thus mapped to Apple Dylan classes.

Cross-Language Calls

After an interface is imported, you can use the resulting Apple Dylan objects to perform **cross-language calls.**

Before you attempt to call an imported function, or examine an imported variable, you must inform Apple Dylan how to find the compiled function or the stored variable. In effect, you must ensure that Apple Dylan has a way to link the interfaces imported from a C header file to actual runtime C objects in

memory. Apple Dylan supports five different linking mechanisms you can choose. These linking mechanisms are called **access paths.** You can use these access paths to find Macintosh Toolbox functions, functions in shared libraries, code fragments, code resources, and so on.

Access paths solve the problem of finding C runtime entities. Another problem that must be addressed is the problem of communicating with these entities. How do you use Apple Dylan objects to pass information in and out of C functions? The answer is **value translation,** which is a consequence of the type mappings established during interface importation. During a cross-language call, the type mappings for the arguments (of the function being called) control how values encoded as Apple Dylan objects are translated to C values, and how C values are translated back into Apple Dylan objects.

Apple Dylan passes many types of values between languages by pointer. To facilitate this, Apple Dylan provides **pointer objects**—objects used by your Apple Dylan program that contain pointers to C runtime entities—pointers to C structures, pointers to C strings, pointers to C functions, and so on. Apple Dylan provides an array of **pointer classes** for the different kinds of pointers. During value translation, Apple Dylan converts between C pointers and Apple Dylan pointer objects that point to the same address in memory.

Pointer-to-function objects provide two more powerful abilities:

- You can use these pointer objects to call C functions by pointer using the **callout** mechanism: You use Apple Dylan to create a function that takes a function pointer as an argument and calls the C function pointed to by that pointer.
- You can also use function-pointer objects to call Apple Dylan code from C library functions using the **callback** mechanism: You create a C-callable Apple Dylan function and Apple Dylan provides you with a function-pointer object for it. When you pass the function-pointer object to C functions, value translation translates it into a valid C pointer, which the C function can then use to call the Apple Dylan function.

Finally, Apple Dylan provides some low-level facilities that you can use to manipulate pointers and data in memory directly. You can also use these facilities to call functions using a machine-level interface.

CHAPTER 2

Interface Definitions

Contents

About Interface Definitions 11
  Interface Importation 12
    The Interface Importation Process 13
    Importing C Definitions 14
    Importing C Containers 16
    Name Mapping 16
    Type Mapping 17
  Interface Definitions 18
    File Clauses 19
    Other Clauses 21
Using Interface Definitions 23
  Using File Clauses 23
    Importing C Header Files 23
    Controlling Conditional Processing 24
    Selectively Importing or Excluding C Definitions 26
    Controlling Dynamism 27
  Using Other Types of Clauses 31
    Using Structure and Union Clauses 31
    Using Function Clauses 34
    Using Variable Clauses 35
    Using Constant Clauses 37
    Using Callout and Callback Clauses 38
Interface Definition Reference 39
  Interface Definition Syntax 39
  Interface Definition Clauses 39
  File Options 42

Container Options 43
Constant Options 46
Variable Options 47
Function Options 49
Callback Options 51

This chapter describes **interface definitions,** which are provided by Apple Dylan to allow you to import C entities. You use interface definitions to specify a standard C header file for importation. When you compile an interface definition, Apple Dylan parses the C header file and imports the entities defined within it as corresponding Apple Dylan entities.

This chapter also describes

- the **default importing behavior** that determines how Apple Dylan imports entities from C header files
- the **clauses** and **options** you can use within interface definitions to override and modify the default importing behavior

This chapter provides a complete overview of interface definitions. Certain clauses and options, however, are discussed more thoroughly in later chapters, which focus on specific aspects of interface importation. In particular, each of these chapters contains additional information concerning interface definitions:

- Chapter 3, "Name Mapping," discusses the default name-mapping behavior and shows how to use interface definition clauses and options to override that behavior.
- Chapter 4, "Cross-Language Calls," discusses, and gives examples of, the clauses and options you use when importing functions and function pointers.
- Chapter 5, "Type Mapping," discusses the default type-mapping behavior and shows how you can use interface definition options to override that behavior.
- Chapter 6, "Access Paths," includes examples of the interface definition options you use to specify access paths.

About Interface Definitions

Interface definitions are the Apple Dylan constituents you use to control **interface importation**—the process of importing definitions from a C header file into Apple Dylan. This section first discusses the interface importation process itself, and then describes how interface definitions allow you to control that process.

Interface Importation

Interface importation is the process by which a programming interface defined in a C header file becomes available to Apple Dylan. When you import an interface, Apple Dylan creates objects that correspond to entities defined in the C header file. These objects, called **imported objects,** provide your program with access to the C definitions.

There are two basic categories of C entities Apple Dylan allows you to import:

- **C definitions,** which include function, constant, variable, and type definitions.
- **C containers,** which include C header files and structure and union type definitions.

Note that structure and union types are both definitions and containers.

Apple Dylan imports C definitions as objects, as described in this table:

This C definition:	is mapped to this kind of Apple Dylan object:
function	function
`#define` function	function
`#define` constant	named constant (read-only variable)
`const` constant	named constant (read-only variable)
`enum` constant	named constant (read-only variable)
variable	getter function and setter function
type	class
function-pointer type	class

C containers may also be imported as Apple Dylan objects:

This C container:	is mapped to this kind of Apple Dylan object:
C header file	*(no corresponding object)*
`struct` type	class
`union` type	class

C header files are not imported as objects. Instead, the entities defined within them are imported individually, as indicated in these two tables. Structure and

union types are imported as Apple Dylan classes. The entities defined within structure and union types are also imported:

This C definition:	is mapped to this kind of Apple Dylan object:
structure member	getter function and setter function
union member	getter function and setter function

The Interface Importation Process

To import a C entity, Apple Dylan parses the definition of the entity in a C header file and translates the definition into Apple Dylan. Two important aspects of the translation process are

- **Name mapping.** Apple Dylan translates the name of the C definition to an appropriate Apple Dylan name. For example the C structure type name `Point` is by default translated to the Apple Dylan name `<Point>`.
- **Type mapping**. Apple Dylan associates an Apple Dylan class with each C type used in the definition. For example, if the definition is a function definition with one argument of type `long`, Apple Dylan associates the Apple Dylan class `<integer>` with that argument when importing the function.

These processes are introduced in more detail later in this chapter. You can find the complete discussions in Chapter 3, “Name Mapping,” and Chapter 5, “Type Mapping.”

After parsing the C header file and translating the C definition, Apple Dylan creates an Apple Dylan object that corresponds to the C entity (or, in some cases, two Apple Dylan objects—a getter and a setter).

Apple Dylan provides **default behavior** for each aspect of the importation process. You can modify this behavior in two ways: you can specify different default behavior for the definitions inside an imported container, and you can explicitly override the default behavior for specific imported definitions.

Importing C Definitions

This section discusses the different types of C definitions you can import from C header files and includes information specific to each.

- **Constants.** You can import three kinds of constants from C header files:
  - variables declared with a `const` modifier
  - constants defined within an `enum` declaration
  - macro constants defined with the `#define` directive

  A **macro constant** is a `#define` macro with no parameter list (macros with parameter lists are imported as functions as described below). The body of the macro must be a constant expression.

  Apple Dylan translates the constant value when the constant is imported. Therefore, when you reference the constant in your program, no cross-language call is necessary.

  Apple Dylan allows you to import C constants as read-only variables or as inline code.

- **Functions.** Apple Dylan allows you to import C function declarations from C header files into Apple Dylan. In addition to standard function declarations, you can also import **macro functions**—`#define` macros with parameter lists. However, Apple Dylan must be able to translate the body of the macro. In particular, after macro expansion the body of the macro must be a simple expression that contains only these types of references:
  - references to the parameters of the macro
  - function calls to imported functions
  - type casts using imported types

  As with constants, you can import C function definitions as runtime objects or as inline code.

  Apple Dylan creates a new function object that provides an Apple Dylan interface to a C function. When you call this function:
  - It uses type mapping to translate the Apple Dylan objects you provide as arguments into C values.
  - It uses an access path to locate the corresponding C library function.
  - It uses C calling conventions to pass the arguments to the C function.
  - It uses type mapping again to translate any C values returned by the C library function into Apple Dylan objects.

- It returns the resulting objects as the function results.

Chapter 4, "Cross-Language Calls," and Chapter 5, "Type Mapping," discusses how values are translated and passed between Apple Dylan and C during cross-language function calls. Chapter 6, "Access Paths," describes the access paths you use to link the imported function interface to the actual C library function.

- **Variables.** You can also import variables into Apple Dylan. As with constants and functions, you can import C variable definitions as runtime objects or as inline code.

  Apple Dylan creates a pair of function objects that provide an Apple Dylan interface to the C variable: a getter function that takes no arguments and a setter function that takes one argument.

  When you call the getter function:

  - It uses an access path to locate the C variable.
  - It uses type mapping to translate the C value to an Apple Dylan object.
  - It returns that object as the result of calling the getter.

  When you call the setter function:

  - It uses type mapping to translate the Apple Dylan object you provide as an argument into a C value.
  - It uses an access path to locate and set the value of the C variable.

  Therefore, accessing an imported variable is simply a cross-language call. See Chapter 4, "Cross-Language Calls," and Chapter 5, "Type Mapping," for more information.

  Not all access paths provide access to C variables. See Chapter 6 for details.

- **Members.** Apple Dylan also imports members of structures and unions as getter/setter function pairs. You use these functions as you use slot accessors. Each member has its own pair of functions. You provide a reference to the structure or union as an argument, and the function reads or writes a particular member of that structure or union.

  Chapter 5, "Type Mapping," gives examples of importing structure and union members.

- **Structures, unions, and other types.** Apple Dylan imports type definitions, including structure type definitions and union type definitions, as Apple Dylan classes. You can use these classes to create and manipulate C values. See Chapter 5, "Type Mapping," for information about importing types.

Importing C Containers

In addition to importing individual C definitions, you can use Apple Dylan to import **containers**—groups of C definitions—all at once. There are two kinds of C containers you can import:

- **C header files.** Apple Dylan allows you to import entire C header files; that is, to import every importable C definition from a C header file into Apple Dylan. Apple Dylan does not create an object to represent the C header file itself.
- **C structures and unions.** Apple Dylan allows you to import C structures and C unions defined in C header files. As mentioned in the previous section, the structure (or union) definition itself is imported as an Apple Dylan class. You also import the members of the structure (or union) as slot accessors—getter/setter pairs of function objects.

Apple Dylan allows you extensive control over how definitions in a container are imported. For example, you do not have to import every definition in a container; you can import any subset either by specifying which definitions to include or which ones to exclude.

You can also specify options that control the default importing behavior for definitions in the container. In addition, you can override these defaults for individual definitions within the container.

Here is an example: By default, Apple Dylan creates both a getter and a setter function for each imported variable or structure member.

You might override the default Apple Dylan behavior when importing a C header file by specifying that all variables imported from that file should be imported as getter functions only.

Name Mapping

Apple Dylan allows you to rename C definitions when you import them into your Apple Dylan program. Common reasons for renaming imported definitions include:

- to resolve name conflicts that arise when importing multiple C definitions into the same Apple Dylan module
- to ensure that the imported names follow Apple Dylan naming conventions

The process of renaming C definitions as they are imported is called **name mapping.** Apple Dylan performs some name mapping by default:

- When importing a C entity as a class, Apple Dylan adds angle brackets around the name of the C entity. For example, importing a C structure type named `Point` results in an Apple Dylan class named `<Point>`.
- When importing a constant, Apple Dylan adds a dollar-sign character before the C name of the constant. For example, the C constant named `kMenuID` is imported as an Apple Dylan constant named `$kMenuID`.
- When importing a member of a structure (or union), Apple Dylan creates the name of the resulting getter function by concatenating the C structure name and a dollar sign character to the C member name. For example, the `x` field of C structure `Point` is imported as a getter function named `Point$x` and a setter function named `Point$x-setter`.
- When importing other C entities, Apple Dylan uses the name of the entity exactly as it is defined in the C header file. For example, the C function named `AddPoint` is imported as an Apple Dylan function named `AddPoint`.

You can override this default behavior in two ways. Whenever you import any container, such as a C header file, you can select from a variety of different default behaviors for name mapping the entities within that container. In addition, you can explicitly override the name mapping when importing any C entity.

Chapter 3, “Name Mapping,” discusses name mapping in detail.

Type Mapping

When you make a cross-language call, Apple Dylan performs **value translation**—the passing of values between the two languages.

Apple Dylan prepares for the value translations necessary during cross-language calls by creating **type mappings** during interface importation. When you import a C interface, Apple Dylan associates an Apple Dylan class with every C type referenced in the imported interface. These associations, between Apple Dylan classes and C types, are called type mappings.

For example, when you import a C function, Apple Dylan creates a type mapping for each of the function’s arguments—each argument has a C type and an associated Apple Dylan class. The C type, declared in the C header file, is the type of C value the function expects for the argument. The associated

Apple Dylan class is the class of Apple Dylan object the imported function expects for the argument.

When you call the imported Apple Dylan function, you provide arguments of the appropriate Apple Dylan classes. Apple Dylan translates each argument to a C value of the C type specified by the argument's type mapping.

In this way, the type mappings established during interface importation allow for type checking and value translation during cross-language calls.

During interface importation, Apple Dylan performs default type mapping for you. You can modify the default type-mapping behavior for any imported C container. You can also override the default behavior by specifying explicit type mappings for any imported C entity.

Chapter 5, "Type Mapping," provides the complete discussion of type mapping and value translation.

Interface Definitions

Apple Dylan provides the **interface definition**, which you can use to import C interfaces. An interface definition is a program constituent—you use it like other types of Apple Dylan definitions.

An interface definition is structured as a series of clauses. Each **clause** specifies a C definition or a C container to import. Each clause can also contain **options,** which you use to override the default behavior provided by Apple Dylan.

The syntax of an interface definition is

```
define interface
    { interface-clause ; }
end [ interface ]
```

An interface definition consists of the words `define interface`, followed by some number of interface clauses separated by semicolons, followed by the word `end`, and optionally the word `interface`.

Each clause specifies the C definition or C container to import, and any options to modify the default importing behavior. There are eight different types of clauses you can specify. The first clause must always be a file clause, which specifies the C header file that contains the definitions you want to import.

When you import a C header file that includes other C header files, Apple Dylan does not automatically import the included C header files. You should provide an interface definition for every C header file that contains definitions you want to import.

You execute an interface definition, like any Apple Dylan definition, in a specific module. Apple Dylan imports the C entities into that module.

The section "Interface Definition Syntax" on page 39 contains reference information about interface definitions.

File Clauses

In an interface definition, you use a **file clause** to control the overall importation of C entities from a C header file. C header files, which can contain multiple definitions, are C containers. Therefore, a file clause is a type of **container clause.**

The file clause is the most important and the most common type of clause. You can import every definition in a C header file using a single file clause. The other types of clauses allow you to specify more information about importing specific C definitions and C containers contained in the C header file.

In a file clause, you specify a file name. If the file name is an absolute pathname (it contains a colon and the first character is not colon), then the file name is the location of the file. Otherwise, the file name is a relative pathname, which can be one of the following:

- a file name (containing no colon)
- a colon, a sequence of sub-folders followed by colons, and a file name.

If you specify a relative pathname, Apple Dylan first looks for the file at the specified location relative to the project folder and then looks for the file relative to the CIncludes folder in the Apple Dylan Files folder. (Note that if the current project is a new project that has never been saved, it has no project folder.)

Here is an example of an interface definition containing a single file clause:

```
define interface
    #include "SampleHeader.h";
end interface;
```

This interface definition imports every importable C entity in the file `SampleHeader.h`. All of the constants, variables, functions, structures, and unions are imported. Since no options are specified, Apple Dylan imports these C entities using the default importation rules:

- All importable C entities are imported.
- Constants are imported as inline code.
- Variables and functions are imported as inline code.
- Variables are not assumed to be read-only; that is, they are imported as both a getter function and a setter function.
- Default name mapping occurs (described fully in the next chapter).
- Default type mapping occurs (described fully in Chapter 5).

You can override these defaults, specifying customized importing behavior for the header file, using **container options**—options that specify how to import C definitions from a C container (in this case, a C header file).

Some common reasons for overriding the default behavior are listed here:

- The C header file might define C entities that you don't want to import into your Apple Dylan program.
- Naming conflicts may result from the different naming rules used by the two languages. The C language is case-sensitive, for example, whereas Apple Dylan is not. Therefore, two entities with different names in the C header file (a variable `random` and a function `Random()`, for instance) may be imported as two Apple Dylan objects with the same name.
- The C header file may contain entities with the same name as entities already defined by Apple Dylan, or already defined in your program.
- The C header file might contain utilities that are not accessible through the access path you choose.

The container options you can use to modify default importing behavior are listed here:

- The **import** and **exclude options** allow you to selectively import definitions from the container. The section "Selectively Importing or Excluding C Definitions" on page 26 gives examples using the import and exclude options.

- The **inline-constants**, **seal-functions**, and **read-only options** allow you to specify how Apple Dylan imports constants, functions, and variables. The section "Controlling Dynamism" on page 27 give examples that use the inline-constants, seal-functions, and read-only options.
- The **prefix, rename,** and **name-mapper options** allow you to control name mapping. See Chapter 3, "Name Mapping," for more information.
- The **type option** allows you to control type mapping. See Chapter 5, "Type Mapping," for more information.

The section "Container Options" on page 43 contains reference information about these options.

The container options listed above control how the C entities contained within a file are imported. To specify information about the C header file itself, Apple Dylan provides the **file options:**

- The **define** and **undefine options** allow you to use preprocessor macros to control the parsing of the header file. The section "File Clauses" on page 19 gives examples using these options.
- The **language**, **external-module**, **CFM-library**, **CFM-library-version**, and **CFM-library-oldest-version options** allow you to specify information about the access path used to locate C library objects during cross-language calls. See Chapter 6, "Access Paths," for more information.

The section "File Options" on page 42 contains reference information about these options.

Other Clauses

In general, file clauses allow you to import entities from a C header file, and container options allow you general control over how these C entities are imported. The other types of interface definition clauses are provided to allow you explicit control over the importation of specific C entities.

In addition to file clauses, Apple Dylan provides these container clauses:

- **Structure and union clauses** allow you to specify information about how to import a specific structure; in particular, you can specify information about how to import the members of the structure.

These clauses, like file clauses, allow you to use container options to specify how the definitions within the container (the members of the structure or

union) are imported. For example, you can use the read-only container option to specify that the members of the structure are imported as getter functions without corresponding setter functions.

The container options you specify in a a structure or union clause override the container options for the file.

Apple Dylan also provides these three definition clauses:

- **Constant clauses** allow you to modify the importation of a specific constant. You use **constant options** to specify information about how to import the constant.
- **Variable clauses** allow you to modify the importation of a specific variable. You use **variable options** to specify information about how to import the variable.
- **Function clauses** allow you to modify the importation of a specific function. You use **function options** to specify information about how to import the function.

Constant, variable, and function options override the default options specified for the container that contains the constant, variable, or function. You can find examples of these options in "Using Other Types of Clauses" on page 31.

There are also two special interface definition clauses. These clauses create interfaces based on imported C pointer-to-function types:

- **Callback clauses** create Apple Dylan functions that can call C functions using a function pointer.
- **Callout clauses** create Apple Dylan macros that you can use to create Apple Dylan functions that can be called from C.

You can find examples using these types of clauses in Chapter 4, "Cross-Language Calls."

The section "Interface Definition Reference" on page 39 contains reference information for all of the interface definition clauses and options.

Using Interface Definitions

This section contains examples that show how to use interface definitions to import C entities from a C header file.

The examples included here illustrate the use of many of the clauses and options you can use with interface definitions. However, examples of some clauses and options are postponed to later chapters. In particular:

- Chapter 3, “Name Mapping,” discusses the name-mapping options.
- Chapter 4, “Cross-Language Function Calls,” discusses the function, callout, and callback clauses, as well as the function options.
- Chapter 5, “Type Mapping,” discusses the type-mapping options, as well as structure and union clauses.
- Chapter 6, “Access Paths,” discusses the file options pertaining to access paths.

Using File Clauses

The file clause controls importation of entire C header files. The examples in the next few sections illustrate the use of this type of clause. In particular, these sections include examples that use file options to control the parsing of a file, and examples that use container options to modify the default importing behavior for C entities within the file.

Importing C Header Files

The simplest version of an interface definition contains a single file clause with no options, for example:

```
define interface
    #include "QuickDraw.h";
end interface
```

This interface definition specifies that Apple Dylan **import** every entity defined in the C header file `QuickDraw.h`. In other words, when this interface definition is compiled, Apple Dylan creates an Apple Dylan interface to every importable C entity defined in the header file.

Apple Dylan provides many options you can specify in the file clause of an interface definition. These options allow you to specify information about the C header file, to specify which C entities to import from the file, and to specify default importing behavior for those entities.

Controlling Conditional Processing

The **define** and **undefine** file options allow you to control conditional processing when Apple Dylan parses a header file.

The define option allows you to define preprocessor macros. For example, this interface definition:

```
define interface
    #include "SampleHeader.h",
        define: { "my_macro" };
end interface;
```

indicates that Apple Dylan should import C entities from the file `SampleHeader.h`, parsing the file as if it contained this macro:

```
#define my_macro
```

You can also use the arrow (=>) option to specify a particular value for a preprocessor macro:

```
define interface
    #include "SampleHeader.h",
        define: { "my_macro" => 10 };
end interface;
```

This interface definition parses the `SampleHeader.h` file as if it contained the macro definition:

```
#define my_macro 10
```

You can include multiple definitions in a single define option. For example, consider the following interface definition:

```
define interface
    #include "SampleHeader.h",
        define: { "SystemSixOrLater" => 0 };
    end interface;
```

This interface definition indicates the Apple Dylan should parse the C header file as if it contained this C preprocessor macro:

```
#define SystemSixOrLater 0
```

which indicates that the functions in this header file will not necessarily be run under Macintosh System Software version 6.0 or later.

The five default settings for C preprocessor macros are

```
#define __Dylan
#define applec
#define SystemSixOrLater 1
#define OLDROUTINENAMES 0
#define CGLUESUPPORTED 0
```

These five definitions specify, respectively, that

- the header file is being parsed by Apple Dylan
- Apple C extensions are supported
- the functions defined in this header file will be run under Macintosh System Software version 6.0 or later
- the new version of system function names (for example `DisposePtr)`, should be used rather than the older versions (`DisposPtr`)
- Apple Dylan should not import functions defined solely to address the issues that arise when accessing the Macintosh Toolbox from the C language

These five options are the **default define options** provided by Apple Dylan—you do not have to specify them when importing a file. However, you do have to use the define (or undefine) option to make any changes to the default settings. For example:

```
define interface
    #include "SampleHeader.h",
        define: { "SystemSevenOrLater" => 1,
                  "_STDC_" => 1};
        undefine: { "applec" };
 end interface;
```

This interface defines two additional preprocessor macros, and also undefines one that would be defined by default.

The other file options concern access paths, and are discussed in Chapter 6, "Access Paths."

Selectively Importing or Excluding C Definitions

The import and exclude container options allow you to control which C entities are imported from a C header file.

The **import** container option allows you to limit which C entities are imported by naming the ones you want to import.

As an example, consider this C header file:

```
define interface
    #include "QuickDraw.h",
        import: { "Random", "GlobalToLocal" };
end interface
```

This interface definition imports two functions from the file `QuickDraw.h`—the `Random` function and the `GlobalToLocal` function. The other C entities defined in that file are not imported.

You can explicitly import all the C entities in a header file by specifying the value `all` for this option:

```
define interface
    #include "SampleHeader.h",
        import: all;
end interface;
```

This value is the default value for this option—if you do not explicitly limit the C entities to import, Apple Dylan imports them all.

Notice that the `all` option is not surrounded by braces or by quotes. The following interface definition does *not* import all of the entities from a C header file:

```
define interface
    #include "SampleHeader.h",
        import: { "all" };       // WARNING!  This is a common mistake.
end interface;
```

This definition attempts to import a C entity named `all`, rather than all of the entities.

You can also use the import option to rename C entities as you import them. See Chapter 3, "Name Mapping," for more details.

While the import option allows you to selectively import C entities by specifying the entities you want to import, the **exclude** option allows you to selectively import C entities by naming the ones you do *not* want to import.

Here is an example:

```
define interface
    #include "Files.h",
        exclude: { "AddDrive", "FSOpen" };
end interface;
```

This interface definition imports every C entity from the header file `Files.h` *except* the functions named `AddDrive` and `FSOpen`. You might want to use the exclude option for a number of reasons:

- to exclude Macintosh Toolbox functions that are not in ROM (because Apple Dylan has no access path to these functions)
- to exclude other unnecessary or undesired C entities
- to exclude redundant types, as explained in Chapter 5, "Type Mapping."

Controlling Dynamism

The container options described in this section allow you to specify information about how C definitions in a C header file should be imported:

- The **inline-constants** container option allows you to specify whether constants imported from the C header file should by imported as named constants (that is, objects that exist at runtime) or as inline code.
- The **read-only** container option allows you to specify whether variables imported from the C header file are imported as getter functions, or as getter/setter function pairs.
- The **seal-functions** container option allows you to specify whether functions imported from the C header file should by imported as sealed generic function methods, open generic function methods, or as inline code. This option also applies to the getter/setter functions for imported variables.

These three options allow you to specify default importing behavior for an entire C header file. You can always override default behavior for a specific C entity by using an interface definition clause for that entity, as described in later sections of this chapter.

The **inline-constants** container option allows you to specify whether imported constants are imported as Apple Dylan objects, or whether the Apple Dylan compiler replaces references to them with inline code—that is, no runtime object is created. Inlining saves space, especially if you import the entirety of a large header file but only use a few of its constants.

By default, Apple Dylan inlines imported constants. You can override this behavior using this container option:

```
define interface
    #include "SampleHeader.h",
        inline-constants: #f;
end interface;
```

This interface definition specifies that constants imported from the header file `SampleHeader.h` be imported as objects that exist during runtime while your program is executing.

Whether the default behavior you choose is to inline constants or to create runtime objects, you can always override that default behavior for specific constants using a constant clause, as described in “Using Constant Clauses” on page 37.

Setting the inline-constants option to `#f` causes constants to be defined in the style of `define constant`. Setting this option to `#t` causes constants to be defined

in the style of `define inline constant` (with the additional feature that no runtime object is created for unused constants).

The **read-only** option allows you to specify whether imported variables (and imported structure members) are imported as getter functions, or as getter/setter pairs of functions. By default, Apple Dylan imports variables as getter/setter function pairs. You can override this behavior using this container option:

```
define interface
    #include "SampleHeader.h",
        read-only: #t;
end interface;
```

With this interface definition, all of the variables imported from the file `SampleHeader.h` are imported as getter functions. No setter functions are created.

You can override the default behavior for specific variables by using variable clauses, as described in "Using Variable Clauses" on page 35.

The **seal-functions** option (which also applies to function macros and to getter and setter functions for imported variables and structure members) allows you to select one of the following three representations for imported functions:

- As a generic function method, specified by `seal-functions: open`. In this case, Apple Dylan creates a runtime object (a method) that represents the C entity, and adds this method to a generic function. This approach maximizes runtime flexibility, but it increases memory usage and may slow cross-language calls slightly.
- As a generic function method with a sealed domain, specified by `seal-functions: seal`. This case is the same as the previous case except it uses `define sealed method` instead of `define method`, allowing for increased compiler optimization.
- As a bare method that is always compiled inline, specified by `seal-functions: inline`. This case is the default. The function is defined in the style of `define inline constant` (with the additional feature that no runtime object is created for unused functions). In this case, Apple Dylan does not create a runtime object to represent the C entity; instead, the development environment maintains information about the imported entity and uses this information to replace references to the imported entity with inline code. This approach provides little runtime flexibility, but it eliminates memory requirements and optimizes the speed of cross-language calls. This

can save a lot of space if you import the entirety of a large header file but only call a few of its functions.

You can override this default behavior to import functions as generic functions. For example:

```
define interface
    #include "SampleHeader.h",
        seal-functions: open;
end interface;
```

This interface definition imports all of the C entities from the header files `SampleHeader.h` as methods added to generic functions.

You can also import functions as generic function methods with a sealed domain:

```
define interface
    #include "SampleHeader.h",
        seal-functions: sealed;
end interface;
```

This interface definition imports functions as method objects, adds the method objects to generic functions, and seals the generic function branches. (It works similarly to `define sealed method`.)

- Open functions provide the greatest amount of dynamism. However, they also potentially require the greatest amount of runtime storage space and the greatest amount of overhead when making a cross-language function call.
- Sealed functions provide less dynamism, but are potentially more efficient than open functions.
- Inline functions provide the least dynamism. However, this is the fastest interface to imported functions and no runtime objects are required.

The seal-functions option specifies the default behavior for all functions and function macros imported from a C header file. It also applies to all of the getter and setter functions created when importing variables.

You can override the default behavior for specific functions and variables by using function and variables clauses, as described in “Using Function Clauses” on page 34 and “Using Variable Clauses” on page 35.

The remaining container clauses control name mapping and type mapping. See Chapter 3, “Name Mapping,” and Chapter 5, “Type Mapping,” for examples using those options.

Using Other Types of Clauses

As you have seen in the previous sections, file clauses and container options allow you to control the process of importing C entities from a C header file.

Apple Dylan provides other types of interface importation clauses, which you use to specify additional information when importing specific C entities. In the following sections, you’ll see examples of

- structure and union clauses
- constant clauses
- variable clauses
- function clauses

You can find related information in these chapters:

- Chapter 3, “Name Mapping,” discusses the name-mapping options you can use with these clauses.
- Chapter 4, “Cross-Language Calls,” gives more examples of function clauses, and also gives examples of callout and callback clauses.
- Chapter 5, “Type Mapping,” discusses the type-mapping options you can use with these clauses and gives further examples of importing structure types.

Using Structure and Union Clauses

Apple Dylan imports C structures (and unions) as Apple Dylan classes. For example, consider this structure type defined in the header file `Types.h`:

```
struct Point {
    short v;
    short h;
};
```

You can import this structure type using the following interface definition:

```
define interface
    #include "Types.h",
     import: { "Point" => <CPoint> };
end interface;
```

This interface definition contains a single clause—a file clause with an import option specifying that the C entity named `Point` should be imported from the C header file `Types.h` (and renamed `<CPoint>` to avoid conflicts with the built-in class `<Point>`).

Importing this C structure type creates a single Apple Dylan entity—a class named `<CPoint>`. You can use this class to create C `Point` structures, and to pass C `Point` structures during cross-language calls. See Chapter 5, "Type Mapping," for examples.

This structure clause is a container clause. By default, a container clause indicates that Apple Dylan should import all entities defined within the container. As a result, the above interface definition creates five Apple Dylan entities:

- an Apple Dylan class named `<CPoint>`
- a function named `Point$v`
- a function named `Point$v-setter`
- a function named `Point$h`
- a function named `Point$h-setter`

You use these getter and setter functions to access members of a `Point` structure as if they were slots of an Apple Dylan class with the names `Point$v` and `Point$h`. (These names result from the default name-mapping rules discussed in the next chapter.)

By default, these functions are inline code. You can override that behavior using the seal-functions container option:

```
define interface
    #include "Types.h",
     import: { "Point" => <CPoint> };
```

```
    struct "Point",
        seal-functions: open;
end interface
```

This interface definition uses the seal-functions container option to override the default Apple Dylan behavior (which is to import structure members as inline code) and instead create methods, which are added to generic function objects, to represent the imported getter and setter functions.

You can also use other container options to modify the importation of these structure members:

```
define interface
    #include "Types.h",
     import: { "Point" => <CPoint> };

    struct "Point"
        seal-functions: open,
        read-only: #t,
        import: { "h" };
end interface
```

This interface definition imports the `Point` structure type. It also uses an import container option to import one of the members of this structure—the `h` member. It also uses the read-only container option to specify that no setter function should be created. As a result of this interface definition, three Apple Dylan objects are created:

- a class named `<CPoint>`
- a generic function named `Point$h`
- a method for `Point$h` specialized to the `<CPoint>` class

When you import a structure type (or a union type) that contains an anonymous structure type (or union type), Apple Dylan ignores the extra level of hierarchy. For example, consider this C struct:

```
struct doubleStruct {
    int i;
    struct {
        char a;
```

```
        char b;
    } s;
};
```

When you import this structure type, Apple Dylan creates a class with three slots: `doubleStruct$i`, `doubleStruct$a`, and `doubleStruct$b`. The original member named `s` is ignored; its members are used directly instead.

The reference at the end of this chapter discusses the syntax of container clauses and container options. You can find more information about importing structure types and union types in Chapter 3, "Name Mapping," and in Chapter 5, "Type Mapping."

Using Function Clauses

Function clauses allow you to specify more options, and override default options, when importing functions. In particular, you can use the **function options**, which are listed here, to specify information about the imported function:

- The **seal** option allows you to specify whether a function is imported as an open method, a sealed method, or as inline code. It overrides the seal-functions option of the C container containing the function.
- The **argument-type** and **result-type** options allow you to specify explicit type mappings for a function's arguments and result, respectively (which you might want to do, for example, when a C integer is used as a Boolean value). These type mappings override type mappings for the C container that contains the imported function. Chapter 5, "Type Mapping," discusses these options.
- The **input-argument, output-argument,** and **input-output-argument** options allow you to specify, for each argument of an imported function, whether the argument is used only for input, only for output, or for both. Chapter 4, "Cross-Language Calls," and Chapter 5, "Type Mapping," discuss these options in detail.

As a simple example, consider the following C function definition:

```
long Negate (long input);
```

You might import this function using the following interface definition:

```
define interface
    #include "SampleHeader.h",
        seal-functions: open;

    function "Negate",
        seal: inline;
end interface;
```

This interface definition imports entities from the C header file `SampleHeader.h`. In particular, it uses the seal-functions option to specify that functions imported from the header file should be imported as open methods by default.

The interface definition uses a function clause to override this default behavior for the imported C function `Negate`. The seal option in this clause specifies that this particular function should be imported as inline code. This option overrides the seal-functions option specified for the header file.

Note
To call imported functions, you must use one of the access paths described in Chapter 6, "Access Paths." ◆

For more examples of importing functions and using function clauses, see Chapter 4, "Cross-Language Calls," and Chapter 5, "Type Mapping."

Using Variable Clauses

Variable clauses allow you to specify more information about the getter and setter functions that Apple Dylan uses to access imported global variables and the members of imported C structures and unions.

You can use the variable clause, without using any variable options, to rename an imported structure member. You can also use the **variable options** to specify additional information, or to override default information, when importing a variable:

- The **getter** and **setter** options allow you explicit control over the names of the imported getter and setter functions. See Chapter 3, "Name Mapping," for more information.
- The **type** option allows you to specify the class that the imported variable should be type-mapped to—that is, the class of object that is returned by the

getter function and expected by the setter function. See Chapter 5, "Type Mapping," for more information.

- The **read-only** option allows you to specify that Apple Dylan create only a getter function for the imported variable or slot—that is, no setter function.
- The **seal** option allows you to specify whether the getter and setter functions for an imported structure member are open methods, sealed methods, or inline code.

For example, consider this interface definition

```
define interface
    #include "SampleHeader.h",
        import: { "Point" => <CPoint>, "origin" };

    variable "origin",
        seal: open;

    variable "Point::h",
        read-only: #t;
 end interface;
```

This interface definition imports a structure type named `Point`, a variable named `origin`, and the `h` member of the `Point` structure (and all the other members of `Point`, which are imported by default by the import option as specified in the file clause). In particular, this interface definition includes two variable clauses:

- The first variable clause uses the seal option to specify that the `origin` variable should be imported as getter/setter functions that are open methods, rather than as inline code.

  The resulting getter function takes no arguments and returns the value of the C variable `origin`. The setter function takes one argument, which it uses to modify this variable.

- The second variable clause uses the read-only option to specify that the `h` member of the `Point` structure type should be imported as a getter function only, rather than a getter/setter pair. Notice the use of `::` notation to identify the C entity and the C container that contains it.

  The resulting getter function takes a single argument, an object of class `<CPoint>`. The function returns the value of the `h` member of the

corresponding `Point` structure. See Chapter 5, “Type Mapping,” for an example.

You can also use the setter option to import a variable as a read-only variable:

```
variable "Point::h",
    setter: #f;
```

The value `#f` for the setter option indicates that no setter function should be created for the imported member.

Note
Not all access paths provide access to C library variables.
See Chapter 6, “Access Paths,” for more information. ◆

See the reference section at the end of the chapter for the syntax description of variable clauses and options. Chapter 3, “Name Mapping,” Chapter 4, “Cross-Language Calls,” and Chapter 5, “Type Mapping,” contain related information.

Using Constant Clauses

Constant clauses allow you to specify information about how Apple Dylan imports constants from C header files, using these two **constant options**:

- The **inline** option allows you to specify whether a constant is imported as inline code or as a named constant object in memory. This option overrides the default behavior for the C container that contains the constant.
- The **value** option allows you to specify an explicit Apple Dylan value for the constant, which you might want to do if the original C value is inappropriate.

Here is an example:

```
define interface
    #include "SampleHeader.h";

    constant "kConstant",
        inline: #f;
        value: 100;
end interface;
```

This interface definition imports C entities from the C header file `SampleHeader.h`, using the default importing behavior for all entities defined in the file except one. When that entity, the C constant named `kConstant`, is imported, Apple Dylan

- creates a runtime object (a named constant) to represent it
- sets the value of that object to be the `<integer>` value 100

Using Callout and Callback Clauses

Callout and callback clauses allow you to make cross-language calls using function pointers. For both of these types of clauses, you specify a C function pointer type to use. (Note: the C function pointer type must be imported separately.)

- A **callout clause** creates a corresponding **callout function.** You pass function pointers of the imported type to the callout function, and it calls the indicated C function for you.
- A **callback clause** creates a corresponding callback macro. You use the **callback macro** to define a **callback function**—a special Apple Dylan function that is callable from C. The callback macro provides you with a function pointer of the imported type that points to the callback function. You can pass this function pointer to C functions, which can use it to call your callback function.

You can use the function options with a callout or callback clause. When used with one of these clauses, they modify the resulting callout or callback function.

The reference section, which follows, gives the syntax of the callout and callback clauses. Chapter 4, "Cross-Language Calls," gives examples of using callout and callback functions.

Interface Definition Reference

This section describes the complete syntax of interface definitions, including each type of interface definition clause and option. It also describes the possible values for each of the options.

Interface Definition Syntax

The syntax of an interface definition is

```
define interface
    { interface-clause ;}
end [ interface ]
```

Each interface definition begins with the words `define interface`, contains some number of interface definition clauses separated by semicolons, and ends with the word `end` followed optionally by the word `interface`. The first clause must be a file clause.

Interface Definition Clauses

There are eight types of interface definition clauses, shown in the following list. Notice that in these clauses, C entities are named using string literals (that is, in quotes), whereas names of Apple Dylan entities are variable names (that is, not in quotes).

Type of clause	Syntax
file	`#include` *filename* { , *file-or-container-option* }
structure	`struct` *structure-name* [=> *imported-name*] { , *container-option* }
union	`union` *union-name* [=> *imported-name*] { , *container-option* }
constant	`constant` *constant-name* [=> *imported-name*] { , *constant-option* }
variable	`variable` *variable-name* [=> *imported-name*] { , *variable-option* }
function	`function` *function-name* [=> *imported-name*] { , *function-option* }
callout	`callout` *function-pointer-name* [=> *callout-name*] { , *function-option* }
callback	`callback` *function-pointer-name* [=> *callback-name*] { , *function-option* }

In these syntax descriptions, the terms *filename*, *structure-name*, *union-name*, *constant-name*, *variable-name*, *function-name*, and *function-pointer-name* are string literals (with quotes) that name C entities. When specifying names of structure and union members, you can use the :: notation to specify the name of the enclosing structure or union type.

The terms *imported-name*, *callout-name*, and *callback-name* are variable names (no quotes) that specify valid Apple Dylan names.

All clauses except for file clauses allow you to use the arrow option to explicitly rename an imported C entity.

The following list provides additional information about these clauses:

- A **file clause** is used to import entities from a header file. You can use file options, described in "File Options" on page 42, to provide information about the header file itself. You can also use container options, described in

"Container Options" on page 43, to specify information about how to import the C entities contained in the file.

- A **structure clause** is used to provide additional information when importing a C structure type and its members. You use the container options, described in "Container Options" on page 43, to provide information about how the members of the structure should be imported.
- A **union clause** is used to provide additional information when importing a C union type and its members. You use the container options, described in "Container Options" on page 43, to provide information about how the members of the union should be imported.
- A **constant clause** is used to provide additional information when importing a C constant definition. You use the constant options, described in "Container Options" on page 43, to provide information about how the constant should be imported.
- A **variable clause** is used to provide additional information when importing a C variable or a C structure or union member. You use the variable options, described in "Variable Options" on page 47, to provide information about how the variable should be imported.
- A **function clause** is used to provide additional information when importing a C function declaration. You use the function options, described in "Function Options" on page 49, to provide information about how the function should be imported.
- A **callout clause** is used to import a C function-pointer type and to create a callout function you can use to call C functions using function pointers. You can specify the function options, described in "Function Options" on page 49, in a callout clause.
- A **callback clause** is used to import a C function-pointer type and to create a callback macro, which you can use to create C-callable Apple Dylan functions. You can specify the function options, described in "Function Options" on page 49, in a callback clause, as well as the callback option, described in "Callback Options" on page 51.

File Options

The file options specify information about the header file and how it should be parsed. File options are specified as part of a file clause.

File option	Syntax
`define:`	{ { *string* [=> *literal*] ,} }
`undefine:`	{ { *string* ,} }
`language:`	`C` | `ASLM`
`external-module:`	*variable-name*
`CFM-library:`	*string*
`CFM-library-version:`	*integer*
`CFM-library-oldest-version:`	*integer*

These six file options exclusive to file clauses are described below.

- The **define option** defines preprocessor macros for conditional parsing of the header file. Define options contain a list of macro definitions, which have the syntax

  string [=> *literal*]

  where *string* is a string literal indicating the name of the macro to define, and *literal* is an optional literal constant that indicates the macro's defined value. (The *literal* constant defaults to the value 1; if you provide it, you must specify a number or a string.) See "Controlling Conditional Processing" on page 24 for the default value of this option.

- The **undefine option** undefines preprocessor macros when parsing the header file. Undefine options contain a list of string literals indicating the names of the C preprocessor macros to undefine.
- The **language option** indicates whether the file is a C header file or an ASLM header file. If the name of the header file ends in `.h`, the default is C. If the header file name ends in `.exp`, the default is ASLM.
- The **external-module option** indicates that the external-module access path should be used to locate the entities imported from a header file. With this option, you specify the name of the external module (using a variable name, not a string literal). This option is always ignored on the PowerPC.
- The **CFM-library option** indicates that the Code Fragment Manager access path should be used to locate the entities imported from a header file. You provide a string literal that indicates the name of the CFM library that contains the runtime C entities defined in the file. For example, on the PowerPC, most Toolbox functions are in `"InterfaceLib"`.

  This option can be used for code that works on both the PowerPC and the 68K. So long as all the functions that are imported have inline code for the 68K specified in the header file, the 68K will ignore this option.
- The **CFM-library-version option** indicates the version number of the CFM library that contains the entities imported from a header file. You provide an integer that indicates the desired version number.
- The **CFM-library-oldest-version option** indicates the oldest acceptable version of the CFM library. You provide an integer that indicates the oldest acceptable version number.

If neither the CFM-library-version nor the CFM-library-oldest-version options are specified, the default behavior is to cause the newest version of the library to be used. If CFM-library-version is specified and CFM-library-oldest-version is not, the latter defaults to the value of the former, thus requesting an exact version of a library, or a compatible version.

Container Options

The container options specify information about C entities within a C container. C containers include header files, structures, and unions. Therefore, the

container options can be used with file clauses, structure clauses, and union clauses.

When one container is inside another (for example, a structure type definition within a C header file), any container options specified for the inner container (for example, the structure) override container options specified for the outer container. The values for container options not specified for the inner container are the same as for the enclosing container (except for import, exclude, and rename).

Here are the syntax descriptions for the container options:

Container option	Syntax
`import:`	`all` | `{` { *string* [`=>` *variable-name*] `,` } `}`
`exclude:`	`{` { *string* `,` } `}`
`inline-constants:`	`#f` | `#t`
`seal-functions:`	`sealed` | `open` | `inline`
`read-only:`	`#f` | `#t`
`rename::`	`{` { *string* `=>` *variable-name* `,` } `}`
`prefix:`	*string*
`name-mapper:`	*function*
`type`	`{` { *string* `=>` *variable-name* `,` } `}`

The nine container options are described below:

- The **import option** allows you to specify which C entities should be imported from the container. Import options can have the syntax

  ```
  import: all
  ```

which indicates that all importable entities within the container should be imported; they can also have the syntax

{ { *string* [=> *variable-name*] ,} }

where *string* is a string literal naming a C entity and *variable-name* is the name for the imported Apple Dylan entity.

The default value for this option is `all`.

- The **exclude option** allows you to specify which C entities should not be imported from the container. This option contains a list of string literals indicating the names of the C entities to exclude.

  The default value is an empty list.

- The **inline-constants option** allows you to specify how constants defined in the C container should be imported. A Boolean value of `#t` indicates that constants should be imported as inline code and a value of `#f` indicates that runtime objects (named constants) should be created to represent the imported constants.

  For files, the default value is `#t`. For structures and unions, the default value is the value of the same option for the enclosing container.

- The **seal-functions option** allows you to specify how functions and accessors variables should be imported from a container. The value `open` indicates that functions should be imported as open generic function methods. The value `sealed` indicates that they should be sealed generic function methods. The value `inline` indicates that they should be bare methods and that calls to them should be replaced with inline code.

  For files, the default value is `inline`. For structures and unions, the default value is the value of the same option for the enclosing container.

- The **read-only option** allows you to specify whether variables and members imported from the container have setter functions or not. A Boolean value of `#t` indicates that variables are imported as getter functions only, and a boolean value of `#f` indicates that variables are imported as getter/setter pairs of functions.

  For files, the default value is `#f`. For structures and unions, the default value is the value of the same option for the enclosing container.

- The **name-mapper option** allows you to specify a name-mapping function that translates the names of C entities imported from the container, as described in Chapter 3, Name Mapping."

For files, the default value is `minimal-name-mapping-with-structure-prefix`. For structures and unions, the default value is the value of the name-mapper option for the enclosing container.

- The **prefix option** allows you to specify a prefix that is added to the front of the translated names of C entities imported from the container, as described in Chapter 3, Name Mapping."

  For files, the default value is the empty string. For structures and unions, the default value is the value of the prefix option for the enclosing container.

- The **rename option** allows you to specify explicit renamings for C entities imported from a container, as described in Chapter 3, "Name Mapping."

  The default value of this option is an empty list.

- The **type option** allows you to specify explicit type mappings for the C entities imported from a container, as described in Chapter 5, "Type Mapping." Type options contain a list of renamings, mapping string literals that represent C type names to symbols that specify the corresponding Apple Dylan class.

  The default value for this option is determined by the default type-mapping behavior.

Constant Options

The constant options allow you to specify additional information about how to import a particular C constant. You can use these options to override the options of an enclosing container.

Here are the syntax descriptions for the constant options:

Constant option	Values
`inline:`	`#t` | `#f`
`value:`	*expression*

- The **inline option** allows you to specify whether a particular constant is imported as a runtime object or as inline code. A value of `#t` indicates that

the constant should be imported as inline code, and a value of `#f` indicates that it should be imported as a runtime object.

The default value for this option is the value of the inline-constants option of the container that encloses the constant.

- The **value option** allows you to specify the Apple Dylan value of an imported constant. Value options allow you to specify an expression that indicates the Apple Dylan value.

  You can use this option when the translated C value would not be appropriate. If you do not specify this option, the translated C value is used.

Variable Options

The variable options allow you to specify additional information about how to import a particular C variable or structure/union member. You can use these options to override the options of an enclosing container.

Here are the syntax descriptions for the variable options:

Variable option	Values
`read-only:`	`#t` | `#f`
`seal:`	`sealed` | `open` | `inline`
`getter:`	*variable-name*
`setter:`	*variable-name* | `#f`
`type:`	*variable-name*

- The **read-only option** allows you to specify whether to create a setter function for a particular imported variable. A value of `#t` indicates that the variable should be imported as a getter function only, and a value of `#f` indicates that it should be imported as a getter/setter pair of functions.

The default value for this option is the value of the read-only option of the container that encloses the variable.

- The **seal option** allows you to specify the dynamism of the functions for a particular imported variable. The value `open` indicates that functions should be imported as open generic function methods. The value `sealed` indicates that they should be sealed generic function methods. The value `inline` indicates that they should be bare methods and that calls to them should be replaced with inline code.

  The default value for this option is the value of the seal-functions option of the container that encloses the variable.

- The **getter option** allows you to explicitly rename the getter function for a particular imported variable.

  The default value for this option is the name created by the name-mapping process.

- The **setter option** allows you to specify information about the setter function for a particular imported variable. Setter options can have the syntax

  `setter: #f`

  which indicates that no setter function should be created for the variable. They can also have the syntax

  `setter:` *variable-name*

  where *variable-name* indicates the name for the setter function for the variable.

  For read-only variables, the default value of this option is `#f`. For others, the default value is the name created by the name-mapping process.

- The **type option** allows you to specify an explicit type mapping for an imported variable. You provide a symbol that indicates the Apple Dylan class that this variable's type is type-mapped to; the getter function for the imported variable returns an object of the indicated class, and the setter function (if any) expects an object of the indicated class as its first argument.

  The default value for this option is determined by the type-mapping process, as described in Chapter 5, "Type Mapping."

Function Options

The function options allow you to specify additional information about how to import a specific C function. You can also use these options to modify callout functions and callback macros. As with constant and variable, you can use these options to override the options of an enclosing container.

Here are the syntax descriptions for the function options:

Function option	Syntax
`seal:`	`sealed` | `open` | `inline`
`result-type:`	*variable-name*
`argument-type:`	`{` **(***var-name* | *string* | *integer***)** `=>` *var-name* `}`
`input-argument:`	*variable-name* | *string* | *integer*
`output-argument:`	*variable-name* | *string* | *integer*
`input-output-argument:`	*variable-name* | *string* | *integer*

- The **seal option** allows you to specify the dynamism of an imported function. The value `open` indicates that the function should be imported as an open generic function method. The value `sealed` indicates that it should be a sealed generic function method. The value `inline` indicates that it should be a bare method, and that calls to the function should be replaced with inline code.
- The **result-type option** allows you to specify an explicit type mapping for the function result of an imported function. You provide a symbol specifying the name of the Apple Dylan class to map the result to.

  The default value for this option is determined by the type-mapping process.
- The **argument-type option** allows you to specify explicit type mappings for the arguments of an imported function. In an argument-type option, you

specify a parameter of the C function and an Apple Dylan class for the parameter to be mapped to.

This option can be used multiple times.

The default value for this option is determined by the type-mapping process, as described in Chapter 5, “Type Mapping.”.

- The **input-argument option** allows you to specify that a particular parameter to an imported C function is used only as input. You can specify the parameter using the parameter’s name as a string literal or a variable name, or the number of the parameter. You can have multiple input-argument options in a single function clause.

  This option can be used multiple times.

  This option is the default for all parameters. The next two options override this option for particular parameters.

- The **output-argument option** allows you to specify that a particular parameter to an imported C function is a pointer to a location used only as output. This option eliminates the indicated parameter from the imported function’s parameter list; instead the value returned by the C function in this location is added as an additional result returned by the imported function.

  This option can be used multiple times.

- The **input-output-argument option** allows you to specify that a particular parameter to a C function is a pointer to a location used as input and as output. This option keeps the indicated parameter in the imported function’s parameter list, and also adds the return value as a function result.

  This option can be used multiple times.

For output-argument and input-output-argument options, Apple Dylan automatically removes one level of indirection for the indicated parameters, as shown in Chapter 4, “Cross-Language Calls,” and Chapter 5, “Type Mapping.”

Callback Options

In addition to the function options described in the previous section, you can also use the universal option in callback clauses.

Here is the syntax descriptions for this option:

Callback option	Syntax
`universal:`	*`string`*

- In callback clauses, you can specify a normal pointer-to-function type, or a Universal Procedure Pointer type. If you specify a Universal Procedure Pointer type in your callback clause, you can use the **universal option** to specify the underlying pointer-to-function type. Apple Dylan uses this information to determine the types of the arguments and the function result.

  If you specify a normal pointer-to-function type in your callback clause, this option is ignored.

CHAPTER 3

Name Mapping

Contents

About Name Mapping 55
Using Name Mapping 56
  Specifying Explicit Name Mapping 56
    Using the Import Option of a Container Clause 56
    Using the Rename Option of a Container Clause 57
    Using the Arrow Option of a Clause 58
    Using the Getter and Setter Options of a Variable Clause 60
  Adding Prefixes to Imported Names 60
  Using the Name-Mapping Functions 62
Name Mapping Reference 64
    The MacApp-to-Dylan Function 64
    The C-to-Dylan Function 65
    The minimal-name-mapping Function 66
    The minimal-name-mapping-with-structure-prefix Function 66
    The identity-name-mapping Function 67

This chapter discusses how Apple Dylan renames C entities during interface importation, and how you can modify the renaming process.

About Name Mapping

There are a number of reasons why you might want to rename C entities as you import them into your Apple Dylan program, for example:

- **To prevent name conflicts.** Apple Dylan and C use different name-scoping rules. Entities defined in a C header file which have perfectly valid names under the scoping rules of C may come into conflict when imported into an Apple Dylan module.

  For example, C is case-sensitive; two names identical except for case are considered distinct. Imported into case-insensitive Apple Dylan, however, the same two names come into conflict.As another example, in a C header file, the members of a structure type have separate name space from the rest of the header file. No such distinction exists in Apple Dylan.

- **To address differences in naming conventions.** C programs use different naming conventions than Apple Dylan programs.

  For example, multiple words in a variable name are commonly delimited by inner capitals in C programs (for example, `hereAndThere`), whereas Dylan programming style uses dashes (`here-and-there`).

To resolve these and other naming issues, Apple Dylan allows you to rename C entities as you import them into your Apple Dylan program. The name-translation process is called **name mapping.**

Name mapping comes in several varieties:

- **Default name mapping** is provided for you automatically whenever you import C entities with an interface definition. Default renaming makes minor changes to imported names to reflect some of the basic naming conventions of Apple Dylan. For example, the *$* character is added to the front of constant names, angle brackets < and > are added around type names, and structure names are added before imported member names.

- **Automatic name mapping** allows you to choose from a variety of other algorithms that translate different C naming conventions to Apple Dylan naming conventions.

- **Prefix name mapping** allows you to add a prefix to the names of entities imported from a specified container. These prefixes are added before any automatic name mapping occurs.
- **Explicit name mapping** allows you complete control over the imported name of any C entity. Explicit name mapping overrides all other forms of name mapping.

The next section shows how you use interface definitions to specify the different kinds of name mapping.

Using Name Mapping

This section provides examples of name mapping. Each section shows how to use different interface definition clauses and options to control name mapping.

Specifying Explicit Name Mapping

Interface definitions allow you to provide an **explicit name mapping** for any imported C entity. There are a number of places in an interface definition where you can specify explicit name mappings, including

- the import option of a container clause
- the rename option of a container clause
- the arrow (=>) option of any clause (except a file clause)
- the getter and setter options of a variable clause

These options are discussed individually in the next four sections.

Using the Import Option of a Container Clause

When you import C entities from a C container, you can use the import option to specify explicit name mappings. If you're already using an import option to selectively import C entities, this method provides a convenient way to rename C entities without having to add more options or clauses to your interface definition.

Here is an example:

```
define interface
    #include “SampleHeader.h”,
        import: { "sqrt" => square-root,
                  "expt" => exponent,
                  "divide",
                  "remainder" };
end interface;
```

This example imports four entities from a header file. Two of those entities are explicitly renamed using the import option; the other two are not. Notice that C entity names are literal strings; they are surrounded by quotes. The Apple Dylan names are variable names; no quotes are required.

Using the Rename Option of a Container Clause

Using an import option to rename specific entities is convenient if you are already using the import option to designate which entities to import.

However, if you are importing all entities from a container or all but specifically excluded ones, then you are probably not using an import option. For this situation, Apple Dylan provides the rename container option. Here is an example:

```
define interface
    #include "SampleHeader.h",
        import: all, // the default
        rename: { "sqrt" => square-root,
                  "expt" => exponent };
end interface;
```

This example imports all of the C entities from the header file `SampleHeader.h` and explicitly renames two of them.

You can also use the import and rename container options to rename members of imported structures and unions. You may want to do this, for example, when importing structure type definitions that contain internal structure of anonymous type.

Consider this C structure type definition:

```
struct doubleStruct {
    int i;
    struct {
        int i;
        int j;
    } s;
};
```

In C, the two members named `i` are in different scopes; however, when you import this structure type into Apple Dylan, both of these members enter the same name space. Therefore, you must rename at least one of these members.

The following example shows how you can use the rename option to resolve this naming conflict:

```
define interface
    #include "SampleHeader.h",
        import: { "doubleStruct" };
    struct "doubleStruct",
        rename: { "i" => "outer-i",
                  "s::i" => "inner-i",
                  "s::j" => "inner-j" };
end interface;
```

This example resolves the name conflict by renaming the `i` member in the outer structure `outer-i` and the `i` member in the inner structure `inner-i`.

This example shows the use of the `::` notation to distinguish member names by the name of their containers. The notation `"s::i"` specifies the `i` member that is inside the `s` member of the `doubleStruct` structure type. Similarly, the notation `"s::j"` specifies the `j` member that is inside the `s` member of the `doubleStruct` structure type.

Using the Arrow Option of a Clause

You can also rename a specific C entity in its own clause, rather than in the clause of its container.

For example, consider this interface definition:

```
define interface
    #include "SampleHeader.h";

    function "sqrt" => square-root;

    struct "Point" => <position>;
end interface;
```

This interface definition uses a file clause to import all of the entities from the file `SampleHeader.h`. It uses a separate function clause to explicitly rename an imported function, and it uses a separate structure clause to explicitly rename an imported structure type.

Since you can rename imported entities in more than one way, it is possible to specify multiple name mappings for the same entity. For example:

```
define interface
    #include "SampleHeader.h",
        rename: { "sqrt" => root };

    function "sqrt" => square-root;
end interface;
```

In this example, the C entity `sqrt` is renamed `root` in the file clause and `square-root` in the function clause. Apple Dylan resolves this conflict using the general rule that more specific clauses override less specific clauses. As the specified function clause is more specific than the file clause, the function is imported under the name `square-root`.

Using the Getter and Setter Options of a Variable Clause

Variable clauses allow you to rename imported variables in two different ways. As with any interface definition clause, you can use the arrow option, as shown here:

```
define interface
    #include "SampleHeader.h";

    variable "global" => get-global-variable;
end interface;
```

This interface definition imports the `global` variable from `SampleHeader.h` as a getter function named `get-global-variable` and a setter function named `get-global-variable-setter`.

You can explicitly rename these two functions, independently, using the getter and setter options. For example:

```
define interface
    #include "SampleHeader.h";

    variable "global",
        getter: get-c-global,
        setter: set-c-global;
end interface;
```

As described in the previous chapter, you can also specify the value `#f` for the setter option, to prevent Apple Dylan from creating a setter function.

Adding Prefixes to Imported Names

Another way to control name mapping is to add prefixes to imported names. Adding prefixes can help you avoid naming conflicts, and can also help you remember which Apple Dylan entities have been imported from C, which C header file an entity was imported from, and so on.

The prefix option is a container option— it allows you to specify a default prefix for the names of all entities imported from a container. For example:

```
define interface
    #include "SampleHeader.h",
        prefix: "imported-",
        import: { "sqrt", "expt" };
end interface
```

This definition imports two C functions from the file `SampleHeader.h`. The prefix option specifies a prefix to be added to the imported names: the `sqrt` function becomes `imported-sqrt`, and the `expt` function becomes `imported-expt`.

The explicit name mappings discussed earlier in this chapter always override other kinds of name mapping—including prefix name mapping. For example:

```
define interface
    #include "SampleHeader.h",
        prefix: "imported-",
        import: { "sqrt", "expt" => exponent };
end interface
```

In this example, the `sqrt` function is imported as `imported-sqrt`, but the `expt` function is imported as `exponent`. Prefixes do not affect entities with explicit name mappings.

Prefixes are applied to the names of imported entities before the default name mapping behavior is applied (or before any of the automatic name-mapping functions described later in this chapter). For example, the default name-mapping algorithm adds structure names and *$* characters before member names. Consider this definition:

```
define interface
    #include "SampleHeader.h";

    struct "Point" => <my-point>,
        import: { "h", "v" };
end interface
```

By default, the `h` and `v` members of the `Point` structure are imported as `Point$h` and `Point$v`.

If you specify a prefix for these imported names, the prefix is added before the default name mapping occurs. For example:

```
define interface
    #include "SampleHeader.h";

    struct “Point” => <my-point>,
        prefix: “slot-”,
        import: { “h”, “v” };
end interface
```

In this definition, Apple Dylan first adds the prefix to the slot names, yielding `slot-h` and `slot-y`, and then performs the default name mapping, which yields `Point$slot-x` and `Point$slot-y`.

Using the Name-Mapping Functions

The **name-mapper** option is another container option that affects how the imported entities within a single container are renamed. To use this option, you specify a **name-mapping function** that automatically adjusts the names of imported entities.

Here is an simple interface definition that includes a name-mapper option:

```
define interface
    #include "SampleHeader.h",
        name-mapper: C-to-Dylan,
        import: { "SampleFunction", "SampleStruct" };
end interface
```

This interface definition uses the name-mapper option to specify a name-mapping function to apply to all the entities imported from `SampleHeader.h`. The name-mapping function specified is `C-to-Dylan`, which is one of the **built-in name-mapping functions** that Apple Dylan provides.

The `C-to-Dylan` function is intended to map names that follow the standard C naming conventions to names that follow the standard Apple Dylan naming conventions. In the above example, the name `SampleFunction` becomes `Sample-Function` and the name `SampleStruct` becomes `<Sample-Struct>`.

The name-mapper option applies recursively down containers. For example:

```
define interface
    #include "SampleHeader.h",
        name-mapper: identity-name-mapping;

     struct "Point",
        import: { "h", "v" };
end interface
```

This interface definition specifies the `identity-name-mapping` function as the name-mapping function for the entire header file. This function imports entities without changing their names, so in the above example, the structure `Point` would be imported as an Apple Dylan class named `Point` and the h and v members would be imported as Apple Dylan functions `h` (and `h-setter`) and v (and `v-setter`).

You can override the name-mapper option for an outer container using the name-mapper option of an inner container. For example:

```
define interface
    #include "SampleHeader.h",
        name-mapper: identity-name-mapping;

     struct "Point",
        name-mapper: C-to-Dylan,
        import: { "h", "v" };
end interface
```

In this example, the `Point` structure is still imported as `Point`, but the members are imported as `Point$h` and `Point$v` (and `Point$h-setter` and `Point$v-setter`).

Here is a complete list of the name-mapping functions provided by Apple Dylan:

- `MacApp-to-Dylan`, which maps MacApp's naming conventions to standard Apple Dylan naming conventions, and adds the prefix `get-` to the front of imported structure and union member names.
- `C-to-Dylan`, which maps standard C naming conventions to standard Apple Dylan naming conventions, and also adds the prefix `get-` to the front of imported structure and union member names
- `minimal-name-mapping`, which unlike `C-to-Dylan`, does not insert hyphens or affect member names.

- `minimal-name-mapping-with-structure-prefix` inserts the structure (or union) name in front of its member names. This is the default name-mapping function.
- `identity-name-mapping`, which performs no name mapping.

For complete descriptions of these functions, see the function reference descriptions in the next section.

Name Mapping Reference

This section describes the five name-mapping functions provided by Apple Dylan.

The MacApp-to-Dylan Function

This name-mapping function translates names from MacApp's naming conventions to Apple Dylan naming conventions.

For all imported entities, this name-mapping function inserts dashes where the original name has a break in case. For example, `SampleName` becomes `Sample-Name`, and `gestaltTEAttr` becomes `gestalt-TE-Attr`.

Other translations performed are based on the type of C entity being imported:

- **Types.** The initial `T`, if any, is removed from the name and the result is surrounded by angle brackets.
- **Constants.** The initial `k`, if any, is removed from the name and an initial dollar sign is added.
- **Members.** The initial `f`, if any, is removed from the name and the prefix `get-` is added.

The following table shows examples of `MacApp-to-Dylan` name mapping:

C entity name	Apple Dylan name
`SampleFunction`	`Sample-Function`
`TSampleType`	`<Sample-Type>`
`kSampleConstant`	`$Sample-Constant`
`fSampleField`	`get-Sample-Field`

The C-to-Dylan Function

This name-mapping function translates names from standard C naming conventions to standard Apple Dylan naming conventions.

For all imported entities, this name-mapping function inserts dashes where the original name has a break in case. Other translations are performed based on the type of entity being imported:

- **Types.** The result is surrounded by angle brackets.
- **Constants.** An initial dollar sign is added.
- **Members.** The prefix `get-` is added.

The following table shows some illustrative examples of `C-to-Dylan` name mapping:

C entity name	Apple Dylan name
`SampleFunction`	`SampleFunction`
`SampleType`	`<Sample-Type>`
`SampleConstant`	`$Sample-Constant`
`sampleMember`	`get-sample-Member`

The minimal-name-mapping Function

This name-mapping function performs minimal name translation. The only two translations performed are based on the type of entity being imported:

- **Types.** The name is surrounded by angle brackets.
- **Constants.** An initial dollar sign is added.

The following table shows some illustrative examples of `minimal-name-mapping` name mapping:

C entity name	Apple Dylan name
`SampleFunction`	`SampleFunction`
`SampleType`	`<SampleType>`
`SampleConstant`	`$SampleConstant`
`sampleMember`	`sampleMember`

The minimal-name-mapping-with-structure-prefix Function

This name-mapping function is the default name-mapping function. It performs minimal name translation, but it also prefixes the name of structures and unions to their fields. Specifically, this function performs these translations:

- **Types.** The name is surrounded by angle brackets.
- **Constants.** An initial dollar sign is added.
- **Members.** The name of the member's structure or union, followed by a dollar sign, is added as a prefix.

The following table shows some illustrative examples of the name-mapping function `minimal-name-mapping-with-structure-prefix`:

C entity name	Apple Dylan name
`SampleFunction`	`SampleFunction`
`SampleType`	`<SampleType>`
`SampleConstant`	`$SampleConstant`
`sampleMember`	`sampleContainer$sampleMember`

The identity-name-mapping Function

This name-mapping function performs no name translation.

The following table shows examples of the `identity-name-mapping` function:

C entity name	Apple Dylan name
`SampleFunction`	`SampleFunction`
`SampleType`	`SampleType`
`SampleConstant`	`SampleConstant`
`sampleMember`	`sampleMember`

Name Mapping

CHAPTER 4

Cross-Language Calls

Contents

About Cross-Language Calls 71
Types of Cross-Language Calls 71
Overview of a Cross-Language Call 72
Using Cross-Language Calls 74
Calling Imported Functions 75
Calling C Library Functions Using Function Pointers 77
Calling Apple Dylan Functions From C Libraries 79

This chapter examines the mechanics of making cross-language calls with Apple Dylan. In particular, this chapter includes examples that illustrate the various aspects of a cross-language call, including: interface importation, type mapping, additional necessary preparations, invoking cross-language calls, passing values between languages, translating calling sequences, and so on.

Type mapping and value translation are key elements of cross-language calls, and so they are included in the discussions in this chapter. However, these processes are complex and are not fully described here. You can find the complete discussion of type mapping and value translation in the next chapter.

About Cross-Language Calls

One of the primary motivations behind the Apple Dylan language extensions is to allow an Apple Dylan program to make **cross-language calls**—in other words, to allow Apple Dylan functions to call C library functions and to allow C library code to call Apple Dylan. Most of the facilities included with these language extensions (such as interface definitions, name mapping, type mapping, and access paths) provide support for cross-language calls.

Types of Cross-Language Calls

There are five basic categories of cross-language calls:

- **Imported functions** provide a simple mechanism for calling C library functions from Apple Dylan. You import a function from a C header file using an interface definition and Apple Dylan creates a corresponding Apple Dylan function for you. You call this function as you call any Apple Dylan function. This imported function performs the necessary translations and calls the C library function for you, using one of the access paths described in Chapter 6 to find the C library function at runtime. “Calling Imported Functions” on page 75 gives examples of cross-language calls using imported functions.
- **Imported variables** are Apple Dylan functions that allow you to read and write the value of C library variables. When you import a variable from a C header file, Apple Dylan creates a getter/setter pair of functions that allow you to read and to write the value of the variable. These functions use an

access path to find the C library variable; therefore, to import a variable, you must use an access path that provides access to variables.

- **Imported members** are Apple Dylan functions that allow you to read and write the value of members of C structures and unions. When you import a structure type from a C header file, Apple Dylan creates a getter/setter pair of functions that allow you to read and to write the values of the members of instances of that type. See Chapter 5, "Type Mapping," for examples of importing and accessing structure members.
- **Callout functions** allow you to call C functions using function pointers. A callout function is an Apple Dylan function that takes a pointer to a C function and values to pass to the C function. The callout function uses the function pointer to call the C function for you, translating and passing the arguments you specified. "Calling C Library Functions Using Function Pointers" on page 77 gives examples that use callout functions and function pointers to call C library functions.
- **Callback macros** allow you to call Apple Dylan functions from C library code. A callback macro is an Apple Dylan macro that you use to create a callback function; a **callback function** is a special kind of C-compatible Apple Dylan function. When you pass a callback function to a C library function, that C function can use the callback function as a standard C function pointer to call your Apple Dylan callback function. "Calling Apple Dylan Functions From C Libraries" on page 79 shows how to use callback macros and callback functions to call Apple Dylan from a C library.

In addition to these types of cross-language calls, you can also make low-level (assembly-language) cross-language calls using the information in Chapter 7, "Low-Level Facilities."

Overview of a Cross-Language Call

This section discusses the dynamics of cross-language calls between Apple Dylan and C. For simplicity, this discussion is limited to cross-language calls using imported functions. You can find analogous information for the other types of cross-language calls in the "Using Cross-Language Calls" sections later in this chapter.

You import functions from C header files using interface definitions. Importing a function definition provides Apple Dylan with information necessary to make a cross-language call to the function. The rest of this section discusses the

kind of information you provide when importing C functions and how Apple Dylan uses that information to perform cross-language calls.

When you import a C function, Apple Dylan creates an Apple Dylan function that represents the C library function. The C header file specifies information about the original C function; your interface definition specifies information about how that C function should be imported.

Here are some examples of options available to you when importing a function:

- You can rename the function. For example, you may want the imported function name to follow Apple Dylan naming conventions.
- You can specify the dynamism of the imported function. For example, you can specify that the function be imported as an open generic function method, a sealed generic function method, or a bare method implemented as inline code.
- You can adjust the parameter list and the result values to reflect differences between the way C functions and Apple Dylan functions pass values. C functions can have input parameters, output parameters, input-output parameters, and a single function result, while Apple Dylan functions use parameters for input and function results for output. If you specify how the C function uses its parameters, Apple Dylan automatically makes the appropriate modifications to the parameter list and the result values of the corresponding Apple Dylan function.
- You can explicitly specify type mappings for the parameters and function result—that is, the Apple Dylan class associated with each of the C function's parameters and function result.

You do not have to specify all of this information each time you import a function—Apple Dylan provides default behavior that handles many imported functions without requiring any customization.

Once you've created the imported function, you call it as you would call any Apple Dylan function. The imported function then performs the cross-language call. In particular, the function:

1. Translates the argument values from Apple Dylan objects to corresponding C values.
2. Adjusts the argument list. If the parameter list was modified when the function was imported, the imported function makes the appropriate adjustments to match the C function's specification.

3. Calls the C function, using the appropriate access path and handling the differences in runtime environments, calling sequences, and so on.
4. Translates the value returned by the C function as well as the values of any output arguments.
5. Returns the translated values as the function result.

As you can see, much of the cross-language call process involves translating values between Apple Dylan and C. This value translation process is discussed fully in Chapter 5, "Type Mapping."

The "Using Cross-Language Calls" sections, which follow, provide examples of the different kinds of cross-language calls—how you set them up, how you make the calls, and what Apple Dylan does for you.

Using Cross-Language Calls

In the following sections, you'll find examples of cross-language calls. For each example, these sections discuss

- what to specify in your interface definition
- additional preparation you must perform, if any
- how you invoke the cross-language call
- the process Apple Dylan follows to implement the call
- the results of the cross-language call

These examples do not address the access path mechanisms used by Apple Dylan to locate the C library functions and C library variables that correspond to imported function and variable definitions. That discussion is left until Chapter 6, "Access Paths." Keep in mind that you may need to modify the examples in this chapter depending upon which access path you are using.

Also, type mapping and value translation are mentioned in these examples, but they are not fully discussed until Chapter 5, "Type Mapping." That chapter also includes examples of cross-language calls using imported structure members.

Calling Imported Functions

Before you can call a C library function from your Apple Dylan program, you must perform the necessary preparations. First, you must create an Apple Dylan interface to the function by compiling an interface definition that imports the definition of the function from a C header file.

As an example, imagine you have a C header file named `SampleHeader.h`, which includes this function declaration:

```
void Withdraw (long amount, long *balance, short *overdrawn);
```

Here is a simple interface definition you could use to import the function:

```
define interface
    #include "SampleHeader.h",
        import: { "Withdraw" };
end interface;
```

This interface definition includes a single file clause, which specifies a single C entity to be imported—the C function named `Withdraw`.

Adding a function clause to this interface definition allows you more control over how the function is imported. For example, consider the following interface definition:

```
define interface
    #include "SampleHeader.h",
        import: { "Withdraw" };

    function "Withdraw" => take-money-out,
        seal: sealed,
        input-argument: "amount",
        input-output-argument: "balance",
        output-argument: "overdrawn",
        argument-type: { "overdrawn" => <Boolean> };
end interface;
```

This interface definition imports the function `Withdraw`, but it also specifies additional information about the Apple Dylan version of the function. In particular, it specifies

- the imported function should be renamed `take-money-out`
- the imported function should be a sealed method of a generic function
- the `amount` parameter is an input parameter
- the `balance` parameter is an input-output parameter
- the `overdrawn` parameter is an output parameter
- the `overdrawn` parameter should be mapped to the Apple Dylan class `<Boolean>`

This interface definition results in an imported Apple Dylan function named `take-money-out`:

- This function expects two `<integer>` arguments—the amount and the balance—corresponding to the two `long` arguments expected as input by the C function. (Actually, the `balance` parameter of the C function is declared as type `long*`, because C uses pointers for parameters whose values can change. Specifying the input-output-argument option for this parameter allows Apple Dylan to adjust for this extra indirection. Therefore, the `balance` parameter is imported as if it were declared to be of type `long`.)
- This function returns two function result values —an `<integer>` object indicating the new balance and a `<Boolean>` object indicating whether the account is overdrawn.

The imported `take-money-out` function provides access to the original C function. You call the imported function as you would any Apple Dylan function, for example:

```
let (new-balance, in-the-red) = take-money-out (bills, old-balance);
```

The imported function, named `take-money-out`, requires two arguments—corresponding to the two input arguments of the C function. When you execute this function call, Apple Dylan translates the values of these arguments from Apple Dylan `<integer>` objects to corresponding C `long` values.

Apple Dylan then uses whichever access path you choose to locate the C library function corresponding to the `Withdraw` function definition. Apple Dylan

calls this C library function, making the appropriate adjustments for the differences in the language calling sequences, and so on.

When the C library function returns, Apple Dylan translates the values returned in the output arguments from C values to corresponding Apple Dylan objects, and returns those objects as the multiple return values of the imported function.

Calling C Library Functions Using Function Pointers

In the world of C programming, function pointers are often used to make function calls. In many situations, a function pointer is the only mechanism available to access a particular function.

A common example occurs when a C library function returns a function pointer as the function result. To call the indicated function, you must use the function pointer provided. As an example, consider the header file in Listing 4-1.

Listing 4-1 Header file with function pointer type and functions

```
/*** FunctionPointer.h ***/

typedef long (*processor) (long, long);

processor getTodaysProcessor(void);
```

This header file defines two C entities:

- A pointer-to-function type, named `processor`. This pointer type points to functions that require two `long` arguments and return one `long` result.
- A function named `getTodaysProcessor`. This function returns a processor function pointer as its function result.

You can import both C entities using this interface definition:

```
define interface
    #include "FunctionPointer.h",
        name-mapper: C-To-Dylan;
```

```
    callout "processor" => call-processor;
end interface;
```

This interface definition uses a file clause to import the two C entities from the header file and also uses a callout clause to create a callout function for the imported function-pointer type. The resulting Apple Dylan entities are:

- A class named `<processor>`, whose instances are Apple Dylan objects that store pointers to C functions.
- A function named `get-todays-processor`, which calls the C function `getTodaysProcessor`.
- A **callout function** named `call-processor`. This function takes three arguments:
  - an object of class `<processor>`, which is a pointer to a C `processor` function
  - two objects of class `<integer>`. which correspond to the arguments expected by `processor` functions

  The callout function applies the function pointed to by the `<processor>` object to the two `<integer>` arguments, and returns a value of type `<integer>`.

To secure a pointer to a C `processor` function, you call the imported `get-todays-processor` function. For example:

```
define variable processor-du-jour = get-todays-processor();
```

The imported `get-todays-processor` function calls the C function `getTodaysProcessor`, which returns a C function pointer of type `processor`. Apple Dylan converts this function pointer from its C format into Apple Dylan object format and returns it as the result of the `get-todays-processor` function.

Although the returned value is an Apple Dylan object, it retains all of the information in the original C function pointer—that is, all of the information necessary to locate the indicated C function.

Once you've secured a pointer to a C function, you can make a cross-language call to that function using the callout function `call-processor`. You pass these arguments to the callout function:

- the pointer to the C `processor` function
- the arguments you want to pass to the `processor` function

Here is an example:

```
define variable result = call-processor(processor-du-jour, 1, 2);
```

The `call-processor` callout function translates its first argument from an Apple Dylan object back into a C function pointer. Then it calls the C function pointed to by that function pointer, passing that function the translated values of the other two arguments. The C function returns a value of type `long`, which Apple Dylan converts to an object of class `<integer>`. This object is returned as the result of the `make-processor-call` callout function.

Calling Apple Dylan Functions From C Libraries

You've seen how to call C library functions from your Apple Dylan program. This section discusses the opposite ability—calling Apple Dylan code from a C library function.

Apple Dylan provides this functionality via a **callback** mechanism, in which you pass a **callback function** to an imported C function that expects a function pointer argument. During that cross-language call, Apple Dylan translates the callback function into a valid C function pointer, and passes it to the C function. The C function can then use that pointer to call your callback function.

To use the callback mechanism, then, you must have a C library function that accepts a function pointer as an argument and uses that function pointer to call the indicated function. Consider the C header file shown in Listing 4-2.

Listing 4-2 A header file using a callback function

```
/*** Callbacks.h ***/

typedef long (*processor) (long, long);

long applyProcessor(processor f, long a, long b);
```

This header file defines two C entities:

- a function pointer type, named `processor`, that points to functions with two `long` arguments and a `long` return value
- a function named `applyProcessor` that takes three arguments (a `processor` function pointer and two values of type `long`) and returns a value of type `long`.

Imagine that the `applyProcessor` function is implemented with the following C code:

```
long applyProcessor (processor f, long a, long b) {

    return ((*f) (a, b));
}
```

This simple function takes a function pointer and two arguments of type `long`, applies the indicated function to the `long` arguments, and returns the result.

You can import the C header shown in Listing 4-2 using this interface definition:

```
define interface
    #include "Callbacks.h",
         name-mapper: C-to-Dylan;

    callback "processor" => make-Dylan-processor;
end interface;
```

This interface definition imports the two C entities from the header file and also creates a callback macro. Executing this interface definition creates three Apple Dylan entities:

- A class named `<processor>`, which is used to represent pointers to `processor` functions.
- A function named `apply-processor`, which applies a function to two integers and returns a single integer.
- A **callback macro** named `make-Dylan-processor`. This macro allows you to create a callback function.

Here is how you use the callback macro to create a callback function.

```
define variable my-callback =
    make-Dylan-processor (a, b)
        (a * a) + (b * b);
    end;
```

Note
This release does not support this syntax. Instead, you must write:

```
define variable my-callback =
    make-Dylan-processor(a(b),
    begin
       (a * a) + (b * b);
    end;
```

With more than two arguments, you would write:

```
define variable my-callback =
    make-Dylan-processor(a(b, c, d),
    begin
       a + b + c + d;
        end);
```

With just one argument, you would write:

```
define variable my-callback =
    make-Dylan-processor(a(),
    begin
       a * a;
    end;
```

With no arguments, you would write:

```
define variable my-callback =
    make-Dylan-processor(#(),
    begin
       105;
    end;  ◆
```

This variable definition calls the `make-Dylan-processor` callback macro, which creates a callback function. The body of the macro is used as the body of the callback function. In this example, the callback function takes two arguments and calculates the sum of their squares.

The `make-Dylan-processor` callback macro returns as its result a callback function as an object of class `<processor>`.

To use this callback function, you pass it to a C function:

```
define variable result = apply-processor(my-callback, 1, 2);
```

Executing this variable definition results in these steps:

1. The `apply-processor` imported function converts its arguments from Apple Dylan objects to corresponding C values:
   - The `my-callback` object is converted to a C function pointer of type `processor`. This pointer points to the callback function.
   - The integer objects representing 1 and 2 are converted to C values of type `long`.
2. The imported function passes the function pointer and the `long` values to the C function `applyProcessor`.
3. The C function `applyProcessor` applies the function pointed to by its first argument to the values provided in its second and third arguments.
4. Since the function pointed to by that first argument is the callback function, control is passed back to Apple Dylan, which converts the `long` values to objects of type `<integer>`, and passes them to your callback function.
5. The callback function evaluates the expression `(1 * 1) + (2 * 2)` and returns the resulting value of 5.
6. Apple Dylan converts the integer object 5 into a C value of type `long` and passes it back to the C function `applyProcessor`.
7. The C function `applyProcessor` finishes executing and returns the C value 5 to Apple Dylan, which converts it to an `<integer>` object and returns that object as the result of the `apply-processor` function.
8. The result is bound to the variable name `result`.

The next chapter, “Type Mapping,” details the mechanism of passing values between Apple Dylan and C-compatible library code during cross-language calls.

CHAPTER 5

Type Mapping

Contents

About Type Mapping 87
  About Importing Type Definitions 87
  About Type Mappings 89
    Implicit Type Mapping 90
    By-Value Types 91
    Untyped-Pointer Types 92
    Statically-Typed-Pointer Types 93
    Explicit Type Mappings 93
  About Value Translation 94
Using Type Mapping 95
  Importing Structures 95
    Arrays as Structure Members 97
  Importing Pointer-To-Structure Types 98
  Importing Pointer-To-Function Types 99
  Using Explicit Type Mapping 100
  Type Mapping Strings 100
  Importing and Exporting Values 101
Type Mapping Reference 105
  Pointer Classes 105
      The <machine-pointer> Class 105
      The <statically-typed-pointer> Class 106
      The <C-string> Class 108
      The <Pascal-string> Class 110
  Structure-Related Functions 111
      The structure-size Function 111
      The destroy Open Generic Function 112

Functions for Importing and Exporting Values 113
The import-value Open Generic Function 114
The export-value Open Generic Function 115
The export-temporary-value Open Generic Function 116

This chapter provides complete description of the type-mapping behavior provided by Apple Dylan during interface importation and the value-translating features provided by Apple Dylan during cross-language calls. This chapter also discusses how you can override these default behaviors.

About Type Mapping

This chapter discusses the type-mapping facilities provided by Apple Dylan to facilitate the flow of data between languages during cross-language calls.

Using interface definitions, you can import types, such as structure or union types, from a C header file. **Importing type definitions** directly allows you to create Apple Dylan classes that can manipulate C data objects. See the next section for details.

Also at interface importation time, Apple Dylan creates a **type mapping** for each C type used by the imported interface. A type mapping is an association between of an Apple Dylan class and a C type. Type mappings are created for each parameter of every imported function, each function result, each imported variable, and each imported structure member or union member. See "About Type Mappings" on page 89 for more information.

During cross-language calls, Apple Dylan uses the type-mapping information to perform **value translation** between Apple Dylan objects and C values. For example, during a call to an imported function, each argument is translated from an object of an Apple Dylan class to a value of the corresponding C type. The C output arguments and the function result are translated from values of C types to instances of the corresponding Apple Dylan classes. See "About Value Translation" on page 94 for more information.

About Importing Type Definitions

Apple Dylan allows you to import types from C header files. Any type that has a name and is based on a struct, union, or function type can be imported. A type is based on another type if:

- the two types are the same
- the first type is a typedef that expands to a type based on the second type

- the first type is a pointer to a type based on the second type
- the first type is an array of a type based on the second type
- the first type is const or volatile plus a type based on the second type

For example, you can import:

- structure types and union types
- pointer-to-structure and pointer-to-union types
- pointer-to-function types
- multilevel pointer types (pointers to pointers to structures, and so on)

Importing a C type results in a corresponding Apple Dylan class. Apple Dylan creates a distinct pointer class for each structure or union type imported. By default, these classes are subclasses of the built-in `<statically-typed-pointer>`. class.

The `<statically-typed-pointer>` class has one slot, which contains a machine pointer. The subclasses of `<statically-typed-pointer>` that Apple Dylan creates to correspond to imported structure or union types inherit this slot.

As an example, imagine importing this definition from a header file:

```
struct Position {
    short x;
    short y;
} Position;
```

This structure type, named `Position`, is imported as a class named `<Position>`, which is a subclass of `<statically-typed-pointer>`. Instances of class `<Position>` contain pointers to C `Position` structures.

C's unions are identical to structures, except for the algorithm that computes the slot offsets. Apple Dylan supports unions; therefore, a pointer to a union is a statically typed pointer. You are responsible to know which variant of the union is used and only call the corresponding accessors (as in C).

See “Importing Structures” on page 95 for an example of importing a C structure type into Apple Dylan.

About Type Mappings

In addition to importing C types directly, Apple Dylan also allows you to import C entities that *use* C types. For example, when you import a C function, Apple Dylan creates a corresponding Apple Dylan function:

- The C function uses C types—that is, its parameters and its function result are defined to be particular C types.
- The Apple Dylan function uses a corresponding set of Apple Dylan types—classes that correspond to the C types used by the imported function.

A **type mapping** is an association between a C type (used by an imported C entity) and an Apple Dylan class (used by the corresponding Apple Dylan entity).

At interface importation time, Apple Dylan automatically creates a type mapping whenever a type is imported or used in a definition that is imported. (At runtime, Apple Dylan uses these type mappings to perform value translation—that is, to translate between Apple Dylan objects and C values.)

A type mapping is created for all

- arguments of imported functions
- function results of imported functions
- imported variables
- imported structure or union members
- imported structure and union types
- imported pointer to structure (and pointer to union) types
- imported pointer to function types
- imported multilevel pointer types (pointer to pointer types)

Apple Dylan provides default type mappings for you. This **implicit type mapping** occurs automatically whenever you import any of the above C entities (which all define or use C types).

Apple Dylan allows you to override the default behavior by specifying **explicit type mappings**. As an example, you can override the default type mapping for a particular parameter of an imported function by explicitly specifying a type mapping for that parameter.

The next few sections introduce the default (implicit) type-mapping behavior; explicit type mapping is introduced on page 93.

Implicit Type Mapping

An **implicit type mapping** is established whenever a type is used in an imported interface (and an explicit mapping for the type is not specified in the same `define interface` statement).

(Also, if a previously compiled `define interface` statement created a mapping for the type and the Apple Dylan class that is the target of that mapping is accessible in the current module, Apple Dylan uses that previously defined mapping rather than an implicit mapping.)

C types fall into two main categories: named types and anonymous types. The **named types** include

- **built-in types,** which include the integer and character types
- **typedef types,** defined and named with `typedef`
- **structure** (or **union**) **types,** defined with `struct` or `union`.

The **anonymous types** include

- **untyped-pointer types,** which include pointers to `void`, pointers to built-in types, and pointers to pointer types
- **statically-typed pointer types,** which include pointers to structures, pointers to unions, and pointers to functions (as well as multilevel pointers to structures, unions, and functions)

Apple Dylan has an implicit type mapping for every **built-in C type.** For example, if you import a function with a parameter of type `long`, Apple Dylan creates an implicit type mapping for that parameter, mapping the C type (`long`) to the Apple Dylan class `<integer>`.

As mentioned earlier in this chapter, Apple Dylan imports a **structure** (or a **union) type** as an Apple Dylan class (whose instances contain a pointer to a C structure or union). Apple Dylan uses this class in the implicit type mappings for any imported C entity that uses the structure or union type.

By default, Apple Dylan automatically maps **untyped-pointer types** (such as `void*`, `long*`, and `long**`) to the Apple Dylan class `<machine-pointer>`, as described in "Statically-Typed-Pointer Types" on page 93.

A **statically-typed pointer type** (for example, a pointer-to-structure type) is mapped to the same Apple Dylan class as the pointer type's base type (which would be the structure type). (This rule only applies to single-level pointers. In other words, a structure or union type maps to the same Dylan class as a single-level pointer to that structure or union type. Adding more than one level of pointer causes it to map to a distinct Dylan class. See "Statically-Typed-Pointer Types" on page 93 for more details.)

When type mapping a named C type defined in a **typedef definition**, the default behavior is to expand the typedef definition and find a type mapping for the expanded definition. (See "Importing Structures" on page 95 for an example.)

The next three sections describe the implicit type-mapping behavior for imported uses of these C types in more detail.

Note
You can specify an explicit type mapping in any of these situations. (See "Explicit Type Mappings" on page 93.) However, if no explicit type mapping is specified, and no type mapping exists from a previous interface definition, then Apple Dylan automatically provides an implicit type mapping. ◆

By-Value Types

Numbers, characters, strings, Booleans, and enumerated types (which are integers) are passed by value during cross-language calls. These are called the **by-value types.**

When an Apple Dylan program passes an instance of a by-value type between languages, it copies the value. C and Apple Dylan can use entirely different internal representations for objects that are instances of by-value types. Therefore, type mappings must be created for imported uses of these types (which, during runtime, result in appropriate value translations).

Implicit type mappings for the by-value types include:

- The C integer types (including enumerated types) are mapped to the `<integer>` class.
- The C float types are mapped to the `<double-float>` class.

- The C Boolean type defined in `Types.h` is mapped to the `<Boolean>` class.
- The C string types defined in `Types.h` are mapped to the `<Pascal-string>` class.

Whenever one of these types is referenced in an imported interface, Apple Dylan automatically sets up the appropriate type mapping. When you make a cross-language call, Apple Dylan automatically performs the correct value translation for you.

The by-value types that map into Apple Dylan immediate objects (that is, characters, Booleans, and integers within the `<small-integer>` range) pass between languages more efficiently than the other by-value types.

You can define your own by-value types when importing interfaces by mapping C types to Apple Dylan classes that are not pointer classes or that have `import-value` or `export-value` methods. In your program, you define an Apple Dylan class corresponding to each by-value type. You mention these classes by name in your `define interface` statement when setting up type mapping. Although not required, the Dylan representation of an instance of the by-value type is usually an instance of this class. See "Importing and Exporting Values" on page 101 for more information.

Untyped-Pointer Types

Apple Dylan considers a C type to be an **untyped-pointer type** if it points to one of the built-in C types. Some common examples of untyped-pointer types are

- `void*`
- `long*`
- `char*`

Pointers to untyped pointers are also considered untyped. For example, these types are also untyped-pointer types:

- `void**`
- `long***`
- `char****`

By default, all untyped-pointer types encountered when importing an interface are mapped to the built-in Apple Dylan class `<machine-pointer>`.

When an Apple Dylan program passes an instance of a pointer type between languages, only the pointer is copied. During cross-language calls, Apple Dylan converts between objects of class `<machine-pointer>` and C pointers.

Statically-Typed-Pointer Types

Statically-typed pointer types are C pointer types that point to structures, unions, or functions. The C type `Position*` is an example of a statically-typed pointer type.

Multilevel pointer types are also statically-typed as long as their base type is a structure, union, or function. For example, `Position***` is also a statically-typed-pointer type.

Like structure types and union types, statically-typed-pointer types are mapped to subclasses of the Apple Dylan class `<statically-typed-pointer>`.

By default, a statically-typed-pointer type that is a pointer to a structure or union is mapped to the same subclass of `<statically-typed-pointer>` as the base type (the structure union type) it points to.

For example, if the C structure type `Position` is mapped to the Apple Dylan class `<Position>`, then the C type `Position*` is also mapped to the Apple Dylan class `<Position>`.

Adding additional pointer levels (that is, additional asterisks) to a statically typed pointer type yields another statically typed pointer type.

Note
Handles to specific records are also statically typed pointers; the slot getters and setters specialized to a record handle do the extra indirection through the handle. ◆

Unlike C, Apple Dylan type checking requires that a function pointer argument be of the same class as declared, rather than any function pointer with compatible argument and result types.

Explicit Type Mappings

Explicit type mapping is the mechanism that allows you to override and customize the default type-mapping behavior provided by Apple Dylan. You

establish an explicit type mapping by naming a type and a class when importing an interface.

The imported type and the Dylan class to which you map must be compatible. In particular:

- A statically typed pointer or an untyped pointer must be mapped to `<machine-pointer>` or to a subclass of `<statically-typed-pointer>`.
- A by-value type can map to any class. However, this class must have `import-value`, `export-value`, and `export-temporary-value` methods if the low-level Dylan value decoded from the C representation and high-level Dylan value that your program handles are not identical. See the next section and "Importing and Exporting Values" on page 101 for more information.

About Value Translation

At runtime, Apple Dylan uses the type mappings established during interface importation to control value translation between Apple Dylan objects and C values. This translation process occurs in stages. Between the C value and the Apple Dylan value, there is intermediate value, called the **low-level value.**

When translating a value from C to Apple Dylan:

1. The C value is **decoded** to produce the low-level value.
2. The low-level value is **imported** to produce the Apple Dylan value.

When translating a value from Apple Dylan to C:

1. The Apple Dylan value is **exported** to produce the low-level value.
2. The low-level value is **encoded** to produce the C value.

Decoding and encoding are the processes that translate between the C value and the low-level value. The low-level value is dictated solely by the type of the C value:

- **Integer types.** The low-level value is a Dylan integer with the same numerical value as the C value.
- **Floating-point types.** The low-level value is an Apple Dylan floating-point number with the same numerical value as the C value. (There can be

round-off error if the C number has different precision from the Apple Dylan number.)

- **Structure and union types.** The low-level value is a statically typed pointer that points to the structure or union. (Where C allocates a temporary copy of the structure or union, Dylan does the same and the low-level Dylan value is a pointer to that copy).
- **Pointer types.** The low-level value is a statically-typed pointer or an untyped pointer.

Importing and exporting are the processes that translate between the low-level value and the Apple Dylan value. These processes are controlled by the Apple Dylan class.

You can extend the import and export steps by adding methods to the Apple Dylan class. You might have to do this when creating an explicit by-value type mapping, for example, in order to correct for discrepancies between the desired Apple Dylan value and low-level value produced by decoding. See "Importing and Exporting Values" on page 101 for more information.

If the low-level-value is a pointer and the class is a statically-typed-pointer class, then the low-level-value is an instance of the class; otherwise, it is an instance of `<machine-pointer>`. The decoding and encoding steps of value translation are aware of the Apple Dylan classes for typed pointers.

Using Type Mapping

Importing Structures

Apple Dylan imports C structure and union types as Apple Dylan statically-typed pointer classes. For example, consider this structure type defined in the header file `Position.h`:

```
struct Position{
    short v;
    short h;
};
```

You can import this structure type using the following interface definition:

```
define interface
    #include "Position.h",
        import: { "Position" };
end interface;
```

This interface definition creates five Apple Dylan entities:

- a subclass of `<statically-typed-pointer>` named `<Position>`
- a function named `Position$v`
- a function named `Position$v-setter`
- a function named `Position$h`
- a function named `Position$h-setter`

The `<Position>` class represents pointers to C `Position` structures. You can use this class to allocate these C structures on the stack, or to create and destroy these structures in the heap.

For example, to allocate a C `Position` structure in the heap, you use the built-in `make` function:

```
define variable my-c-point = make(<Position>);
```

This call to the `make` function

1. allocates a C `Position` structure in the heap (just as the C function `malloc` would create)
2. creates a new Apple Dylan object of class `<Position>`
3. stores a pointer to the C `Position` structure in the new Apple Dylan object

You access the members of this C `Position` structure just as if they were slots of the Apple Dylan object named `Position$v` and `Position$h`. For example, these assignments use the Apple Dylan object to store values into the C structure:

```
my-c-point.Position$v := 5;
my-c-point.Position$h := 10;
```

These expressions retrieve the values stored in the C structure:

```
my-c-point.Position$v
my-c-point.Position$h
```

As a result, your Apple Dylan program uses objects of class `<Position>` as if they were native Apple Dylan objects with two slots: `Position$h` and `Position$v`. However, they are actually specially-encoded pointers that point to C `Position` structures.

The functions `Position$h` and `Position$v` are not actually slot accessors, but rather functions that use the statically-typed pointer to retrieve information from the members of a C structure. Similarly, the functions `Position$h-setter` and `Position$v-setter` use the pointer to modify the members of the C structure. All four of these functions perform the necessary value translations between C integer values and Apple Dylan integer objects.

Apple Dylan does not perform garbage collection on C structures that you allocate using statically-typed pointers. You are responsible for releasing any memory allocated for these structures using the Apple Dylan function `destroy`:

```
destroy(my-c-point);
```

The Macintosh Toolbox provides functions that deallocate memory for certain types of Toolbox structures. When applicable, you should import those functions and use them, rather than the `destroy` function, to deallocate memory.

The reference section at the end of this chapter contains reference descriptions for the structure functions `make`, `destroy`, and `structure-size`. Chapter 7, "Low-Level Facilities," discusses additional structure functions.

Arrays as Structure Members

When you import a structure (or union) member whose type is an array type, the corresponding getter/setter functions take an extra argument that specifies an array index. For example, consider this C structure type:

```
struct sample {
    int a [10];
    int b [10];
} sample;
```

Importing this structure type creates a statically-typed pointer class named <sample>. You can use this class to create a C structure:

```
define variable my-struct = make (<sample>);
```

Importing the C sample structure type also creates getter/setter function pairs for the members sample$a and sample$b. These functions require an extra argument that specifies the array index. Therefore, these expressions are invalid:

```
my-struct.sample$a := 10;   // not enough arguments to sample$a-setter
let x = my-struct.sample$a; // not enough arguments to sample$a
```

You can access the elements of the member arrays using this code:

```
sample$a(my-struct, 1) := 10;    // sets element 1 to value 10
let x = sample$a(my-struct, 1);  // retrieves element 1
```

If the array is multi-dimensional, the getter and setter take as many extra arguments as the array has dimensions. Apple Dylan accesses only one array element at a time. Array bounds are not checked because C does not check them and many interfaces declare them incorrectly.

However, if the slot's type is mapped to a statically typed pointer class, Apple Dylan does not require the extra arguments. For example, Types.h defines Str31 as unsigned char[32]. A structure slot of type unsigned char[32] is treated as an array of unsigned 8-bit integers; however, because the type Str31 is mapped to <Pascal-string>, the array in the slot is treated as a string and the getter and setter do not take an extra argument.

Importing Pointer-To-Structure Types

Consider the following header file:

```
typedef struct {
    short x;
    short y;
} Position;

typedef Position *PosPtr;
```

```
void DrawLineBetween (PosPtr start, Position* end);
```

If you import the `Position` structure type, the Apple Dylan class `<Position>` results. This class is a subclass of `<statically-typed-pointer>`.

Similarly, if you import the `PosPtr` type, the Apple Dylan class `<PosPtr>` results. This class is a subclass of `<Position>`.

If you import the function `DrawLineBetween`, Apple Dylan creates implicit type mappings for the two parameters:

- The first parameter, `start`, is of type `PosPtr`. Apple Dylan maps the C type of this parameter to the Apple Dylan `<PosPtr>` class.
- The first parameter, `end`, is of type `Position*`. By default, Apple Dylan maps this C type to the same Apple Dylan class that `Position` maps to—in this case, the Apple Dylan class `<Position>`.

As a result, the C function expects two pointers to `Position` structures. The Apple Dylan function expects an object of class `<PosPtr>` and an object of class `<Position>`.

(Remember that `<PosPtr>` is a subclass of `<Position>`—because the typedef expansion of `PosPtr` results in a pointer to the structure type `Position`.)

Importing Pointer-To-Function Types

Consider the following header file:

```
typedef int (*FunPtr) (int, int);
typedef FunPtr FooPtr;
```

Importing type `FunPtr` results in the Apple Dylan class `<FunPtr>`.

To import the type `FooPtr`, Apple Dylan examines its typedef expansion. In this case, the typedef expansion is `FunPtr`, which maps to the Apple Dylan class `<FunPtr>`. As a result, `FooPtr` imports to the Apple Dylan class `<FooPtr>`, which is a subclass of `<FunPtr>`.

Using Explicit Type Mapping

There are a number of options you can use in your interface definitions to override the default type-mapping behavior provided by Apple Dylan:

- The type container option used with a file clause allows you to specify type mappings that apply to an entire C header file.
- The type container option used with a structure or union clause, allows you to specify type mappings that apply when importing members of a structure or union.
- The type variable option allows you to specify the type mapping for a particular imported variable, structure member, or union member.
- The argument-type function option allows you to specify the type mapping for individual arguments of an imported function.
- The result-type function option allows you to specify the type mapping for the function result of an imported function.

As with most interface definition options, the more specific options override the options specified for enclosing containers.

Type Mapping Strings

Apple Dylan provides the classes `<C-String>` and `<Pascal-String>` to represented imported string types. As subclasses of `<statically-typed-pointer>`, these classes inherit a machine-pointer slot:

- Objects of class `<C-String>` use this slot to point to null-terminated strings.
- Objects of class `<Pascal-String>` use this slot to point to strings with an initial length byte.

These classes are also subclasses of `<string>`. Therefore, they support all of the collection class functionality of native Apple Dylan strings.

Since the C programming language does not have a standard string type, each interface must deal with strings in its own way. Here are two possibilities:

```
typedef unsigned char *StringPtr;
typedef unsigned char Str255[256];
```

Both of these types are pointers to `char` types, which are implicitly type mapped to the Apple Dylan class `<machine-pointer>`.

To access the power of the Apple Dylan `<string>` class for imported strings, you can specify explicit type mappings when importing string types. For example:

```
define interface
    #include "StringTypes.h",
        type: { "StringPtr" => <C-String>,
                "Str255" => <Pascal-String> };
end interface
```

This type mapping allows you to manipulate imported strings as if they were Apple Dylan strings, and provides Apple Dylan with the information necessary to pass these types of strings between languages during cross-language function calls.

(The string types defined in `Types.h` are imported for you already, as described on page 92.)

Apple Dylan also allows you to pass a native Apple Dylan string to any imported function that expects a `<C-String>` or a `<Pascal-String>` argument. When you do, Apple Dylan allocates storage and copies the elements of the native string into the appropriate format automatically. (The next section describes how Apple Dylan implements this functionality.)

Finally, you can convert an imported string to a native Apple Dylan string using the `as` function:

```
define variable native-string = as (<byte-string>, imported-string);
```

For more information, see the reference descriptions for the `<C-String>` and `<Pascal-String>` classes, starting on page 110.

Importing and Exporting Values

The import step of the value translation process translates a low-level value (a decoded C value) to an Apple Dylan value. The export step translates the Apple Dylan value to a low-level value, ready to be encoded to C.

There are times when you might want to specify an explicit type mapping that presents importing and exporting issues that Apple Dylan does not know how to handle. Since the by-pointer types are all imported and exported in the same way, these issues typically arise when you are defining a new by-value type mapping.

Note
As a simple example, let's examine how to create a Boolean by-value type mapping. This is example for illustrative purposes only. Apple Dylan already provides the Boolean type mapping behavior shown here.) ◆

In C, a value of 0 represents false, while the value 1 (or any other value) represents true. The file `Types.h` contains the definition

```
typedef unsigned short Boolean;
```

By the default type-mapping rules, this C type maps to the Apple Dylan class `<integer>`. Suppose you want to create a new by-value type mapping:

```
define interface
    #include "Types.h",
        type: { "Boolean" => <Boolean> };
end interface
```

This interface definition type maps the C type `Boolean` to the built-in Apple Dylan class `<Boolean>`. As a result, any C function that expects a `Boolean` argument is imported as an Apple Dylan function that expects a `<Boolean>` argument.

When you call such an imported function, you provide an Apple Dylan `<Boolean>` value—`#t` or `#f`. Unfortunately, the C function expects an `unsigned short`. The value needs to undergo an appropriate translation.

Apple Dylan provides three generic functions used when importing and exporting values:

- The `import-value` function is called to translate a decoded low-level value to an Apple Dylan value.
- The `export-value` and `export-temporary-value` functions are called to translate an Apple Dylan value to a low-level value ready to be encoded and passed to C:

- The `export-value` function is called when returning values from Apple Dylan functions to C functions and when setting a member of a C structure or union.
- The `export-temporary-value` function is called when passing arguments to C functions.

It is only necessary to define `import-value`, `export-value`, and `export-temporary-value` methods for classes where the high-level and low-level Dylan representations are not the same.

In most cases, you should define your methods for these generic functions using `define inline method`, so the compiler can inline them and avoid extra boxing and unboxing steps.

You specialize these functions when creating your own type-mapping behavior. For example, to import a C `Boolean` value, you can use this code:

```
define method import-value (class == <Boolean>, low-level-value)
    if (low-level-value == 0) #f else #t end if;
end method;
```

This method definition adds to the `import-value` generic function a method specialized to the `<Boolean>` class. When Apple Dylan imports a C `Boolean` value, it calls this method, sending the decoded low-level value as the second argument. This decoded value is an Apple Dylan integer with the same numerical value as the C value being imported. The method translates this number to an Apple Dylan `<Boolean>` object.

When exporting a `<Boolean>` object, you need to perform the reverse translation:

```
define method export-value (class == <Boolean>, high-level-value)
    if (high-level-value) 1 else 0 end if;
end method;
```

The `export-temporary-value` method, called when passing arguments to C functions, works differently. This function allows you to allocate storage, pass the low-level value back to Apple Dylan, and deallocate the storage. For `<Boolean>` values, you won't need to allocate any storage, so this method is relatively straightforward:

```
define method export-temporary-value (class == <Boolean>,
                                      high-level-value,
                                      function :: <function>)
    function(if (high-level-value) 1 else 0 end if);
end method;
```

The only difference between this method and the `export-value` method is that Apple Dylan passes a function object to this method, which is expected to pass the low-level value to that function rather than returning it as the method result.

The reference sections for these three generic functions begin on page 114. Chapter 7, "Low-Level Facilities," contains additional information you may find useful when specializing these functions.

In the previous section, you saw that Apple Dylan allows you to pass native strings (of class `<string>`) to imported functions that expect imported strings. This mechanism is implemented using export methods.

For example, Apple Dylan defines these two `export-value` methods:

```
define method export-value (class == <C-string>,
                            high-level-value :: <C-string>)
     high-level-value      // it's already a C string
end method;

define method export-value (class == <C-string>,
                            high-level-value :: <string>)
    as(<C-string>, string) // copy into static memory in C string format
end method;
```

The first of these methods accepts a `<C-string>` object as the high-level value and uses it as the low-level value.

The second method accepts any `<string>` object as the high-level value. This method allocates C memory for a C string, copies the elements of the native string, and returns a statically-typed pointer to the C string as the low-level value.

Similar methods exist for the `export-temporary-value` generic function.

The class `<Pascal-String>` has an analogous set of exporting methods.

Type Mapping Reference

This section contains reference material about pointer classes, structure-related functions, and functions Apple Dylan calls when importing and exporting values.

Pointer Classes

This section describes the Apple Dylan classes that represent machine pointers, and the Apple Dylan superclass for statically-typed-pointer classes. Two built-in statically-defined-pointer classes—the `<C-string>` class and the `<Pascal-string>` class—are also described here.

Chapter 7, "Low-Level Facilities, " provides more information and illustrative examples using these classes.

The <machine-pointer> Class

The `<machine-pointer>` class represents untyped pointers.

Superclass `<basic-machine-pointer>`

DESCRIPTION

Objects of this class are Apple Dylan representations of machine pointers—addresses of raw bits of data. These pointers are untyped—Apple Dylan stores no information about the type of data pointed to.

When importing C header files, Apple Dylan considers pointers to built-in C types (such as `void*` or `long*`) to be untyped. By default, Apple Dylan maps all of these C types to the class `<machine-pointer>`.

You can use objects of this class:

- To pass information to imported functions. However, Apple Dylan cannot perform type checking on the information passed.
- To access the raw data pointed to by the object. However, you must use the low-level facilities described in Chapter 7. Without type information, Apple Dylan cannot provide high-level functions for this purpose.

Two machine pointers are == if they point to the same address; two machine-pointers are also = if they point to the same address.

Two pointers of mixed types (machine pointer and statically typed pointer) can never be == or =.

The < operation is applicable to machine pointers. It compares addresses but not classes. The arguments can be of mixed types (for example, a statically typed pointer can be compared to a machine pointer).

SEE ALSO

For an introduction to this class, see "Untyped-Pointer Types" on page 92.

For more information related to this class, see Chapter 7, "Low-Level Facilities."

The <statically-typed-pointer> Class

The `<statically-typed-pointer>` class is the abstract class that Apple Dylan subclasses when importing statically-typed-pointer types from C interfaces.

Superclass	`<basic-machine-pointer>`
Subclasses	`<C-string>` `<Pascal-string>`

DESCRIPTION

Apple Dylan creates a different subclass of `<statically-typed-pointer>` for each structure type, union type, or statically-typed-pointer type imported from a C header file. The original C type is type mapped to the corresponding subclass.

The purpose of these subclasses is twofold:

- To allow you to pass statically-typed pointers (for example, pointers to structures) to, and receive them from, C library functions.

  Instances of these subclasses are Apple Dylan representations of C pointers to C structures. During cross-language function calls (as well as variable and member accesses), Apple Dylan translates these objects into C pointers.

  Subclassing `<statically-typed-pointer>` for different imported types allows Apple Dylan to perform type-checking during cross-language calls. If an imported function expects a pointer to a particular type of structure, Apple Dylan can ensure the object you provide is of the corresponding subclass.

- To allow you to manipulate C-compatible structures directly from your Apple Dylan program.

  Subclassing `<statically-typed-pointer>` for different imported types allows functions to be specialized on the different subclasses. For example, functions that operate on a conceptual object represented by a particular C structure type can be specialized on the subclass of `<statically-typed-pointer>` for that type.

Two pointers of mixed types (machine pointer and statically typed pointer) can never be == or =.

The < operation is applicable to statically-typed pointers. It compares addresses but not classes. The arguments can be of mixed types (for example, a statically typed pointer can be compared to a machine pointer).

INITIALIZATION

Apple Dylan extends the standard Dylan function `initialize` for statically-typed pointers.

```
define method initialize (<statically-typed-pointer>,
                          #key pointer, extra-bytes)
```

`extra-bytes`	number of bytes larger than declared size
`pointer`	address of existing storage, if any

If the C-compatible structure pointed to by this object ends in an array whose actual length does not match the declared length, you can use the `extra-bytes` argument to indicate how many extra bytes are needed by the structure.

The `pointer` argument allows you to provide an integer, untyped pointer, or statically-typed pointer value. If you provide a value for this argument, Apple Dylan initializes the statically-typed pointer using that value as the address. If you do not provide this argument, Apple Dylan allocates in the heap a C-compatible structure of the appropriate C type and uses the address of that structure to initialize the object.

As a result, by applying the standard Dylan function `make` to statically-typed-pointer classes, you can use Apple Dylan to allocate C-compatible structures in the heap.

SEE ALSO

For an introduction to this class, see "Statically-Typed-Pointer Types" on page 93.

For some examples using this class, see "Importing Structures" on page 95.

For information about functions you can use with this class, see "Structure-Related Functions," which begins on page 111.

The <C-string> Class

The `<C-string>` class provides an Apple Dylan interface to C-compatible, null-terminated strings.

Superclass `<statically-typed-pointer> <string>`

DESCRIPTION

Objects of class `<C-string>` are strings; you can manipulate them as you would manipulate any Apple Dylan `<string>` object.

However, Apple Dylan stores C-string objects differently from native string objects. C-string objects are stored as statically-typed pointers to C-compatible strings.

Therefore, you can manipulate these objects as strings within Apple Dylan, and you can pass them as strings to C library functions.

INITIALIZATION

The `initialize` method for the `<C-string>` class allows you to allocate C-compatible strings.

```
define method initialize (<C-string>,
                          #key pointer, size, maximum-size, fill)
```

`pointer`	The address of the string, if specified.
`size`	The initial number of elements in the string.
`maximum-size`	The maximum number of elements for this string. The default is the initial size.
`fill`	The initial value of the elements. The default is the space character.

To make a C-string object that shares the memory of an existing C-compatible string, you supply the address of the string as the `pointer` argument and the size of the allocated storage minus one as the `maximum-size` argument. If you want Apple Dylan to allocate a C-compatible string for you, you supply the size of the initial string as the `size` argument, the maximum size for the string as the `maximum-size` argument, and the initial value to fill the elements of the string as the `fill` argument.

After the string is initialized, you may use the standard Dylan function `size-setter` to reset its size. However, you may not increase the size of the string above the maximum size indicated in the initialization.

Note that the size of a `<C-string>` scans the string looking for a null character. The size is not cached because it is difficult to know when the cache becomes invalid. The `<C-string>` implementation of the Dylan iteration protocol is reasonably efficient.

SEE ALSO

For a discussion of the C-compatible string classes, see “Type Mapping Strings” on page 100.

The <Pascal-string> Class

The `<Pascal-string>` class provides an Apple Dylan interface to C-compatible strings that have an initial length byte.

Superclass `<statically-typed-pointer> <string>`

DESCRIPTION

Objects of class `<Pascal-string>` are strings; you can manipulate them as you would manipulate any Apple Dylan `<string>` object.

However, Apple Dylan stores Pascal-string objects differently from native string objects. Pascal-string objects are stored as statically-typed pointers to C-compatible strings (with an initial length byte, rather than a null termination).

Therefore, you can manipulate these objects as strings within Apple Dylan, and you can pass them as strings to imported functions.

INITIALIZATION

The `initialize` method for the `<Pascal-string>` class allows you to allocate C-compatible strings with initial length bytes.

```
define method initialize (<Pascal-string>,
                          #key pointer, size, maximum-size, fill)
```

`pointer`	The address of the string, if specified.
`size`	The initial number of elements in the string.
`maximum-size`	The maximum number of elements for this string. The default is the initial size.
`fill`	The initial value of the elements. The default is the space character.

To make a Pascal-string object that shares the memory of an existing string, you supply the address of the string as the `pointer` argument and the size of the allocated storage minus one as the `maximum-size` argument. If you want Apple Dylan to allocate a string for you, you supply the size of the initial string as the

`size` argument, the maximum size for the string as the `maximum-size` argument, and the initial value to fill the elements of the string as the `fill` argument.

After the string is initialized, you may use the standard Dylan function `size-setter` to reset its size. However, you may not increase the size of the string above the maximum size indicated in the initialization.

SEE ALSO

For a discussion of the C-compatible string classes, see "Type Mapping Strings" on page 100.

Structure-Related Functions

This section discusses the functions provided by Apple Dylan that allow you to manipulate C-compatible structures. The functions described here allow you to

- determine the size of a C-compatible structure
- free the memory allocated fora C-compatible structure

To allocate a C-compatible structure in the heap, you use the standard Dylan function `make`, as described in the `<statically-typed-pointer>` reference section on page 106.

To allocate a C-compatible structure on the stack, you use the Apple Dylan macro `with-stack-structure`, which is described in Chapter 7, "Low-Level Facilities," or the macro `with-stack`, defined in the Apple Dylan Framework.

The structure-size Function

You can use the structure-size function to determine the size in bytes required for instances of an imported C structure or union type.

`structure-size (` *class* `) =>` *size*

class	a statically-typed pointer class
size	the size in bytes of instances of the class

DESCRIPTION

The Apple Dylan function `structure-size` is analogous to the C function `sizeof`; you can use it to find the size of imported structures.

The argument to this function must be a subclass of the class `<statically-typed-pointer>`. Typically, you specify a class created by Apple Dylan as the result of importing a C structure or union type. For example, if you use an interface definition to import this C type:

```
struct CPoint {
    long h;
    long v;
} CPoint;
```

Apple Dylan creates a statically-typed pointer class named `<CPoint>`. You can call the `structure-size` function on this class:

```
structure-size(<CPoint>);
```

The result is 8, indicating the number of bytes used by structures of type `CPoint` (two C `long` values of 4 bytes each).

The destroy Open Generic Function

You use the `destroy` generic function to deallocate memory for C structures or unions you created with the `make` function.

`destroy(` *instance* `)`

instance an instance of a statically-typed pointer class

DESCRIPTION

The Apple Dylan function `destroy` is analogous to the C functions `DisposePtr` and `DisposeHandle`.

When you use the `make` function to instantiate a statically-typed pointer class, Apple Dylan creates a C-compatible object. These objects are not subject to

garbage collection—you are responsible for deallocating the memory allocated to them when you no longer need them.

For example, consider this definition:

```
define variable my-structure = make(<CPoint>);
```

Assuming `<CPoint>` is an imported statically-typed pointer class, this call to the `make` function allocates a C-compatible structure of type `CPoint`. The memory used by this structure is not subject to garbage collection. You must explicitly deallocate this memory by calling the `destroy` function:

```
destroy(my-structure);
```

As a result, the allocated structure is freed, and the `my-structure` object contains a null pointer.

SEE ALSO

For information about creating C structures and unions, see the description of the `<statically-typed-pointer>` class on page 106.

For an example of creating a C structure from Apple Dylan, see "Importing Structures" on page 95.

Functions for Importing and Exporting Values

The functions described in this section are called by Apple Dylan during value translation—that is, when passing values between C and Apple Dylan during cross-language calls.

You can specialize these functions to override the default type mapping and value translation behavior.

The import-value Open Generic Function

Apple Dylan calls the `import-value` function when translating C values into Apple Dylan.

`import-value (`*class*`, `*low-level-value*`) => `*high-level-value*

class	The target class for the imported value.
low-level-value	The decoded low-level Apple Dylan object
high-level-value	The fully-translated Apple Dylan value.

DESCRIPTION

When a C value is passed into Apple Dylan during a cross-language call (or a variable or member access), Apple Dylan determines

- the **target Apple Dylan class,** which is specified by a type mapping
- the **low-level value,** which is the result of decoding the C value

Apple Dylan passes these values to the import-value function, which is responsible for translating the low-level value to the appropriate class.

When you specify an explicit type mapping, the Apple Dylan class you specify may not match the class of the low-level values produced by Apple Dylan during decoding.

For example, the section "Importing and Exporting Values" on page 101 includes an explicit type mapping that maps a C integer type to the Apple Dylan class `<Boolean>`. Since C integer types decode to Apple Dylan integer objects, additional translation is required for this type mapping. You perform this translation by adding an `import-value` method specialized for the class `<Boolean>`:

```
define method import-value (class == <Boolean>, low-level-value)
    if (low-level-value == 0) #f else #t end if;
end method
```

When a C value that is type mapped to the class `<Boolean>` is passed into Apple Dylan, Apple Dylan decodes the C value, and calls the `import-value` function,

with the class `<Boolean>` as the first argument and the decoded value as the second argument.

By providing an `import-value` method specialized to the class `<Boolean>`, as is the one above, you can perform the additional translation necessary to convert the low-level value provided by Apple Dylan into an object of the class `<Boolean>`.

Apple Dylan does not require that the value returned by an `import-value` method be an instance of the specified target class. However, it is recommended that your export methods be able to handle whatever classes of objects your import method can produce.

SEE ALSO

For information on how Apple Dylan encodes and decodes values, see "About Value Translation" on page 94.

For an example of specializing this function, see "Importing and Exporting Values" on page 101.

The export-value Open Generic Function

Apple Dylan calls the `export-value` function when translating Apple Dylan objects to C values.

`export-value (`*class*`, `*high-level-value*`) => `*low-level-value*

class	The target Apple Dylan class.
high-level-value	The Apple Dylan object to export.
low-level-value	The low-level value, ready to encode.

DESCRIPTION

Apple Dylan calls this function:

- when returning objects from an Apple Dylan function to C library code

- when setting the value of a member of a C structure or union

In all other cases, Apple Dylan calls the `export-temporary-value` function.

Note that the target Apple Dylan class provided as the first argument is not necessarily the class of the high-level value. Instead, it is the Apple Dylan class specified by the type mapping established for the particular value translation. In this way, it is the type mappings established during interface importation that determine which `export-value` method is called.

If you have specified an explicit type mapping that requires translation before an Apple Dylan object can be encoded as a C value, you should add a method to this function specialized on the target class of that type mapping, as discussed in "Importing and Exporting Values" on page 101.

SEE ALSO

For information on how Apple Dylan encodes and decodes values, see "About Value Translation" on page 94.

For an example of specializing this function, see "Importing and Exporting Values" on page 101.

The export-temporary-value Open Generic Function

Apple Dylan calls the `export-temporary-value` function when translating Apple Dylan objects to C values. In particular, this function is used:

- When passing Apple Dylan objects as arguments to C library functions.
- When setting the value of a member of a C structure or union (but only when the member being set is itself a structure (or union). In this case, Apple Dylan uses `export-temporary-value` and copies the temporary value bit-for bit into the member.

In all other cases, Apple Dylan calls the `export--value` function.

`export-temporary-value (`*class*`,` *high-level-value*`,` *function*`)`

class The target Apple Dylan class.

high-level-value The Apple Dylan object to export.

function A function to pass the low-level-value to.

DESCRIPTION

As with the function `export-value`, you can specialize this function for a particular class to perform any translation necessary to prepare objects of that class to be encoded.

When you specialize this function, however, you typically

1. allocate temporary storage
2. calculate the low-level value
3. pass that value to the function provided
4. deallocate any allocated storage
5. return all values returned by the function provided

SEE ALSO

For information and examples of allocating temporary storage, see Chapter 7, "Low-Level Facilities."

For information on how Apple Dylan encodes and decodes values, see "About Value Translation" on page 94.

For an example of specializing this function, see "Importing and Exporting Values" on page 101.

Type Mapping

CHAPTER 6

Access Paths

Contents

About Access Paths 121
Using Access Paths 122
Using File Options to Specify Information About Access Paths 122
Using the Inline Machine Code Access Path 124
Using The Code Fragment Manager Path 126
Creating a CFM Shared Library 126
Using CFM Shared Libraries 128
Using The Apple Shared Library Manager Access Path 129
Creating an ASLM Shared Library 131
Using ASLM Shared Libraries 132
Using The External Module Access Path 133
Creating a Custom Application Nub 134
Using Statically Linked External Modules 135
Using The Direct Pointer Access Path 135
Preparing a Code Resource 136
Loading and Unloading a Code Resource 138
Calling Functions Imported from a Code Resource 140

This chapter discusses the various access paths that Apple Dylan can use to locate C entities during a cross-language call. A programming example is included for each type of access path.

About Access Paths

This chapter discusses the five access paths your Apple Dylan program can use to locate imported functions during runtime. In particular you'll learn:

- The **Inline Machine Code Access Path** is the access path Apple Dylan uses for Macintosh Toolbox functions on the 68K. This access path is 68K only.
- The **Code Fragment Manager Access Path** allows you to call functions packaged as a CFM shared library. You use this access path if you have access to an existing CFM shared library, or if you want to compile and package your own C-compatible code as a CFM shared library. On the PowerPC, you almost always use this access path.
- The **Apple Shared Library Manager Access Path** allows you to call functions packaged as an ASLM shared library. You use this access path if you have access to an existing ASLM shared library, or if you want to compile and package your own C-compatible code as an ASLM shared library.
- The **External Module Access Path** allows your Apple Dylan program to locate C code compiled normally (into a `".o"` file, for example, rather than as an ASLM or CFM shared library). With this access path, you must use MPW to create a customized version of the Application Nub that contains the compiled C functions. This access path is 68K only.
- The **Direct Pointer Access Path** allows your Apple Dylan program to locate C code compiled and packaged in a code resource (such as a HyperCard XCMD). With this access path, you must write more involved Apple Dylan interface code to create, load, and use the code resource.

For each type of access path, this chapter shows you

- how the access path works
- how to modify your interface definitions for each of the access paths
- how to write Apple Dylan code to perform the cross-language function calls

- how to package your compiled code so that it is accessible at runtime

Using Access Paths

This section provides examples that show how you prepare and how you use the different types of access paths.

Using File Options to Specify Information About Access Paths

Apple Dylan provides a number of file options that you can use to specify the access path to use for C entities imported from a particular C header file.

The **language** file option allows you to specify whether the imported header file is a standard C header file or an ASLM header file. Here are two examples:

```
define interface
    #include "SampleHeader.h",
        language: C;
end interface;

define interface
    #include "SampleHeader.exp",
        language: ASLM;
end interface;
```

In these examples, the specified language option is redundant, because Apple Dylan assumes that header files with filenames ending in `.h` are C header files and that files with filenames ending in `.exp` are ASLM header files. However, if you have a header file that does not use one of these suffixes, you should specify the language:

```
define interface
    #include "SampleHeader.crypticsuffix",
        language: ASLM;
end interface;
```

The **external-module** file option allows you to specify that Apple Dylan should use the external module access path to locate imported functions during runtime. This option requires that you to specify the name of the external module. For example:

```
define interface
    #include "SampleHeader.h",
        external-module: my-external-module;
end interface;
```

In this example, `my-external-module` is the name of an Apple Dylan module that you have created to contain the C-compatible code. "Using The External Module Access Path" on page 133 describes how you create this kind of external module.

You use the **CFM-library** option, the **CFM-library-version** option, and the **CFM-library-oldest-version** option to specify information when you are using the Code Fragment Manager access path.

- The CFM-library option allows you to specify the CFM shared library that contains the compiled functions you want to call from your Apple Dylan program.
- The CFM-library-version option allows you to specify the preferred version of the CFM shared library.
- The CFM-library-oldest-version option allows you to specify the oldest acceptable version of the CFM shared library.

Here is an example that use these three options:

```
define interface
    #include "SampleHeader.h",
        CFM-library: "my_cfm_library",
        CFM-library-version: 3,
        CFM-library-oldest-version: 2;
end interface;
```

You do not have to specify the CFM-library-version or the CFM-library-oldest-version options when you specify the CFM-library option. This interface definition is perfectly acceptable:

```
define interface
    #include "SampleHeader.h",
        CFM-library: my_cfm_library;
end interface;
```

If you don't specify either of these two options, the default behavior causes the newest version of the library to be used. If you specify the CFM-library-version option, but not the CFM-library-oldest-version option, the latter defaults to the former, thus requesting a specific version of a library.

Using the Inline Machine Code Access Path

Apple Dylan uses the Inline Machine Code Access Path automatically whenever you import a C header file that specifies trap instructions, or other inline machine code, as the implementation for a function.

This access path is available only on the 68K, not the PowerPC.

For example, the file `QuickDraw.h` includes this function definition:

```
extern pascal short Random(void)
= 0xA861;
```

This definition specifies the implementation of the `Random` function in machine code. In this case, the machine code is a Macintosh trap instruction.

Note
Actually, the function `Random` is defined as

```
extern pascal short Random(void)
ONEWORDINLINE(0xA861);
```

The `ONEWORDINLINE` macro is defined in the file `ConditionalMacros.h`; if the `CFMSystemCalls` preprocessor macro is not set, the `ONEWORDINLINE` macro is defined as

```
#define ONEWORDINLINE(trapNum) = trapNum
```

which is a macro that is expanded into an equals sign followed by the argument to the macro. ◆

When you import the `Random` function, using this interface definition

```
define interface
    #include "QuickDraw.h",
        import: { "Random" };
end interface;
```

Apple Dylan parses the `QuickDraw.h` header file and notices that the machine code implementation of the `Random` function is specified. Therefore, when you call the `Random` function from your Apple Dylan program, for example:

```
define variable *new-number* = Random();
```

Apple Dylan simply executes the machine code specified in the header file. In the case of Macintosh Toolbox functions, such as `Random`, this machine code is a trap instruction and the Macintosh trap dispatcher performs the necessary linking dynamically.

When the Macintosh Toolbox function returns, Apple Dylan performs type conversion on the function result. The function definition in `QuickDraw.h` specified that the return value is of type `short`, so Apple Dylan converts the value to an object of class `<integer>`. In the example above, this object is bound to the name `*new-number*`.

When you call imported Toolbox functions that take arguments, Apple Dylan performs type conversion on the Apple Dylan objects that you specify as arguments, sets up the arguments on the stack, and then executes the specified trap instruction.

For the above example to work on both 68K-based and PowerPC-based Macintosh computers, you need to use the inline machine code access path on the 68K and the Code Fragment Manager access path on the PowerPC. You do this by adding the file option

```
CFM-Library: "InterfaceLib"
```

which will be ignored on the 68K-based machine because the Random function has inline machine code, which supersedes this option.

Using The Code Fragment Manager Path

The Code Fragment Manager (CFM) access path allows your Apple Dylan program to call functions compiled and packaged as a CFM shared library. This is the most commonly used access path on the PowerPC.

This is the only access path that provides access to imported variables.

You must use this access path if you want to use code that has already been packaged as a CFM shared library. To use a prepackaged CFM shared library, you need the Code Fragment Manager system extension, which handles the dynamic linking necessary when your Apple Dylan program calls a function packaged in a CFM shared library. (This extension is always present on PowerPC-based Macintosh computers. On 68K-based machines, this extension is named "CFM68K".)

In addition to the CFM system extension, you also need

- The CFM shared library that contains the compiled functions you want to call from your Apple Dylan program. This shared library will also be in the form of a system extension, which must be installed on your Apple Dylan runtime machine.
- The C header files that declare the C entities available in the shared library. These files use standard C header file syntax to declare the entities exported by the shared library. In your interface definitions, you instruct Apple Dylan how to parse these files to create Apple Dylan objects that correspond to the shared library entities.

If you want to use the CFM access path to call C-compatible functions that you have written (or C-compatible functions for which you have the source code), you must create your own CFM shared library, as described in the next section.

Creating a CFM Shared Library

Before you can use the CFM access path to call your own C-compatible functions from Apple Dylan, you must package your functions as a CFM shared library.

You also need to create a C header file to declare the C entities you want to export from your shared library.

For CFM shared libraries, you specify information about the shared library and its exported functions in your source code file and in a CFM resource.

In your source code file, you specify which functions are exported by the CFM shared library using a special preprocessor directive. For example, here is the definition of the function exported by the sample CFM shared library included with Apple Dylan:

```
#pragma lib_export on
long scaled_sine(long angle, long amplitude) {
    float argument = angle * 3.1415926535 / 180.0;
    float wave = sin(argument) * (float)amplitude;
    return((long)wave);
}
```

The first line in this example uses the `#pragma` preprocessor directive to specify that the following function should be exported by the shared library.

Once you have the required elements (the CFM system extension, the C header files, and the C-compatible source code with the proper CFM preprocessor directives), you use a development environment such as MPW to compile and package your source code as a CFM shared library system extension.

If you are using MPW, you need to create a rez file that controls the creation of the system extension file that contains your shared library. Here is the sample resource file for the CFM shared library example included with Apple Dylan:

Listing 6-1 A sample rez file for a CFM shared library

```
#include "systypes.r"
#include "types.r"
#include "CodeFragmentTypes.r"

resource 'cfrg' (0) {
    {
        'm68k',     /* archType */
        kFullLib,   /* kFullLib | kUpdateLib */

        0,          /* currentVersion (only 0 works) */
        0,          /* oldDefVersion */
```

```
        0,          /* appStackSize (0 = use default )*/
        0,          /* appSubFolderID  (0 = none)*/
        kIsLib,     /* Frag type kIsApp | kIsLib | kIsDropIn */
        kOnDiskSegmented, /* container model & location */
        'rseg',           /* dataFork offset | rsrc Type */
        0,                /* dataFork length | rsrc ID */
        FRAG_NAME,        /* Fragment container name  */
    };
};
```

This rez file specifies the information needed by the resource compiler to create a code fragment resource containing the compiled sample code. The Code Fragment Manger uses this resource to load and locate the compiled functions at runtime.

When you create your own CFM shared library, you must be sure that the CFM system extension and your CFM shared library are installed on any machine that runs your Apple Dylan application.

Using CFM Shared Libraries

Before you can call CFM shared library functions from your Apple Dylan program, you must import those functions, and any other necessary C entities, using an interface definition. Since CFM shared libraries do not use a special header file, as ASLM shared libraries do, you import CFM shared library entities directly from a C header file.

When importing a CFM shared library, however, you must specify the name of the code fragment that contains the functions you want to import. For example, you could use this interface definition to import C entities from the CFM example included with Apple Dylan:

```
define interface
    #include "scaled_sine.h",
        CFM-library: "dyln.scaled_sine_example";
end interface;
```

This interface definition imports all the exported entities defined in the C header file `scaled_sine.h`. It also specifies which code fragment Apple Dylan should use to locate the functions during runtime.

You can use any of the standard interface definition clauses and options to provide more specific control of the importation process. For example:

```
define interface
    #include "scaled_sine.h",
        CFM-library: "dyln.scaled_sine_example",
        rename: { "scaled_sine" => imported-scaled-sine };
end interface;
```

As with functions imported from a CFM shared library, you use the standard Apple Dylan function call syntax to call functions imported from a CFM shared library. Here is an example:

```
define method test-imported-function ()
    let angle = 45;
    let amplitude = 2;

    imported-scaled-sine (angle, amplitude);
end method;
```

From the point of view of your program, the imported function is indistinguishable from a standard Apple Dylan function.

Using The Apple Shared Library Manager Access Path

The Apple Shared Library Manager (ASLM) access path allows your Apple Dylan program to call functions compiled and packaged as an ASLM shared library.

You must use this access path if you want to use code that has already been packaged as an ASLM shared library. To use a prepackaged ASLM shared library, you need

- The Apple Shared Library Manager system extension, version 1.1 or newer. This extension must be installed on your Apple Dylan runtime machine (or any machine on which you plan to run your Apple Dylan application). The Apple Shared Library Manager handles the dynamic linking necessary when your Apple Dylan program calls a function packaged in an ASLM shared library.

- The ASLM shared library that contains the compiled functions you want to call. This shared library will also be in the form of a system extension, which must be installed on your Apple Dylan runtime machine.
- The ASLM header file that contains information about the shared library, such as its name and version, and lists the functions exported by the shared library—that is, the functions that you can import into your Apple Dylan program. By convention, ASLM header files have filenames that end with the `.exp` suffix.
- The C header files that declare the C entities available in the shared library. These files use standard C header file syntax to declare the entities exported. In your interface definitions, you instruct Apple Dylan how to parse these files to create Apple Dylan objects that correspond to the shared library entities.

You use the ASLM header file to import information into your Apple Dylan program. For example, here is the sample ASLM header file included with the Apple Dylan sample code:

Listing 6-2 A sample ASLM header file

```
#include "scaled_sine.h"

Library
{
    id = "dyln:dyln$scaled-sine-example,1.0";
    version = 1.0d1;
};

FunctionSet scaled_sine
{
    id = "dyln:dyln$scaled-sine-example,1.0";
    version = 1.0d1;
    exports = extern scaled_sine;
};
```

This ASLM header file includes the C header file `scaled_sine.h`, which contains the C declaration of the sample `scaled_sine` function:

```
long scaled_sine(long angle, long amplitude);
```

The ASLM header file also includes a `Library` definition, which specifies the name and version of the ASLM library, and a `FunctionSet` definition, which specifies the function exported from the library.

(You cannot import ASLM classes into Apple Dylan; only function sets can be imported. In addition, only functions declared extern in a function set can be imported.)

If you want to use functions that have already been packaged into an ASLM shared library, you must have the library's ASLM header file, so that you can import the necessary entities into your Apple Dylan program.

If you want to use the ASLM access path to call C-compatible functions that you have written (or C-compatible functions for which you have the source code), you create your own ASLM header file and package your C-compatible functions into an ASLM shared library, as described in the next section.

Creating an ASLM Shared Library

Before you can use the ASLM access path to call your own C-compatible functions from Apple Dylan, you must package your functions as an ASLM shared library.

You also need to create an ASLM header file that contains information about your shared library, as well as C header files that use standard C syntax to declare the C entities you want to export from your shared library.

Once you have the required elements (the ASLM system extension, the ASLM header file, the C header files, and the C-compatible source code), you use a development environment such as MPW to compile and package your source code as an ASLM shared library system extension.

If you are using MPW, you will also need to create a rez file that controls the creation of the system extension file that contains your shared library. Here is the sample resource file for the ASLM shared library example included with Apple Dylan:

Listing 6-3 A sample rez file for an ASLM shared library

```
#include "VersionResource.r"

INCLUDE "scaled_sine.RSRC";
```

When you create your own ASLM shared library, you must be sure that the ASLM system extension and your ASLM shared library are installed on any machine that runs your Apple Dylan application.

Using ASLM Shared Libraries

Before you can call ASLM shared library functions from your Apple Dylan program, you must import those functions using an interface definition. You must also import any constants and types needed to call those functions.

In general, you

1. import the constants and types (but not the functions) from the C header file
2. import the functions from the ASLM header file (the `.exp` file)

The ASLM example included with Apple Dylan imports one function. No imported C types or constants are necessary, so the example only needs one interface definition:

```
define interface
    #include "scaled_sine.exp";
end interface;
```

This interface definition imports all the entities defined in the C header files included in the `scaled-sine.exp` header file. In particular, it imports all of the functions explicitly exported by that file.

You can use any of the standard interface definition clauses and options to provide more specific control of the importation process. For example:

```
define interface
    #include "scaled_sine.exp",
        rename: { "scaled_sine" => imported-scaled-sine };
end interface;
```

When you call an imported ASLM function, you simply use the standard Apple Dylan function call syntax, as in this example:

```
define method test-imported-function ()
    let angle = 45;
    let amplitude = 2;

    imported-scaled-sine (angle, amplitude);
end method;
```

Apple Dylan converts the integer objects named `angle` and `amplitude` to C values of type long, calls the imported function, and converts the return value from a C value of type `long` into an Apple Dylan integer object.

From the point of view of your program, the imported function is indistinguishable from a standard Apple Dylan function.

Using The External Module Access Path

The External Module access path provides a mechanism for your Apple Dylan program to call C-compatible functions that are not packaged as an ASLM or CFM shared library. This access path is available only on 68K-based Macintosh computers.

In particular, you use this access path

- if you have the header file and the compiled code (the `.h` and `.o` files), but you do not have access to the source code
- if you have access to the source code, but you want to call the imported functions in an environment similar to the traditional environment of a Macintosh application
- if you want your application to run on Macintosh computers that have neither the ASLM nor the CFM system extension installed

To use this access path, you must use MPW or a compatible development environment to create a customized version of the standard Apple Dylan Application Nub. This custom Application Nub includes a special module, called an **external module,** that contains the compiled C-compatible code.

Apple Dylan includes the external module in your Apple Dylan project automatically when you specify an `external-module` option in one of your

interface definitions, described in the next section. The external module is used by Apple Dylan as a linking mechanism. It is not a real Dylan module; you don't use it directly to access module variables, and so on.

The next section also explains how you use a header file, a compiled code file, Apple Dylan, and MPW to create an external module and link it into your own custom version of the Application Nub.

Creating a Custom Application Nub

To create a custom Application Nub, you follow these six steps:

1. Launch the Apple Dylan development environment, but do not launch or connect to an Application Nub.
2. Use the Apple Dylan development environment to create a project. Compile interface definitions that import the desired C entities from your C header file. Apple Dylan collects information from these interface definitions into an external module. For example, here is the interface definition used in the Static Linking example included with Apple Dylan:

   ```
   define interface
       #include ":Application:Sources:scaled-sine.h",
           external-module: scaled-sine-external-module;
   end interface;
   ```

   This interface definition specifies that Apple Dylan import all of the C entities declared in the `scaled-sine.h` file into an external module named `scaled-sine-external-module`. (If you have already connected to the Application Nub, this interface definition results in an error because the standard Application Nub does not contain an external module by that name.)
3. Execute the special compile-time function `write-external-modules`. This function collects information about all the external modules that you have created and writes a description of them to an MPW file. Here is an example:

   ```
   write-external-modules("external-modules.a");
   ```

   This compile-time function call creates an MPW assembler file named `external-modules.a` that contains information about any external modules you have previously defined using interface definitions like the one in Step 2. This file may also contain information about external modules that Apple Dylan defines; you can ignore these. The `write-external-modules` function

creates the `external-modules.a` file in the project folder. You need to move this file to a folder where it can be found by the MPW Linker.

4. Launch MPW (or a compatible development environment). If you are using MPW, you need to use version 3.3 or newer.
5. Use the MPW linker to link your `external-modules.a` file, your compiled C-compatible code (your `.o` file), and the `RunDynamoLib.o` file. into a custom Application Nub. You can use the sample build file included with Apple Dylan (as part of the Static Linking example), which builds a program named `"My Application Nub"`. You might have to redefine some variables in this build file to reflect your file names and file hierarchy.
6. In the Apple Dylan development environment, connect to your custom Application Nub. You can now update your project and access the imported C entities from your Apple Dylan program.

Using Statically Linked External Modules

Once you've gone to the trouble to create the custom Application Nub, you can call the imported functions exactly as you would any Apple Dylan function.

As with functions imported from an ASLM shared library or a CFM shared library, you use the standard Apple Dylan function call syntax to call functions imported from an external module. As with the ASLM and CFM access paths, the imported function is indistinguishable from a standard Apple Dylan function.

Using The Direct Pointer Access Path

The Direct Pointer access path allows you to call functions given C-compatible function pointers. In the following discussion, this access path is used to provide a mechanism for calling C-compatible functions stored in a code resource.

Using this access path to call code resource functions requires more work than using the other access paths. In particular, you need to

- modify the code resource so that it begins with a transfer vector, which transfers control to the various functions in the code resource
- write Apple Dylan code to load and unload the code resource

- create callout functions by using an interface definition to import pointer-to-function types for the various functions in the code resource

Then, to call a function from the code resource, you need to

- calculate the location in the transfer vector that transfers control to the function
- use the appropriate callout function to apply the imported function to specified arguments

Note
The code resource approach introduced here and described in the next three sections is specific to 68K-based Macintosh computers; it would require modification to work on a PowerPC-based Macintosh. ◆

Preparing a Code Resource

To prepare a code resource that contains functions suitable for calling from Apple Dylan using the Direct Pointer access path, you add a **transfer vector** to the beginning of the compiled C-compatible code and link the result into a code resource.

As an example, consider a sample C program that contains three functions:

```
long modify (long input) {
    /* . . . implementation . . .*/
}

long morph (long input) {
    /* . . . implementation . . .*/
}

long mogrify (long input) {
    /* . . . implementation . . .*/
}
```

Each of these functions requires one argument of type `long` and returns a function result of type `long`. The result of compiling this source file is a compiled code (`.o`) file. To make this compiled code suitable for calling from Apple Dylan using the Direct Pointer access path, you need to compile a

transfer vector suitable for the compiled code. A **transfer vector** is a series of assembly language instructions (of equal length) that transfer control to each of the exported functions.

A sample transfer vector, written in assembly language and suitable for the C program above, is shown in the following code listing:

Listing 6-4 An assembly language transfer vector

```
    case on

    macro
    xfer &to
    import &to:code
    bra.w &to
    endm

transfer_vector: proc export
     xfer modify         ; offset to this line = 0
     xfer mogrify        ; offset to this line = 4
     xfer morph          ; offset to this line = 8
end
```

This transfer vector contains three elements, each of length 4. The first transfers control to the `modify` function, the second transfers control to the `mogrify` function, and the third transfers control to the `morph` function.

To create a suitable code resource, you compile the transfer vector and link it to the beginning of the compiled C code, storing the result in a code resource. The linker links the transfer vector to the compiled C code by name, so you must be sure that the function names you specify in your transfer vector match the function names in the compiled C code.

In this example, your code resource contains your transfer vector and the three compiled functions. The transfer vector, located at the beginning of the code resource, specifies instructions at locations 0, 4, and 8 that transfer control to each of the three functions, respectively.

See the Code Resource example included with Apple Dylan for examples of the MPW commands you can use to accomplish these compilations, linking, and code resource creation.

During development, and when creating a standalone application, you should give your project the code resource as a resource file. In this way, you won't have to explicitly open and close the resource file.

However, if you resource file is not added to your project, you will need to specifically open and close it, as described in the next section.

Loading and Unloading a Code Resource

Before you can call the functions stored in your code resource, you must load the code resource into memory—and when you are finished calling the imported functions, you should unload the resource. To load and unload a code resource, you need to import these Macintosh Toolbox functions:

```
define interface
    #include "Resources.h",
        import: { "OpenResFile", "GetResource",
                  "HomeResFile", "CloseResFile" };
    function "OpenResFile",
        argument-type: {"fileName" => <Pascal-string>};

    #include "Memory.h",
        import: { "MoveHHi", "HLock",
                  "RecoverHandle", "HUnlock" };

    #include "OSUtils.h",
        import: { "FlushCodeCache" };
end interface;
```

You might also want to define a constant that represents the resource ID of your code resource, which you would have chosen during the MPW linking process. Here is an example:

```
define constant $my-resource-id = 10001;
```

You will also need a variable to use as a pointer to the code resource when it is loaded in memory. Apple Dylan provides the `<machine-pointer>` class for this purpose; you can use machine-pointer objects to store actual memory addresses, called **machine pointers.** Apple Dylan also provides the constant

`$null-machine-pointer`, which you can use to initialize your code resource pointer variable:

```
define variable *code-resource* :: <machine-pointer>
                = $null-machine-pointer;
```

Once you have imported the appropriate Macintosh Toolbox functions and created your resource ID constant and your code resource pointer variable, you can use this Apple Dylan function to load your code resource and initialize your pointer to it:

```
define constant load-code-resource = method ()
    let file = OpenResFile("My Code Resource");
    if (file < 0)
        error("Can't open code resource file.");
    end if;

    let code-resource = GetResource(OSType("code"),
                                    $my-resource-id);
    if (code-resource = $null-machine-pointer)
        error("Can't find code resource");
    end if;

    MoveHHi(code-resource);
    HLock(code-resource);
    FlushCodeCache();
    *code-resource* := pointer-at(code-resource); // deref handle
end method;
```

This function opens your resource file, loads your code resource, and stores a machine pointer to your code resource in your `*code-resource*` variable.

Here is the corresponding code for unloading the resource and resetting your machine pointer variable:

```
define constant unload-code-resource = method ()
    let handle = RecoverHandle(*code-resource*);
    let file = HomeResFile(handle);
    *code-resource* := $null-machine-pointer;
```

```
    HUnlock(handle);
    CloseResFile(file);
end method;
```

This function unlocks your code resource, closes the code resource file, and resets the value of the module variable `*code-resource*`.

Calling Functions Imported from a Code Resource

To call the functions stored in your code resource, you must have a header file that declares pointer-to-function types for the imported functions. You might also want to include constants in this header file that specify the offsets into your transfer vector for each of the imported functions. Here is an example:

Listing 6-5 Header file for functions imported from a code resource

```
typedef long (*modify-proc-ptr) (long);

typedef long (*morph-proc-ptr) (long);

typedef long (*mogrify-proc-ptr) (long);

enum { kmodify = 0;
       kmorph = 4;
       kmogrify = 8 };
```

Then, you use an interface definition to import the pointer-to-function types as callout functions:

```
define interface
    #include "SampleHeader.h";

    callout "modify-proc-ptr" => call-modify;
    callout "morph-proc-ptr" => call-morph;
    callout "mogrify-proc-ptr" => call-mogrify;
end interface;
```

This interface definition creates three callout functions. Here is an example of how you would use one of these callout functions to call a function from your code resource:

```
define method modify (input :: <integer>)

    if (*code-resource* = $null-machine-pointer)
        load-code-resource();
    end if;

    let ptr-to-modify = as(<modify-proc-ptr>,
                           *code-resource* + kmodify);

    call-modify(ptr-to-modify, input);

end method;
```

This function ensures that the `*code-resource*` global variable has been initialized; if not, it calls the `load-code-resource` function (defined in the previous section) to load the code resource and initialize `*code-resource*`.

Then, the function creates a local variable, `ptr-to-modify`, which points directly to the transfer vector that transfers control to the `modify` function in the code resource.

Finally, this function uses the `call-modify` callout function to apply the imported `modify` function to the `input` argument.

CHAPTER 7

Low-Level Facilities

Contents

About Low-Level Facilities 145
Using Low-Level Facilities 145
  Using Machine Pointers Directly 146
  Allocating Stack Storage 148
  Making Low-Level Function Calls 150
  Calling Dylan Methods From Other Languages 151
  Using Preemption and Event Checking 151
Low-Level Facilities Reference 152
  Reading and Writing Data Using Machine Pointers 152
    Reading and Writing A Single Byte 153
      The signed-byte-at Method 153
      The signed-byte-at-setter Method 154
      The unsigned-byte-at Method 155
      The unsigned-byte-at-setter Method 155
      The character-at Method 156
      The character-at-setter Method 157
    Reading and Writing Pairs of Bytes 158
      The signed-short-at Method 158
      The signed-short-at-setter Method 158
      The unsigned-short-at Method 159
      The unsigned-short-at-setter Method 160
      The unicode-character-at Method 161
      The unicode-character-at-setter Method 161
    Reading and Writing Long Words 162
      The signed-long-at Method 162
      The signed-long-at-setter Method 163
      The unsigned-long-at Method 164

The unsigned-long-at-setter Method 164
The pointer-at Method 165
The pointer-at-setter Method 166
The ostype-name-at Method 167
The ostype-name-at-setter Method 167
Reading and Writing Strings 168
The c-string-at Method 168
The c-string-at-setter Method 169
The Pascal-string-at Method 170
The Pascal-string-at-setter Method 170
Low-Level Macros for Allocating Stack Storage 171
The with-stack-block Macro 171
The with-stack-structure Macro 173
Low-Level Functions for Cross-Language Function Calls 174
The %Dynamo-alien-call Function 174
The alien-method Macro 176
Preemption Facilities Reference 178
Preemption Constants 178
Preemption Functions 178
The %preemption-status Function 178
The %preemption-status-setter Function 179
The %preemption-disable Function 180
The %preemption-status-enabled? Function 180
Preemption Macros 181
The without-preemption Macro 181
The saving-preemption-status Macro 182
Event-Checking Facilities Reference 183
Break-Request Variables and Functions 183
The break-request-key-chord? Function 184
Event-Checking Functions 185
The event-check Function 185
The event-check-interval Function 186
The set-event-check-interval Function 187
Microsecond Functions 187
The %microsecond-time Function 187
The %microsecond-time-difference Function 188
The %microsecond-time-microseconds Function 189

In this chapter, you'll see some of the low-level facilities provided by the Apple Dylan language extensions. You'll learn how to use the low-level facilities for direct control of machine memory, cross-language calls, preemption, and event checking.

About Low-Level Facilities

The low-level facilities discussed in this chapter allow you to read and write bytes of data directly between your Apple Dylan program and machine memory. The section "Using Machine Pointers Directly" on page 146 introduces the functions that allow you direct memory control.

You can also use functions discussed in this chapter to allocate temporary blocks of memory on the stack, and access the data stored there from within your Apple Dylan program. The section "Allocating Stack Storage" on page 148 gives a complete example.

The low-level facilities also allow you to call foreign code from Apple Dylan and to call Apple Dylan from foreign code. The sections "Making Low-Level Function Calls" on page 150 and "Calling Dylan Methods From Other Languages" on page 151 provide a brief introduction to low-level cross-language calls. Typically, you'll use the more convenient mechanisms discussed in Chapter 4, "Cross-Language Calls," instead of the techniques described in this chapter.

The Apple Dylan preemption facilities are included with the low-level facilities. Preemption allows you to interrupt an executing program to temporarily handle another process and then resume the original task.

Finally, the event-checking facilities implement a feature of the Dylan runtime in which periodically it preempts the executing program for the purpose of checking whether the Macintosh toolbox has any events to deliver.

Using Low-Level Facilities

The following six sections discuss the low-level facilities and give examples of their use. You might want to be familiar with the material presented in the

earlier chapters, particularly Chapter 5, “Type Mapping,” before approaching some of the material presented in this chapter.

Using Machine Pointers Directly

Apple Dylan provides a number of utility functions you can use to access machine memory directly. These functions accept a machine pointer as an argument and allow you to read or write the value stored in the memory location indicated by that pointer.

Apple Dylan provides a variety of these utility functions; you use different functions to read or write memory depending on:

- the size of the value you want to read or write
- how the raw memory value should be interpreted

The following table compares the various functions available to you:

Function	Number of bytes	Result
`character-at`	1	`<character>`
`unicode-character-at`	2	`<character>`
`signed-byte-at`	1	`<integer>`
`unsigned-byte-at`	1	`<integer>`
`signed-short-at`	2	`<integer>`
`unsigned-short-at`	2	`<integer>`
`signed-long-at`	4	`<integer>`
`unsigned-long-at`	4	`<integer>`
`pointer-at`	4	`<machine-pointer>`
`OStype-name-at`	4	`<byte-string>`
`c-string-at`	variable	`<byte-string>`
`pascal-string-at`	variable	`<byte-string>`

Each of these functions requires a single argument—a machine pointer—and returns an Apple Dylan object that represents the value stored in the corresponding memory location. For example, if you have a machine pointer

named raw-pointer, you can use the following code to determine the character stored at the address indicated by that machine pointer:

```
let raw-data :: <character> = character-at(raw-pointer);
```

The character-at function examines one byte of memory at the specified address, interprets that byte as a character, and returns an Apple Dylan character object.

The signed-byte-at and unsigned-byte-at functions also examine one byte of memory and return an Apple Dylan object; however, these functions interpret the value of the byte as an integer, and therefore they return Apple Dylan integer objects. The signed-byte-at function interprets the byte as a signed value, ranging from -128 to 127. The unsigned-byte-at function interprets the byte as a value between 0 and 255.

Similarly, the signed-short-at, unsigned-short-at, signed-long-at, and unsigned-long-at functions return Apple Dylan integers. These functions, however, examine more than a single byte of memory.

The pointer-at function reads four bytes of memory and returns an Apple Dylan <machine-pointer> object equivalent to the value of those four bytes.

The c-string-at and pascal-string-at functions examine variable lengths of memory and return their results in Apple Dylan strings. The c-string-at function searches for a null character to terminate the string, copying the preceding bytes into an Apple Dylan string object. The pascal-string-at function interprets the first byte of the memory as a length byte, and copies the subsequent indicated number of bytes to a string object.

Apple Dylan provides a setter function that corresponds to each of the functions described above. You can use these setter functions to write values directly to memory locations.

For example, imagine that you import the NewPtr function from the file memory.h:

```
define interface
    #include “Memory.h”,
        import: { “NewPtr” };
end interface;
```

You can use this function to create raw machine pointers:

```
define variable char-pointer = NewPtr(1);

define variable short-pointer = NewPtr(2);
```

You can then use these machine pointers to control memory directly:

```
Dylan> character-at-setter('Q', char-pointer);
‘Q’

Dylan> character-at(char-pointer) == ‘Q’
#t

Dylan> unsigned-short-at (short-pointer) := 1000; // uses setter function
1000

Dylan> unsigned-short-at(short-pointer) == 500
#f
```

Notice the use of the extended syntax for the assignment operation:

```
unsigned-short-at (short-pointer) := 1000;
```

This is equivalent to the following function call:

```
unsigned-short-at-setter (1000, short-pointer);
```

The reference section “Reading and Writing Data Using Machine Pointers,” beginning on page 152, contains a detailed description of each of the functions for direct memory manipulation.

Allocating Stack Storage

Apple Dylan provides two low-level mechanisms for allocating temporary storage on the stack:

- The `with-stack-block` macro allows you to allocate a block of memory on the stack and provides you with a a pointer to that block of memory. You can then copy data into that memory and perform any other desired calculations and function calls. When the macro finishes executing, it deallocates the temporary memory from the stack.

- The `with-stack-structure` macro, which is built on top of the `with-stack-block` macro, provides a convenient mechanism for creating C structures on the stack.

Note
The Application Framework provides the `with-struct` macro, a somewhat higher-level interface for allocating temporary memory. If you are using the Application Framework, it is recommended that you use the `with-struct` macro, rather than the macros described in this section. ◆

The following example shows how you might use the `with-stack-block` macro to implement the `export-temporary-value` method for `<Pascal-string>` objects. (This example is for illustrative purposes only; Apple Dylan already defines this method for you. See Chapter 5, "Type Mapping," for more discussion concerning the `export-temporary-value` function.)

Listing 7-1 Using the `with-stack-block` macro to allocate stack storage

```
define method export-temporary-value ( for-class == <Pascal-string>,
                                       high-level-value :: <string>,
                                       function :: <function> )
let n = size (high-level-value);
if (n > 255)
    error("The string is too long to export as a Pascal string.");
end if;

with-stack-block ( string(<Pascal-string>, n + 1 ),
                   begin
                       let temp = as (<machine-pointer>, string);
                       unsigned-byte-at(temp) := n;
                       for (c in high-level-value,
                            p = (temp + 1) then (p + 1))
                          character-at(p) := c;
                       end for;
                       function(string);
                   end);
end method;
```

In this example, notice that the call to the `with-stack-block` macro takes two arguments. The first argument is

```
string(<Pascal-string>, n + 1)
```

This is actually a macro call, not a function call; it indicates to the `with-stack-block` macro to allocate `n + 1` bytes of temporary memory for an object of class `<Pascal-string>`, and bind that object to the variable name `string`. (The class you specify must be a `<machine-pointer>` or a statically-typed pointer class—such as the class `<Pascal-string>`.)

The second argument to the `with-stack-block` macro is a body of code to execute. During the execution of this code, the variable name `string` references a valid `<Pascal-string>` object, which has been allocated on the stack. In the example, this body of code uses direct memory access:

- to write a length byte into the temporary string object
- to copy, character by character, the elements of the high-level string into the bytes of the temporary memory allocation

When the high-level value has been translated to Pascal-string format and copied into the temporary memory, the body of code passes the `string` variable (which is effectively a pointer to the temporary memory) to the specified function. When this body of code finishes executing, Apple Dylan deallocates the temporary memory and unbinds the variable name `string`.

The `with-stack-structure` macro is similar, except it allows you to allocate memory for a temporary C structure and computes the structure size automatically.

See "Low-Level Macros for Allocating Stack Storage," beginning on page 171 for reference information for the `with-stack-block` and `with-stack-structure` macros.

Making Low-Level Function Calls

The `%Dynamo-alien-call` function provides a low-level mechanism for making cross-language function calls from Apple Dylan.

The callout mechanism, described in Chapter 4, "Cross-Language Calls," provides a high-level interface to `%Dynamo-alien-call`. If you use the techniques

described in that chapter, you should not have to call `%Dynamo-alien-call` directly.

However, this function is provided for you so that you can handle situations in which the callout mechanism is not desired or appropriate. For example, you can use this function when calling assembly-language interfaces. Any use of this function results in machine-dependent Apple Dylan code.

The reference section on page 174 describes this function in more detail.

Calling Dylan Methods From Other Languages

The `alien-method` macro provides a low-level mechanism for calling Apple Dylan code from C-compatible libraries. This macro returns a pointer to an Apple Dylan method which you can pass to C-compatible library code; this pointer can be used to call the Apple Dylan method.

The callback mechanism, described in Chapter 4, "Cross-Language Calls," provides a high-level interface to `alien-method`. Normally, you use the techniques described in that chapter, rather than calling `alien-method` directly.

The reference section on page 176 describes this macro in more detail.

Using Preemption and Event Checking

Preemption is the act of interrupting an executing program to allow another task to run temporarily and then resuming the program where it was preempted.

Apple Dylan provides a number of features that allow you to control preemption behavior during the execution of your program. For example, you can use the `%preemption-status-setter` function to enable or disable preemption. You can also use the without-preemption macro to disable preemption during the execution of a particular body of code.

See the section "Preemption Facilities Reference" on page 178 for more information about the built-in preemption facilities.

Event checking is a feature of Apple Dylan in which the runtime periodically preempts your program and executes the `event-check` function. The built-in event-check function simply determines if a break has been requested, and if

so, calls the built-in `break` function. You can provide your own `event-check` function to override this default behavior.

Low-Level Facilities Reference

Reference information for the low-level facilities follows. The low-level facilities are divided into three groups:

- those that allow direct memory access, which are described beginning on page 152
- those that allow temporary memory allocation, which are described beginning on page 171
- those that provide low-level mechanisms for cross-language calls, which are described beginning on page 171.

Reading and Writing Data Using Machine Pointers

The methods in this section allow you to use machine pointers to access memory locations directly. These methods come in pairs: one method in each pair allows you to read information at a particular address by copying that information into an Apple Dylan object; the other method in each pair allows you to copy information from an Apple Dylan object directly into memory.

The functions differ by how many bytes of information they read or write, and how they interpret the information in those bytes.

Three pairs of functions provide access to single bytes in memory:

- The `signed-byte-at` and `signed-byte-at-setter` methods interpret the byte as a signed integer value.
- The `unsigned-byte-at` and `unsigned-byte-at-setter` methods interpret the byte as an unsigned integer value.
- The `character-at` and `character-at-setter` methods interpret the byte as a character value.

These function provide access to memory in two-byte chunks:

- The `signed-short-at` and `signed-short-at-setter` methods interpret the two bytes as a signed integer value.
- The `unsigned-short-at` and `unsigned-short-at-setter` methods interpret the two bytes as an unsigned integer value.
- The `unicode-character-at` and `unicode-character-at-setter` methods interpret the two bytes as a unicode character value.

There are four ways to read and write long words:

- The `signed-long-at` and `signed-long-at-setter` methods interpret four bytes as a signed integer value.
- The `unsigned-long-at` and `unsigned-long-at-setter` methods interpret four byte as an unsigned integer value.
- The `pointer-at` and `pointer-at-setter` methods interpret four bytes as a machine pointer.
- The `ostype-name-at` and `ostype-name-at-setter` methods interpret the four bytes as a four-character string.

Finally, two pairs of methods allow access to strings of bytes:

- The `C-string-at` and `C-string-at-setter` methods allow you to convert between Apple Dylan string objects and null-terminated arrays of bytes.
- The `Pascal-string-at` and `Pascal-string-at-setter` methods allow you to convert between Apple Dylan string objects and arrays of bytes with a leading length byte.

Reading and Writing A Single Byte

The signed-byte-at Method

You can use the `signed-byte-at` method to read a signed byte of data from a particular address.

`signed-byte-at (` *machine-pointer* `) =>` *integer*

machine-pointer A machine pointer containing the address of the byte to read.

integer The integer value of the signed byte at the indicated address.

DESCRIPTION

The Apple Dylan method `signed-byte-at` examines one byte of memory at the address specified by the machine-pointer argument and returns its value as an integer between –128 and 127.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The signed-byte-at-setter Method

You can use the `signed-byte-at-setter` method to write a signed byte of data to a particular address.

`signed-byte-at-setter (` *integer*, *machine-pointer* `) =>` *result*

integer The integer value to write to the indicated address.

machine-pointer A machine pointer containing the address of the byte to write.

result The value of the *integer* argument.

DESCRIPTION

The Apple Dylan method `signed-byte-at-setter` copies the rightmost byte of the binary representation of the *integer* argument to the address specified by the *machine-pointer* argument.

As the function result, the function simply returns the value of the *integer* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
signed-byte-at (my-machine-pointer) := my-new-value;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unsigned-byte-at Method

You can use the `unsigned-byte-at` method to read an unsigned byte of data from a particular address.

`unsigned-byte-at (` *machine-pointer* `) =>` *integer*

machine-pointer A machine pointer containing the address of the byte to read.

integer The integer value of the unsigned byte at the indicated address.

DESCRIPTION

The Apple Dylan method `unsigned-byte-at` examines one byte of memory at the address specified by the machine-pointer argument and returns its value as an integer between 0 and 255.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unsigned-byte-at-setter Method

You use the `unsigned-byte-at-setter` method to write a byte of data to a particular address.

`unsigned-byte-at-setter (` *integer*`,` *machine-pointer* `) =>` *result*

integer	The integer value to write to the indicated address.
machine-pointer	A machine pointer containing the address of the byte to write.
result	The value of the *integer* argument.

DESCRIPTION

The Apple Dylan method `unsigned-byte-at-setter` copies the rightmost byte of the binary representation of the *integer* argument to the address specified by the *machine-pointer* argument.

As the function result, the function simply returns the value of the *integer* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
unsigned-byte-at (my-machine-pointer) := my-new-value;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The character-at Method

You can use the `character-at` method to read one byte of data from a particular address as a character.

`character-at (` *address* `) =>` *character*

address	A machine pointer containing the address of the byte to read.
character	The value of that byte as a character object.

DESCRIPTION

The Apple Dylan method `character-at` examines one byte of memory at the address specified by the *address* argument and returns the value as a character object. This function uses the Macintosh extended ASCII encoding.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The character-at-setter Method

You can use the `character-at-setter` method to write a character to a particular byte.

```
character-at-setter ( character , address ) => result
```

character	The character to write to the indicated address.
address	A machine pointer containing the address to write to.
result	The value of the *character* argument.

DESCRIPTION

The Apple Dylan method `character-at-setter` copies the binary representation of the character specified by the *character* argument to the byte specified by the *address* argument.

This function uses the Macintosh extended ASCII encoding.

As the function result, the function simply returns the value of the *character* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
character-at (my-address) := my-character;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

Reading and Writing Pairs of Bytes

The signed-short-at Method

You can use the `signed-short-at` method to read two bytes of data from a particular address.

`signed-short-at (` *machine-pointer* `) =>` *integer*

machine-pointer A machine pointer containing the address of the bytes to read.

integer The signed integer value of the two bytes at the indicated address.

DESCRIPTION

The Apple Dylan method `signed-short-at` examines two bytes of memory at the address specified by the machine-pointer argument and returns the value as an integer between –32768 and 32767.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The signed-short-at-setter Method

You can use the `signed-short-at-setter` method to write two bytes of data to a particular address.

```
signed-short-at-setter ( integer, machine-pointer ) => result
```

integer The integer value to write to the indicated address.

machine-pointer A machine pointer containing the address of the bytes to write.

result The value of the *integer* argument.

DESCRIPTION

The Apple Dylan method `signed-short-at-setter` copies the two rightmost bytes of the binary representation of the *integer* argument to the address specified by the *machine-pointer* argument.

As the function result, the function simply returns the value of the *integer* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
signed-short-at (my-machine-pointer) := my-new-value;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unsigned-short-at Method

You can use the `unsigned-short-at` method to read two bytes of data from a particular address as an unsigned integer.

```
unsigned-short-at ( machine-pointer ) => integer
```

machine-pointer A machine pointer containing the address of the bytes to read.

integer The integer value of the unsigned bytes at the indicated address.

DESCRIPTION

The Apple Dylan method `unsigned-short-at` examines two bytes of memory at the address specified by the *machine-pointer* argument and returns the value as an integer between 0 and 65535.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unsigned-short-at-setter Method

You use the `unsigned-short-at-setter` method to write a unsigned byte of data to a particular address.

`signed-short-at-setter (` *integer*, *machine-pointer* `) =>` *result*

integer The integer value to write to the indicated address.

machine-pointer A machine pointer containing the address of the byte to write.

result The value of the *integer* argument.

DESCRIPTION

The Apple Dylan method `unsigned-short-at-setter` copies the two rightmost bytes of the binary representation of the *integer* argument to the address specified by the *machine-pointer* argument.

As the function result, the function simply returns the value of the *integer* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
unsigned-short-at (my-machine-pointer) := my-new-value;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unicode-character-at Method

You can use the `unicode-character-at` method to read two bytes of data from a particular address as a Unicode character.

`character-at ( address ) => character`

address	A machine pointer containing the address of the bytes to read.
character	The value of those bytes as a character object.

DESCRIPTION

The Apple Dylan method `character-at` examines two bytes of memory at the address specified by the *address* argument, interprets the value as a Unicode character encoding, and returns the value as a character object.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unicode-character-at-setter Method

You can use the `unicode-character-at-setter` method to write the Unicode encoding of a character to a pair of bytes.

`unicode-character-at-setter ( character, address ) => result`

character	The character to write to the indicated address.

address A machine pointer containing the address to write to.

result The value of the *character* argument.

DESCRIPTION

The Apple Dylan method `character -at-setter` copies the Unicode representation of the character specified by the *character* argument to the pair of bytes indicated by the *address* argument.

As the function result, the function simply returns the value of the *character* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
unicode-character-at (my-address) := my-character;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

Reading and Writing Long Words

The signed-long-at Method

You can use the `signed-long-at` method to read a long word of data from a particular address.

`signed-long-at (` *machine-pointer* `) =>` *integer*

machine-pointer A machine pointer containing the address of the long word to read.

integer The signed integer value of the long word at the indicated address.

DESCRIPTION

The Apple Dylan method `signed-long-at` examines four bytes of memory at the address specified by the *machine-pointer* argument and returns the value as an integer between -2^{31} and $2^{31}-1$.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The signed-long-at-setter Method

You can use the `signed-long-at-setter` method to write four bytes of data to a particular address.

`signed-long-at-setter (` *integer*, *machine-pointer* `) =>` *result*

integer The integer value to write to the indicated address.

machine-pointer A machine pointer containing the address of the bytes to write.

result The value of the *integer* argument.

DESCRIPTION

The Apple Dylan method `signed-long-at-setter` copies the low four bytes of the binary representation of the *integer* argument to the address specified by the *machine-pointer* argument.

As the function result, the function simply returns the value of the *integer* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
signed-long-at (my-machine-pointer) := my-new-value;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unsigned-long-at Method

You can use the `unsigned-long-at` method to read a long word of data from a particular address as an unsigned integer.

`unsigned-long-at (` *machine-pointer* `) =>` *integer*

machine-pointer A machine pointer containing the address of the bytes to read.

integer The integer value of the unsigned long word at the indicated address.

DESCRIPTION

The Apple Dylan method `unsigned-long-at` examines four bytes of memory at the address specified by the *machine-pointer* argument and returns the value as an integer between 0 and 2^{32}–1.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The unsigned-long-at-setter Method

You use the `unsigned-long-at-setter` method to write four bytes of data to a particular address.

`unsigned-long-at-setter (` *integer*`,` *machine-pointer* `) =>` *result*

integer The integer value to write to the indicated address.

machine-pointer A machine pointer containing the address of the bytes to write.

result The value of the *integer* argument.

DESCRIPTION

The Apple Dylan method `signed-long-at-setter` copies the low four bytes of the binary representation of the *integer* argument to the address specified by the *machine-pointer* argument.

You can call this function using the extended assignment syntax, as shown here:

```
unsigned-long-at (my-machine-pointer) := my-new-value;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The pointer-at Method

You can use the `pointer-at` method to read four bytes of data from a particular address as a machine pointer.

`pointer-at (` *address* `) =>` *machine-pointer*

address A machine pointer containing the address of the four bytes to read.

machine-pointer The value of those four bytes as a machine pointer object.

DESCRIPTION

The Apple Dylan method `pointer-at` examines four bytes of memory at the address specified by the *address* argument and returns the value as a machine-pointer object.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The pointer-at-setter Method

You can use the `pointer-at-setter` method to write a machine pointer to a particular address.

`pointer-at-setter (` *machine-pointer*`,` *address* `) =>` *result*

machine-pointer The pointer to write to the indicated address.

address A machine pointer containing the address to write to.

result The value of the *machine-pointer* argument.

DESCRIPTION

The Apple Dylan method `pointer-at-setter` copies the machine pointer specified by the *machine-pointer* argument to the address specified by the *address* argument.

As the function result, the function simply returns the value of the *machine-pointer* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
pointer-at (my-address) := my-machine-pointer;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The ostype-name-at Method

You can use the `ostype-name-at` method to read a four-byte `OSType` value at a particular address into a four-character native Apple Dylan string.

`ostype-name-at ( address ) => string`

address	A machine pointer containing the address of the four bytes to read.
string	A native Apple Dylan string containing four characters corresponding to those four bytes.

DESCRIPTION

The Apple Dylan method `ostype-name-at` reads four bytes of memory at the address specified by the *address* argument, converts the bytes to characters, and returns a native Apple Dylan string object containing those characters.

You use this method primarily to read the values of the Macintosh Toolbox type `OSType`. Some examples include `"CODE"`, `"CDEV"`, `"MENU"` and so on.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The ostype-name-at-setter Method

You can use the `ostype-name-at-setter` method to write a four-character native Apple Dylan string as an `OSType` value at a particular address.

`ostype-name-at-setter ( string , address ) => result`

string	The native Apple Dylan string containing the four characters to write to the indicated address.
address	A machine pointer containing the address to write to.

result The value of the *string* argument.

DESCRIPTION

The Apple Dylan method `ostype-name-at-setter` writes the first four characters contained the native Apple Dylan object provided as the *string* argument as four bytes at the address specified by the *address* argument.

The Apple Dylan string object provided as the string argument must have at least four characters; additional characters are ignored.

As the function result, the function simply returns the value of the *string* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
ostype-name-at (my-address) := "CDEV";
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

Reading and Writing Strings

The c-string-at Method

You can use the `c-string-at` method to read a C-compatible string (a null-terminated array of characters) into a native Apple Dylan string.

`c-string-at (` *address* `) => ` *string*

address A machine pointer containing the address of the C string to read.

string That string as a native Apple Dylan string.

DESCRIPTION

The Apple Dylan method `c-string-at` examines memory at the address specified by the *address* argument for a null-terminated array of bytes, interprets that array as a string of characters, and returns a native Apple Dylan string object containing the same characters.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The c-string-at-setter Method

You can use the `c-string-at-setter` method to write a native Apple Dylan string as a null-terminated array of bytes at a particular address.

`c-string-at-setter (` *string*`,` *address* `) =>` *result*

string	The native Apple Dylan string containing the characters to write to the indicated address.
address	A machine pointer containing the address to write to.
result	The value of the *string* argument.

DESCRIPTION

The Apple Dylan method `c-string-at-setter` writes the characters contained the native Apple Dylan object provided as the *string* argument as a null-terminated array of bytes starting at the location specified by the *address* argument.

As the function result, the function simply returns the value of the *string* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
c-string-at (my-address) := my-native-string;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The Pascal-string-at Method

You can use the `Pascal-string-at` method to read a Pascal-compatible string (an array of bytes with a leading length byte) into a native Apple Dylan string.

`Pascal-string-at ( `*address*` ) => `*string*

address	A machine pointer containing the address of the Pascal-compatible string to read.
string	That string as a native Apple Dylan string.

DESCRIPTION

The Apple Dylan method `pascal-string-at` examines memory at the address specified by the *address* argument for a Pascal-compatible string, and returns a native Apple Dylan string object containing the same characters.

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

The Pascal-string-at-setter Method

You can use the `Pascal-string-at-setter` method to write a native Apple Dylan string as a Pascal-compatible string at a particular address.

`Pascal-string-at-setter ( `*string*`, `*address*` ) => `*result*

string	The native Apple Dylan string containing the characters to write to the indicated address.
address	A machine pointer containing the address to write to.
result	The value of the *string* argument.

DESCRIPTION

The Apple Dylan method `Pascal-string-at-setter` writes the characters contained the native Apple Dylan object provided as the *string* argument as an array of bytes (with an initial length byte) starting at the location specified by the *address* argument.

As the function result, the function simply returns the value of the *string* argument unchanged.

You can call this function using the extended assignment syntax, as shown here:

```
Pascal-string-at (my-address) := my-native-string;
```

SEE ALSO

For examples of this and other machine-pointer methods, see "Using Machine Pointers Directly" on page 146.

Low-Level Macros for Allocating Stack Storage

The with-stack-block Macro

You can use the `with-stack-block` macro to allocate a temporary block of memory.

```
with-stack-block ( variable ( class, size [, keyword-value-pairs ] ),
                   begin
                       body
                   end )
```

variable	A variable name. This variable name is bound to a pointer object that references the temporary block of memory.
class	A machine-pointer or statically-typed-pointer class. The `with-stack-block` macro creates an object of this class and uses that object to hold a pointer to the temporary memory.
size	The number of bytes of temporary memory to allocate.
keyword-value-pairs	Optional list of initialization values used when creating the pointer object.
body	A body of Apple Dylan code that the `with-stack-block` macro executes while the temporary memory is allocated.

DESCRIPTION

This macro allocates a temporary block of memory on the stack, executes any specified code, and then deallocates the temporary memory.

More specifically, this macro allocates *size* bytes of temporary memory, creates a pointer object of the specified *class*, stores a pointer to the temporary memory in the pointer object, and binds the specified *variable* name to the pointer object.

You may also supply optional keyword-value pairs to specify additional initialization information, as appropriate for the specified *class*.

The macro then executes the *body* of code you supply. While this code is executing, the temporary memory remains allocated and can be accessed through the pointer object.

When the body of code finishes executing, Apple Dylan deallocates the temporary memory and the pointer object, and the variable is unbound.

SEE ALSO

For an example using this macro, see "Allocating Stack Storage" on page 148.

For a higher-level version of this macro, see the description of the `with-stack` macro in Part II of this book.

The with-stack-structure Macro

You can use the `with-stack-structure` macro to allocate a temporary block of memory for a C structure.

```
with-stack-structure
        ( variable ( class [, extra-bytes: n ] [, keyword-value-pairs ] ),
          begin
              body
          end )
```

variable A variable name. This variable name is bound to a pointer object that references the temporary block of memory.

class A statically-typed-pointer class that corresponds to a C structure type or a C pointer-to-structure type. The `with-stack-structure` macro creates an object of this class and uses that object to store a pointer to the temporary memory.

n The number of bytes to allocate beyond the number required for the C structure type. This parameter is optional, but if you provide it, it must a non-negative integer.

keyword-value-pairs Optional initialization values for use when creating the statically-typed pointer object.

body A body of Apple Dylan code that the `with-stack-structure` macro executes while the temporary memory is allocated.

DESCRIPTION

Like the `with-stack-block` macro, this macro allocates a temporary block of memory on the stack, executes any specified code, and then deallocates the temporary memory. However, this macro allows you access to the temporary memory as a C structure.

Specifically, this macro allocates enough temporary memory to store the C structure type indicated by the `class` parameter, plus additional bytes if you so specify using the *n* parameter. It creates a pointer object of the specified class and uses this object to store a pointer to the temporary memory block. This pointer object is bound to the specified *variable* name.

You may supply optional keyword-value pairs to specify additional initialization information, as appropriate for the specified *class*.

Apple Dylan then executes the *body* of code you supply. While this code is executing, the temporary memory remains allocated and can be accessed as an imported C structure through the pointer object.

When the body of code finishes executing, Apple Dylan deallocates the temporary memory and the pointer object, and the variable is unbound.

SEE ALSO

For an example using the related `with-stack-block` macro, see "Allocating Stack Storage" on page 148.

For a slightly higher-level version of this macro, see the description of the `with-stack` macro in Part II of this book.

Low-Level Functions for Cross-Language Function Calls

The %Dynamo-alien-call Function

You can use the `%Dynamo-alien-call` compiler function to call a foreign function from Apple Dylan.

```
%Dynamo-alien-call language access-path callee
                   #( { return-location , return-encoding }, )
                   { argument-location , argument-encoding , argument },
```

language The keyword `C:` or the keyword `Pascal:`. This argument indicates the order arguments are pushed onto the stack, the size of Boolean values passed on the stack, and who pops the stack. (On the PowerPC, only the keyword `C:` is valid.)

access-path One of the keywords `pointer:`, `Macintosh-trap:`, `ASLM:`, `direct:`, `XVector:`, or `CFM:`. (Only the last three of these are valid on the PowerPC.) This argument determines how the cross-language call is accomplished; in particular, it indicates how the *callee* argument is interpreted.

callee This argument specifies the cross-language function to call. The `access-path` argument determines how the function is specified.

return-location The location where a return value is passed. For every return value expected from the foreign function, you specify the location where the foreign function returns the value. For 68k machines, you can specify the keyword `stack:` or a keyword representing a machine register (`D0:`, `A1:` and so on). For PowerPC, you can specify a register(`R3:`, `R4:`, `...` `R10:`).

return-encoding A keyword that indicates how the return value is encoded. Possible values include `long:`, `short:`, `byte:`, `char:`, `boolean:`, `pointer:`, `unsigned-long:`, `unsigned-short:`, `unsigned-byte:`, and `unicode-char:`.

argument-location
The location where an argument is passed. For every argument expected by the foreign function, you specify the location where the foreign function expects to find the argument value. For 68k machines, you can specify the keyword `stack:` or a keyword representing a machine register (`D0:`, `A1:` and so on). For PowerPC, you can specify a register(`R3:`, `R4:`, `...` `R10:`) or you can specify the stack using the notation `#(stack:,` *byte-offset*`)` where *byte-offset* is a positive integer.

argument-encoding
A keyword that indicates how the return value is encoded. Possible values include `long:`, `short:`, `byte:`, `char:`, `boolean:`, `pointer:`, `unsigned-long:`, `unsigned-short:`, `unsigned-byte:`, and `unicode-char:`.

argument An Apple Dylan expression that evaluates to an argument value.

DESCRIPTION

This function calls the foreign function indicated by the *callee* parameter using the access path indicated by the *access-path* parameter. The specified access path determines how the *callee* is interpreted:

Access Path	Callee
`pointer:`	The address of the function to call.
`Macintosh-trap:`	The trap instruction, plus 2^{16} to enable error checking.
`direct:`	A module variable whose value is the function to call.
`XVector:`	The address of a CFM XVector to be called through.
`CFM:`	A function pointer for CFM.
`ASLM:`	A function pointer for ASLM.

For each return value returned by the foreign function, you provide a pair of arguments that indicate the location of the return value and the encoding used for the return value.

Similarly, for each argument to the foreign function, you provide three values:

- the location to pass the argument value
- the encoding used for the argument
- the value to pass for the argument

The alien-method Macro

You can use the `alien-method` macro to create a callback function and a C-compatible pointer to the callback function.

```
alien-method class language
             ( { return-location , return-encoding }, )
             ( { argument-location , argument-encoding , parameter-variable }, )
             body
```

class A function-pointer class or an untyped-pointer class.

language The keyword `C:` or the keyword `Pascal:`. (Only the keyword `C:` is valid for PowerPC.) This argument indicates the order arguments are pushed onto the stack, the size of Boolean values passed on the stack, and who pops the stack.

return-location The location where a return value is passed. For every return value returned by the callback function, you specify the location where the value is returned. For 68k machines, you can specify the keyword `stack:` or a keyword representing a machine register (`D0:`, `A1:` and so on). For PowerPC, you can specify a register(`R3:`, `R4:`, . . . `R10:`) or you can specify the stack using the notation `#(stack:,` *byte-offset*`)` where *byte-offset* is a positive integer.

return-encoding A keyword that indicates how the return value is encoded. Possible values include `long:`, `short:`, `byte:`, `char:`, `boolean:`, `pointer:`, `unsigned-long:`, `unsigned-short:`, `unsigned-byte:`, and `unicode-char:`.

argument-location
The location where the callback function expects an argument to be passed. For 68k machines, you can specify the keyword `stack:` or a keyword representing a machine register (`D0:`, `A1:` and so on). For PowerPC, you can specify a register(`R3:`, `R4:`, . . . `R10:`).

argument-encoding
A keyword that indicates how the callback function expects the argument to be encoded. Possible values include `long:`, `short:`, `byte:`, `char:`, `boolean:`, `pointer:`, `unsigned-long:`, `unsigned-short:`, `unsigned-byte:`, and `unicode-char:`.

parameter-variable
The parameter variable name bound to the argument value passed to the callback function.

body The Apple Dylan code that makes up the body of the callback function.

DESCRIPTION

This macro creates a callback function with the specified parameters, return values, and body. The result of this macro is a function pointer of the specified class that you can pass to C-compatible library code.

Preemption Facilities Reference

The preemption facilities allow you to control the **preemption status**—that is, whether preemption is currently **enabled** or **disabled**.

Preemption Constants

Apple Dylan defines two preemption-related constants:

Constant	Meaning
`$preemption-enabled`	Preemption status indicating that preemption is permitted.
`$preemption-disabled`	Preemption status indicating that preemption is disallowed.

You can use these constants to specify preemption status when calling the functions described in the next section.

Preemption Functions

The %preemption-status Function

You can use the `%preemption-status` function to determine the current preemption status.

```
%preemption-status() => result
```

result The current preemption status.

DESCRIPTION

This function returns as its function result a preemption constant indicating whether preemption is currently enabled or disabled.

SEE ALSO

The values returned by this function are discussed in the previous section, "Preemption Constants.".

For an introduction to preemption, see "Using Preemption and Event Checking" on page 151.

To change the current preemption status, use the `%preemption-status-setter` function, described in the next section.

The %preemption-status-setter Function

You can use the `%preemption-status-setter` function to set the current preemption status.

`%preemption-status-setter ( status )`

status The new preemption status.

DESCRIPTION

This function sets the current preemption status to the value indicated by the *status* argument. If *status* indicates that preemption should be enabled, this function immediately checks for and processes any pending event checks. The value of status must be one of the following:

- the result of a call to the built-in function `%preemption-status`
- the value bound to the variable in a `saving-preemption-status` macro call
- one of the constants `$preemption-enabled` and `$preemption-disabled`.

SEE ALSO

To determine the current preemption status, use the `%preemption-status` function, described in the previous section.

To preserve the current preemption status before entering a body of code, use the `saving-preemption-status` macro, described on page 182.

The %preemption-disable Function

You can use the `%preemption-disable` function to disable preemption.

```
%preemption-disable()
```

DESCRIPTION

This function, which takes no arguments and returns no values, disables preemption by setting the current preemption status to `$preemption-disabled`.

SEE ALSO

To enable preemption, use the `%preemption-status-setter` function, described in the previous section.

The %preemption-status-enabled? Function

You can use the `%preemption-status-enabled?` function to determine if a specified value represents enabled preemption status or disabled preemption status.

`%preemption-status-enabled? (` *status* `) =>` *result*

status	The preemption value to test.
result	A Boolean value indicating if value of *status* indicates enabled preemption.

DESCRIPTION

This function takes a single argument and returns true if that argument indicates that preemption is enabled. The value of the *status* argument must be one of the following:

- the result of a call to the built-in `%preemption-status` function
- the value bound to the variable in a `saving-preemption-status` macro call
- one of the constants `$preemption-enabled` or `$preemption-disabled`

SEE ALSO

To determine the current preemption status, use the `%preemption-status` function, described in the previous section.

To preserve the current preemption status before entering a block of code, use the saving-preemption-status macro, described on page 182.

For the discussion of the possible preemption statuses, see "Preemption Constants" on page 178.

Preemption Macros

The macros described in this section allow you to control preemption status during the execution of specific bodies of code.

The without-preemption Macro

You can use the `without-preemption` macro to disable preemption during execution of a specified body of code.

`without-preemption` *body* `=>` *results*

body	The body of code for which you want to disable preemption.
results	The values returned by the *body* of code.

DESCRIPTION

This macro executes the *body* with preemption disabled. When the body of code finishes executing, this macro restores the preemption status to its original value. If preemption is enabled as a result, this macro immediately checks for and processes any pending event checks.

This macro returns the values returned by the *body* of code.

SEE ALSO

For examples of using the preemption facilities, see "Using Preemption and Event Checking" on page 151.

To determine the current preemption status, use the `%preemption-status` function, described in the previous section.

To preserve the current preemption status before entering a block of code, use the `saving-preemption-status` macro, described in the next section.

For the discussion of the possible preemption statuses, see "Preemption Constants" on page 178.

The saving-preemption-status Macro

You can use the `saving-preemption-status` macro to preserve preemption status around the execution of a body of code.

`saving-preemption-status (` [*variable*] `)` *body* `=>` *results*

variable	A variable name to bind to the initial preemption status.
body	The body of code around which to preserve preemption status.
results	The values returned by the *body* of code.

DESCRIPTION

This macro stores the current preemption status, executes the specified body of code, and then restores the preemption status. If the restored status is

preemption enabled, then this macro immediately checks for and processes any pending event checks.

You may supply a *variable* name to this macro; if you do, the macro binds the *variable* to the initial preemption status.

Note that the preemption status is not changed by this form. However, you may set the preemption status within the body, and this macro will restore it to its original value on exit.

This macro returns the values returned by the body.

SEE ALSO

For examples of using the preemption facilities, see "Using Preemption and Event Checking" on page 151.

To determine the current preemption status, use the `%preemption-status` function, described in the previous section.

To disable preemption during execution of a particular body of code, use the `without-preemption` macro, described in the previous section.

For the discussion of the possible preemption statuses, see "Preemption Constants" on page 178.

Event-Checking Facilities Reference

Break-Request Variables and Functions

Apple Dylan defines two variables related to break requests:

Variable	Class	Default Value
`*break-request*`	`<Boolean>`	`#f`
`*break-request-key-chord*`	`<simple-object-vector>`	`command-shift-escape`

The first of these variables is a Boolean value indicating whether there is a break request pending. If you set this variable to a true value, the Apple Dylan runtime executes a program break during the next event check (if you're using

the default event-checking function). The development environment can set this variable; if you provide your own event-checking function, you should examine this variable.

The second of these variables is a sequence of key codes. If the user presses all the keys indicated by the sequence simultaneously, the Apple Dylan executes a program break. (This happens if the keys are being held down during an event check if the default event-checking function is being used.)

The break-request-key-chord? Function

You can use the `break-request-key-chord?` function to determine if the user is requesting a break.

`break-request-key-chord?() =>` *result*

result	A Boolean value indicating whether the break request keys are currently being pressed.

DESCRIPTION

This function determines if the keys indicated by the `*break-request-key-chord*` variable are currently being pressed. If so, this function returns a true value; if not, it returns a false value.

The default event-checking function calls this function to determine if the user is requesting a break.

SEE ALSO

To specify the break request keys, use the `*break-request-key-chord*` variable, described in the previous section.

To request a break programmatically, use the `*break-request*` variable, also described in the previous section.

To see an example using this function, see the description of the `event-check` function on page 185.

Event-Checking Functions

The functions described in this section allow you to override the default event-checking behavior and control the amount of time between preemptive event checks.

The event-check Function

You can use the event-check function to implement preemptive event checking.

```
event-check()
```

DESCRIPTION

While preemption is enabled, Apple Dylan makes periodic preemptive calls to the `event-check` function. When preemption is disabled, Apple Dylan calls this function as soon as preemption becomes enabled again.

By default, `event-check` is a variable name bound to a built-in function. This function tests if the `*break-request*` variable has a true value or if a call to `break-request-key-chord?()` function returns a true value. If either of these is true, and if the `*stand-alone-p*` variable is false, the default `event-check` function calls `break`.

Since `event-check` is a variable name, you can override the default event-checking behavior by assigning a new function object as the value bound to event-check.

Apple Dylan calls the `event-check` function with preemption disabled. If you are supplying your own `event-check` function, and the body of your function is going to run for a significant amount of time and it is safe to do so, you should enable preemption around its body. For example, you could use the following code:

```
saving-preemption-status ()
    %preemption-status() := $preemption-enabled;
    do-long-computation();
end;
```

SEE ALSO

For information about break requests, see "Preemption Constants" on page 178 and "Using Preemption and Event Checking" on page 151.

To determine the current preemption status, use the `%preemption-status` function, described in the previous section.

To disable preemption during execution of a particular body of code, use the `without-preemption` macro, described in the previous section.

For the discussion of preemption status, see "Preemption Constants" on page 178.

The event-check-interval Function

You can use the `event-check-interval` function to determine the current time interval between preemptive event checks.

`event-check-interval() =>` *interval*

interval The current event-check interval.

DESCRIPTION

This function returns the current event check interval, as set by the most recent call to `set-event-check-interval`. A positive number indicates milliseconds, while a value of 0 indicates that preemptive event checking is disabled.

SEE ALSO

To change the event-check interval, use the `set-event-check-interval` function, described in the next section.

The set-event-check-interval Function

You can use the `set-event-check-interval` function to set the time interval between preemptive event checks.

```
set-event-check-interval ( interval )
```

interval The new event-check interval.

DESCRIPTION

This function takes a single argument, which you use to specify the desired length of time between preemptive event checks. A positive number for this argument indicates the number of milliseconds; a zero value indicates that preemptive event checking should be disabled.

For example, calling

```
set-event-check-interval(333)
```

turns on event-checking 3 times per second.

SEE ALSO

To examine the current event-check interval, use the `event-check-interval` function, described in the previous section.

Microsecond Functions

The %microsecond-time Function

You can use the `%microsecond-time` function to determine the current time in microseconds.

```
%microsecond-time() => low high
```

low	The low part of the function result.
high	The high part of the function result.

DESCRIPTION

This function gets the current time in microseconds, and returns it as two small-integers.

The %microsecond-time-difference Function

You can use the `%microsecond-time-difference` function to find the number of microseconds between a start time and an end time.

`%microsecond-time(` *end-low*, *end-high*, *start-low*, *start-high* `) =>` *low high*

end-low	The low part of the end time.
end-high	The high part of the end time.
start-low	The low part of the start time.
start-high	The high part of the start time.
low	The low part of the difference.
high	The high part of the difference.

DESCRIPTION

This function calculates the difference between two times in the format returned by the `%microsecond-time` function, returning the difference in the same format.

The %microsecond-time-microseconds Function

You can use the `%microsecond-time-microseconds` function to convert the two-value time format into a single-value time format

`%microsecond-time-microseconds (` *low high* `) =>` *microseconds*

low	The low part of the time to convert.
high	The high part of the time to convert.
microseconds	The converted, single-value time format.

DESCRIPTION

This function converts a time in the format returned by the `%microsecond-time` function into a single integer representing the number of microseconds.

CHAPTER 7

Low-Level Facilities

CHAPTER 8

Garbage Collection

Contents

About Garbage Collection 193
Overview of Apple Dylan Garbage Collection 193
Ephemeral Garbage Collection 194
Garbage-Collecting Algorithms 195
Large-Object Garbage Collection 196
Using Garbage Collection 197
Using Spontaneous Garbage Collection 197
Requesting Garbage Collection 197
Controlling Garbage Collection Settings 198
Debugging 199
Error Handling 200
Garbage Collection Reference 201
Constants and Variables 201
Functions 202

Although **memory management** is one of the most important aspects of programming, most conventional programming languages offer little memory management support. As a result, each programmer is burdened with the substantial responsibility of managing memory allocation and deallocation. The legendary status of common memory errors, such as dangling pointers and memory leaks, attests to the difficulties inherent in memory management.

Apple Dylan addresses this historical oversight by providing a number of memory-management facilities for you, freeing you to focus on the more distinctive and interesting aspects of your application.

This chapter describes the garbage collection facilities, which handle deallocation of memory for you, eliminating most of the common memory errors (such as dangling pointers and memory leaks). This chapter also discusses memory-related error handling, such as low-memory conditions.

The next chapter, "Finalization," discusses a related set of Apple Dylan facilities, which allow you special control when deallocating objects in memory.

About Garbage Collection

Garbage collection is a method of memory management that periodically examines memory to locate objects (allocated blocks of memory) that are **inaccessible**—that is, no path to them exists from the current state of computation or from any module variable. These blocks of memory, which are called **garbage,** are then reclaimed as available memory.

By automatically providing garbage collection for you, Apple Dylan frees you from the burden of deallocating objects you no longer need. Memory leaks, in which you continue to allocate objects but forget to deallocate them when they are no longer useful, are virtually eliminated. Also, since you don't deallocate objects yourself, you no longer create dangling pointers (references to objects that have been deallocated).

Overview of Apple Dylan Garbage Collection

The garbage collection mechanism provided by Apple Dylan is fast, flexible, and convenient. When memory conditions make it necessary, **spontaneous**

garbage collection works for you, reclaiming memory behind the scenes, undetectable except for a possible delay of a fraction of a second.

Although the default garbage collection behavior eliminates the need for you to worry about memory management, Apple Dylan also provides flexibility in the garbage collection system, allowing you to adjust its behavior when fine-tuning your application's performance. For example, you can disable spontaneous garbage collection during time-critical tasks, and you can invoke **requested garbage collections** at times more appropriate for your application.

Two important points to note:

- Apple Dylan does move objects around in memory during garbage collection. However, Apple Dylan never requires you to rely on the actual address of an Apple Dylan object.
- Not all memory is subject to garbage collection. For example, objects in C libraries are allocated and deallocated using conventional memory management techniques. Garbage collection does not affect these objects.

Apple Dylan also provides some debugging and error-handling facilities relating to garbage collection and memory management. See "Debugging" on page 199 and "Error Handling" on page 200 for more information.

Ephemeral Garbage Collection

Apple Dylan uses a technique called **ephemeral garbage collection** to improve the efficiency of the garbage collection process. This technique divides allocated objects into four **generations:**

- The **youngest generation** contains the most recently created objects.
- The **middle generation** contains older objects—ones that have survived previous garbage collections.
- The **eldest generation** contains the oldest objects—ones that have survived many garbage collections.
- The **static generation** contains objects that are not subject to garbage collection. These objects include objects that exist in libraries (as opposed to objects allocated during program execution), and objects that do not occupy any memory, such as characters and small-integers, which are actually stored within their object references.

The youngest and middle generations are called the **ephemeral generations,** because objects do not belong to these generations for very long; instead they are either garbage collected or promoted.

The middle generation contains two **steps.** Objects promoted to the middle generation are placed in the first step. As they survive garbage collections, they are promoted to the second step, and then eventually to the eldest generation.

The eldest generation is not an ephemeral generation; objects in the eldest generation remain there until they are reclaimed.

Generations are numbered with nonnegative integers. Older generations have larger numbers. The youngest generations number is zero. See the description of the function `object-generation-number`, which is on page 203.

Spontaneous garbage collection is invoked when there is an attempt to allocate memory and memory is exhausted, or when one of the ephemeral generations reaches its size limit.

In general, the younger the generation, the fewer objects it contains and the larger percentage of garbage (inaccessible objects) it contains. For example, the youngest generation tends to be smaller and contain more garbage than the middle generation.

Apple Dylan capitalizes on these statistical differences between generations to increase the overall efficiency of garbage collection. By garbage collecting from younger generations more frequently than older generations, Apple Dylan maximizes the amount of garbage collected and minimizes the amount of memory examined.

Apple Dylan allocates memory for these generations from the Macintosh application heap. You can exert some control over this memory allocation using the variables described in “Controlling Garbage Collection Settings” on page 198.

Garbage-Collecting Algorithms

Apple Dylan provides two different algorithms for freeing memory during garbage collection: the **copying** algorithm and the **compacting** algorithm.

The default algorithm is the copying algorithm. During garbage collection, this algorithm examines regions of memory. For each region, the algorithm copies the accessible objects into a new memory region, and ignores the inaccessible

objects. When an entire region of memory has been examined, all the accessible objects have been copied, so the entire region of memory can be reclaimed.

An alternative is the compacting algorithm. As this algorithm examines a region of memory, it moves the accessible objects to one end of the region. When the entire region has been examined, one end of the region contains accessible objects and the other end of the region is reclaimed.

The copying algorithm is generally faster, though it requires more memory. Apple Dylan automatically switches to the compacting algorithm when there is not enough memory for the copying algorithm.

You should note that under certain virtual memory situations, the extra memory used by the copying algorithm causes this algorithm to be slower than the compacting algorithm. Apple Dylan allows you to choose the algorithm, as described in "Controlling Garbage Collection Settings" on page 198.

Large-Object Garbage Collection

To further improve efficiency, Apple Dylan manages large objects—those larger than a couple hundred bytes—differently than small objects. Separate blocks of memory are allocated for large objects; these blocks are called **large-object spaces** (LOS). When you create a large object, Apple Dylan allocates the memory for it from one of these large-object spaces.

Apple Dylan allows you some control over the allocation of large-object spaces, as described in "Controlling Garbage Collection Settings" on page 198.

Large objects, like small objects, belong to one of four generations. However, the algorithm for garbage collecting large-object memory is different from the copying and compacting algorithms used for small objects. When a large object is reclaimed, its memory is put on a free list. Adjacent free large objects merge into one free memory block. Large objects are not relocated during garbage collection, which improves the efficiency of the garbage collection, but can lead to fragmentation of large-object spaces.

Using Garbage Collection

Using Spontaneous Garbage Collection

By default, you don't have to do anything to take advantage of spontaneous garbage collection. When Apple Dylan detects that an object generation is sufficiently full, it invokes garbage collection for objects of that generation and younger.

You can disable spontaneous garbage collection by setting the value of the Boolean variable `*enable-garbage-collection*`. By setting the value of this variable to `#f`, you prevent Apple Dylan from performing spontaneous garbage collection. As an example, you might want to temporarily disable spontaneous garbage collection to avoid the fraction of a second delay during a time-critical portion of your program.

You should generally re-enable spontaneous garbage collection (by setting the variable to `#t`) promptly, or perform regular requested garbage collections, to avoid exhausted-memory error conditions. See the next section to learn how to request garbage collection.

Requesting Garbage Collection

You invoke a requested garbage collection using the `collect-garbage` function. This function allows you to specify three keyword arguments:

- The `generation:` argument allows you to specify the oldest generation to include during the garbage collection. Apple Dylan automatically includes all younger generations. The default value for this argument is 0, which specifies that only the youngest generation be collected. You can use the constant `$eldest-generation` to specify all generations.
- The `report:` argument allows you to specify whether debugging information regarding the garbage collection should be printed to the Apple Dylan Listener. The default value of this argument is determined by the `*report-garbage-collections*` variable, as discussed in the section "Debugging" on page 199.

- The `compact:` argument allows you to specify whether Apple Dylan should perform copying or compacting garbage collection. The default value is determined by the `*enable-compacting-garbage-collections*` variable, as discussed in the next section.

When trying to optimize performance as much as possible, you can perform occasional requested garbage collections, even if you're taking advantage of spontaneous garbage collection. You can increase the efficiency of garbage collections by requesting them occasionally before spontaneous garbage collection is necessary—that is, when your program has ample free memory available. It is less efficient to perform garbage collection only when memory is nearly exhausted.

Controlling Garbage Collection Settings

Apple Dylan allows you to control certain aspects of garbage collection by setting the values of global variables. Apple Dylan uses the information you provide in these variables during garbage collection and memory allocation.

You've already seen one example: the `*enable-garbage-collection*` variable discussed in "Using Spontaneous Garbage Collection" on page 197 allows you to enable or disable spontaneous garbage collection completely.

The following list describes some other variables you can use to influence garbage collection behavior. Note that you are not required to set the value of any of these variables—Apple Dylan provides default values, and default behavior, for you.

- You can use the `*minimum-free-bytes*` variable to specify the minimum number of bytes that Apple Dylan should leave free in the Macintosh heap. The default value is 100,000.
- You can use the `*minimum-free-block*` variable to specify the minimum block size that Apple Dylan should leave free in the Macintosh heap. The default value is 40,000.
- You can use the `*bytes-per-chunk*` variable to specify the minimum number of bytes that Apple Dylan should allocate from the Macintosh heap when creating a new object block. The default value is 100,000.
- You can use the `*bytes-per-chunk-expansion*` variable to specify the preferred number of bytes that Apple Dylan should allocate from the

Macintosh heap when expanding an existing object block. The default value is 50,000.

- You can use the `*bytes-per-LOS-cluster*` variable to specify the minimum number of bytes that Apple Dylan should allocate from the Macintosh heap when creating a new large-object space. The default value is 200,000.
- You can use the `*bytes-per-LOS-cluster-expansion*` variable to specify the preferred number of bytes that Apple Dylan should allocate from the Macintosh heap when expanding a large-object space. The default value is 50,000.
- You can use the `*enable-compacting-garbage-collection*` variable to specify that Apple Dylan should use the compacting algorithm, rather than the copying algorithm, when garbage collecting. This variable applies to all spontaneous garbage collections. It also applies to requested garbage collections (calls to `collect-garbage`) if you do not specifically override it using a `compact:` argument. The default value for this variable is `#f`, which indicates that the copying algorithm should be used (unless there is not enough memory available).

Debugging

Apple Dylan also provides two variables you may find helpful when debugging and optimizing your program.

The first of these is `*clobber-oldspace*`. When this variable is `#t`, Apple Dylan fills reclaimed memory with the hexadecimal pattern: `#xDEADBEEF`. This distinctive pattern makes it easy for you to identify any references that mistakenly point to reclaimed memory—a situation that should never occur if Apple Dylan is working correctly. The default value for this variable is `#t`. Setting it to `#f` makes garbage collection slightly more efficient.

The second debugging-related variable is `*report-garbage-collections*`, which by default is set to `#f`. If you set this variable to `#t`, Apple Dylan collects pertinent information during each garbage collection and prints the information to the Listener window. (Note that requesting garbage collection reports slows garbage collection down significantly.) This variable applies to all spontaneous garbage collections; it also applies to requested garbage collections (that is, calls to `collect-garbage`) if you do not specifically override it using a `report:` argument.

Here is a sample garbage collection report:

```
Warning: GC: Collected generation 0, GC tick is now 3.
Condemned: 14868 bytes, 4500 bytes survived, 10368 bytes reclaimed.
Compacted 0 blocks (0 moved), saving 0 bytes and reclaiming 0 bytes.
14308 bytes of oldspace was freed (including both garbage and old copies
of survivors).
Scavenger remembered set had 32 entries.
Scanned 5848 bytes, marked 548 objects and 24 locatives.
```

Error Handling

Apple Dylan provides a number of tools you can use to anticipate, avoid, and address the error situations caused by low-memory conditions.

The `memory-available` function allows you to determine the amount of memory, in bytes, available for new objects. You should be aware of the following when interpreting the integer returned by this function:

- The function simply totals the number of available bytes; it does not consider memory fragmentation. You may not be able to allocate a single object of the size indicated by the return value of the function.
- The function does not include memory committed to large-object spaces.
- The function does not include reclaimable space. You may want to request a garbage collection immediately prior to calling this function to ensure the most accurate total.

Apple Dylan also provides the following two variables to help you anticipate low-memory situations:

- You use the `*memory-shortage-level*` variable to indicate the minimum amount of available memory, in bytes, needed for your application to operate normally. Apple Dylan initializes this to 20,000.
- The `*memory-shortage*` variable, initially `#f`, is set to `#t` whenever the memory available immediately after a garbage collection is less than the amount you indicated in the `*memory-shortage-level*` variable. This variable is set back to `#f` if a subsequent garbage collection yields an acceptable amount of available memory.

Apple Dylan also provides two condition classes that signal low-memory conditions:

- The `<memory-shortage>` class, which is a subclass of `<condition>`. This condition is signaled when there is not enough memory to create a requested object.
- The `<memory-exhausted>` class, which is a subclass of `<serious-condition>`. This condition is signaled when there are less than `*memory-shortage-level*` bytes available after garbage collection and `*memory-shortage*` is not already true.

See the "Condition-Handling" chapter of the *Apple Dylan Tutorial* for information about these classes.

Garbage Collection Reference

Constants and Variables

The following table lists the constants and variables relating to garbage collection. The variables are organized by subject, and for each variable, the variable's class and default value are shown:

Enabling Garbage Collection	Class	Default Value
`*enable-garbage-collection*`	`<boolean>`	`#t`
`*enable-compacting-garbage-collection*`	`<boolean>`	`#f`
Specifying Sizes of Memory Blocks		
`*minimum-free-bytes*`	`<small-integer>`	`100,000`
`*minimum-free-block*`	`<small-integer>`	`40,000`
`*bytes-per-chunk*`	`<small-integer>`	`100,000`
`*bytes-per-chunk-expansion*`	`<small-integer>`	`50,000`
`*bytes-per-LOS-cluster*`	`<small-integer>`	`200,000`
`*bytes-per-LOS-cluster-expansion*`	`<small-integer>`	`50,000`

Enabling Garbage Collection	Class	Default Value
Identifying Memory Generations		
`$eldest-generation`	`<small-integer>`	`4`
Debugging and Error Handling		
`*report-garbage-collections*`	`<boolean>`	`#f`
`*clobber-oldspace*`	`<boolean>`	`#t`
`*memory-shortage*`	`<boolean>`	`#f`
`*memory-shortage-level*`	`<boolean>`	`20,000`

Notice that the value of the `$eldest-generation` constant is 4, rather than 2 as you might expect (since 0 indicates the youngest generation and 1 indicates the middle generation). Apple Dylan reserves the generation numbers 2 and 3 for possible future use as additional ephemeral generations.

The section "Using Garbage Collection," which begins on page 197, provides examples using these variables and discusses how you can change the values of these variables to control various aspects of garbage collection.

Functions

Apple Dylan provides functions you can use

- to invoke requested garbage collection
- to determine the generation to which a specific object belongs
- to determine the amount of available memory

The collect-garbage Function

You can use the `collect-garbage` function to invoke a requested garbage collection. The function accepts three optional keyword arguments.

`collect-garbage (` *#key generation report compact* `)`

generation An optional keyword parameter (keyword `generation:`) that specifies the oldest generation to garbage collect. The default value is 0, indicating that only the youngest generation should be garbage collected.

report An optional keyword parameter (keyword `report:`) that indicates whether Apple Dylan should report relevant information about the garbage collection to the Listener window. The default value is the current value of the `*report-garbage-collections*` variable.

compact An optional keyword parameter (keyword `compact:`) that indicates whether Apple Dylan should use the compacting, rather than the copying, garbage-collecting algorithm. The default value is the current value of the `*enable-compacting-garbage-collections*` variable.

DESCRIPTION

Calls to this function invoke a requested garbage collection. The *generation* argument allows you to specify the oldest generation to be collected. All younger generations are also collected.

SEE ALSO

For a discussion of when you might use this function, see "Requesting Garbage Collection" on page 197.

The object-generation-number Method

You can use the `object-generation-number` method to determine which generation a specific object belongs to.

`object-generation-number(` *object* `) =>` *generation*

object The object you want to determine the generation of.

generation An integer indicating the object's generation.

DESCRIPTION

This function returns a value that indicates the generation to which the specified *object* belongs:

- 0 indicates the youngest generation
- 1 indicates the middle generation
- 4 (the value of `$eldest-generation`) indicates the eldest generation
- 5 indicates the static generation

If an object's generation number is less than `$eldest-generation`, it can change at any time that a garbage collection can occur. However, it can only increase.

The memory-available Function

The `memory-available` function allows you to determine the amount of memory, in bytes, available for new objects.

```
memory-available ( ) => size
```

size The number of bytes of available memory.

DESCRIPTION

This function returns the number of bytes of available memory. It does not take into consideration fragmentation, large-object memory, or reclaimable memory.

SEE ALSO

For a discussion of this function, see "Error Handling" on page 200.

CHAPTER 9

Termination

Contents

About Termination 207
Termination and Garbage Collection 207
Termination Processing 208
Using Termination 209
Enabling Termination For an Object 209
Termination Processing 209
Specifying Termination Behavior 210
Resurrecting an Object 210
Termination Reference 211
Variables 211
Functions 211

About Termination

Termination is the facility provided by Apple Dylan that allows an object to perform additional computation after it becomes inaccessible but before it is reclaimed by the garbage collector and its memory freed.

As an example, suppose you create a class whose objects contain slots that point to memory allocated directly (using the Toolbox routine `NewPtr`, for example). For those objects, you might want to specify termination behavior in which you deallocate that memory since the garbage collector does not manage it and will not reclaim it.

By default, objects have no termination behavior: when Apple Dylan determines during garbage collection that an object is inaccessible, the object is reclaimed immediately and its memory is freed.

You can override this default behavior by **enabling termination** for an object, (which indicates to Apple Dylan that the object is **terminatable**) and by specifying **termination behavior** (which Apple Dylan invokes when terminating the object).

Termination and Garbage Collection

During garbage collection, Apple Dylan searches for inaccessible objects. Apple Dylan processes each inaccessible object in one of two ways:

- If the object does not have termination enabled, it is reclaimed and its memory freed.
- If the object does have termination enabled, it is not reclaimed immediately. Instead, it is added to a queue for later termination processing, as described in the next section.

Note that some objects are not subject to garbage collection (such as small integers). Therefore, these objects are never terminated, even if you specifically enable termination for them.

Termination imposes a small amount of overhead on the garbage collection process:

- **Memory requirements.** Termination requires approximately 4 bytes of memory times the maximum number of objects that ever have termination enabled simultaneously.
- **Time requirements.** Garbage collecting a terminatable object requires slightly more time than garbage collecting other objects, but the difference is negligible. (The additional processing time is less than the time already required to process the object.)

Termination Processing

During garbage collection, Apple Dylan creates a **termination queue**—a queue that contains the terminatable objects that the garbage collector determined to be inaccessible.

After garbage collection completes, Apple Dylan performs **termination processing**—that is, Apple Dylan executes the termination behavior for each object in the termination queue.

Automatic termination processing occurs by default immediately after garbage collection completes. (Actually, if preemption is not enabled, automatic termination processing is delayed until the next time preemption is enabled.) You can enable or disable automatic termination processing. You can also invoke termination processing explicitly, using **requested termination processing.**

During termination processing, the order in which the queued objects are terminated is unspecified, with one exception. If one terminatable object references a second terminatable object, but both objects are otherwise unreachable, the first object (the one containing the reference) is terminated before the second object (the referenced one).

Unless a reference to the object is created by the termination behavior, the object will be reclaimed by the next garbage collection.

Using Termination

Enabling Termination For an Object

You use the `terminate-when-unreachable` function to enable termination for a specific object. This function takes one argument—the object for which you want to enable termination—and returns that object as the function result:

```
terminate-when-unreachable(my-terminatable-object);
```

This function enables termination for the object—that is, it identifies the object as needing termination processing. If Apple Dylan determines during garbage collection that this object is inaccessible, the object is not immediately reclaimed. Instead, Apple Dylan disables termination for the object and places it in the termination queue for later termination processing.

You can call this function multiple times on the same object. When the object becomes inaccessible, Apple Dylan executes the object's termination behavior a corresponding number of times. However, multiple execution of the object's termination behavior is not guaranteed in future versions of Dylan and so you should not depend on it.

Termination Processing

By default, you don't have to do anything to enable automatic termination processing. During garbage collection, Apple Dylan automatically queues inaccessible objects that have termination enabled. Automatic termination processing occurs immediately following the garbage collection (if preemption is enabled; otherwise, automatic processing occurs the next time that preemption becomes enabled).

You can disable automatic termination processing by setting the value of the Boolean variable `*enable-automatic-termination*` to `#f`. By setting the value of this variable to `#f`, you prevent Apple Dylan from automatically processing the termination queue after garbage collection.

You can re-enable automatic termination processing by setting the value of this variable to `#t`.

You can also explicitly invoke termination processing by calling the `drain-termination-queue` function. This function, which takes no arguments, calls the `terminate` function on each object awaiting termination in the termination queue.

Specifying Termination Behavior

Termination processing consists of calling the `terminate` function on each object in the termination queue. You specify termination behavior for your terminatable objects by adding specialized `terminate` methods to the `terminate` generic function. For example:

```
define method terminate (terminating-object :: <my-terminatable-class>)
    my-final-processing (terminating-object);
end method;
```

For Apple Dylan to call this method:

1. you must enable termination for an object of class `<my-terminatable-class>`
2. garbage collection must determine that the object is inaccessible
3. termination processing must be invoked

During termination processing, Apple Dylan calls the `terminate` generic function on your object, which invokes your specialized `terminate` method.

Resurrecting an Object

Generally, once an object has been terminated, it is reclaimed during the next garbage collection of its generation. However, there are two ways in which you can avert this impending reclamation, thereby **resurrecting** the object.

- You can use the `terminate` method to create a reference to the object:

```
define method terminate (terminating-object :: <my-terminatable-class>)
    *global-reference* := terminating-object;
end method;
```

This `terminate` method stores a reference to the terminating object in a module variable. Therefore, during the next garbage collection, this object is accessible and is not reclaimed.

- You can also use the `terminate` method to re-enable termination for the object:

```
define method terminate (terminating-object :: <my-terminatable-class>)
    terminate-when-unreachable(terminating-object);
end method;
```

This `terminate` method re-enables termination for the terminating object. Therefore, during the next garbage collection, this object is not reclaimed, but instead is queued up for another round of termination processing.

Termination Reference

Variables

Apple Dylan defines one termination-related variable:

Variable	Class	Default Value
`*enable-automatic-termination*`	`<Boolean>`	`#t`

You can use this variable to enable or disable automatic termination processing, as described in “Termination Processing” on page 209.

Functions

The functions described in this section allow you to enable termination for an object, specify termination behavior, and invoke termination processing.

The terminate-when-unreachable Function

You can use the `terminate-when-unreachable` function to enable termination for a specific object. This function requires one argument and returns that argument as the function result.

`terminate-when-unreachable ( object ) => object`

object The object for which you want to enable termination.

DESCRIPTION

This method marks the specified object as terminatable. When Apple Dylan garbage collection determines that the object is inaccessible, it is queued for termination processing.

SEE ALSO

For examples of function, see "Enabling Termination For an Object" on page 209.

To specify termination behavior, see the description of the `terminate` generic function in the next section.

The terminate Generic Function

Apple Dylan calls the `terminate` generic function on each object in the termination queue during termination processing.

`terminate ( object )`

object The object Apple Dylan is terminating.

DESCRIPTION

You provide specialized method implementations for this generic function to specify the termination behavior for your terminatable objects.

SEE ALSO

For examples of specializing this generic function, see "Specifying Termination Behavior" on page 210.

The drain-termination-queue Function

You use the `drain-termination-queue` function to invoke requested termination processing.

```
drain-termination-queue ( )
```

DESCRIPTION

This function invokes termination processing, calling the `terminate` generic function on each object in the termination queue. Note that this function is only useful if automatic termination processing is disabled.

SEE ALSO

For examples of this function, see "Termination Processing" on page 208.

CHAPTER 9

Termination

PART TWO

Framework Reference

PART TWO

CHAPTER 10

Introduction

Contents

Conceptual Model 219
Framework Foundation 220
Event Handling 223
User Interface Support 228
Documents and Data Interchange 236

Introduction

This chapter introduces the conceptual model of Apple Dylan Framework and briefly describes the the classes provided by Framework.

Conceptual Model

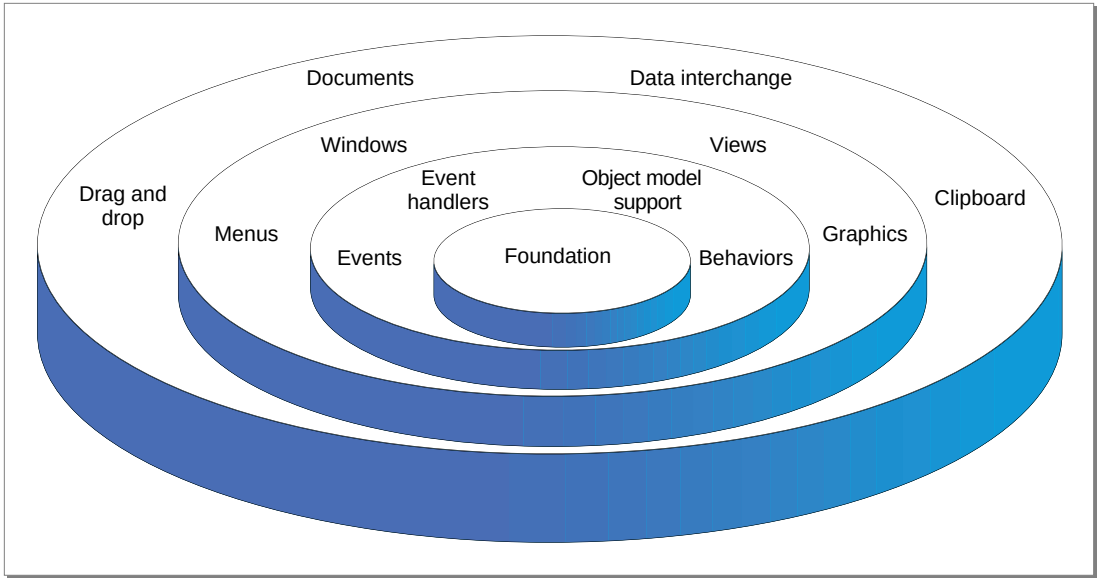
Conceptually, the Framework can be divided in three major subsystems:

- event handling support.
- support for the Macintosh user interface.
- support for data manipulation.

The three major subsystems are built on a small foundation that provides additional basic services beyond those provided by the Dylan language and environment.

Figure 10-1 shows the three subsystems as rings around the foundation.

Figure 10-1 Framework subsystems



The conceptual model is designed to show related categories of features and provide a way to identify the Framework classes that implement the features. This conceptual model does not specify the interaction between the subsystems; for example, it does not show that windows and views are event handlers or that documents might use a view's graphics buffer (the relationship between a window, view, graphics, and document) to store data. These kinds of interactions are described in the chapter that discusses a feature.

The following sections discuss the foundation and the three subsystems. Use the information in these sections as an overview of how the Framework is organized and to locate classes of interest. The classes that are part of a subsystem are identified along with the way they are intended to be used. Three codes identify the use as follows:

- Make—Create (`make`) objects directly from this class. There is no need to define a subclass. For example, you create `<window>` objects, however, you need not create subclasses from the `<window>` class.
- Define subclass—Define a subclass of the described class and then create objects from your subclass. Creating an object of the class directly is either not possible or does not implement the protocol completely enough to do all that is necessary to do. For example, you must define a subclass of `<behavior>` so that you can implement the actions you want your behavior to perform.
- Specialize only—The Framework defines the class and creates the necessary objects for you. You only need to define methods whose parameters specialize on the class. For example, the Framework creates `<mouse-moved-event>` objects at the appropriate times. You only need to define methods whose parameters specialize on the `<mouse-moved-event>` class.

Classes not used in these three ways are internal to the Framework. Either their names are not exported or, if they are, they are intended to be used only by other parts of the Framework. You should not specialize on an internal class or access its slots.

Framework Foundation

The foundation provides general services not implemented in the Dylan language and environment. (Services that are provided by Dylan include

collections and memory management, thus it is not necessary for the Framework to provide them.) Services provided by the Framework include:

- streams
- resource management
- file I/O
- error reporting for conditions
- geometry for points, rectangles, and regions
- dependency tracking and notification
- environmental support, such as determining gestalt, and so on
- utilities

Many, but not all of these services are defined by classes. Table 10-1 shows the classes that support streams, resources, and files.

Table 10-1 Files, resources, and streams

Class	Description	Use	See Page
`<stream>`	A stream of data.	Define subclass	517
`<handle-stream>`	A stream that is read from or written to a handle.	Make	519
`<file-stream>`	A stream that is read from or written to a file.	Make	518
`<object-stream>`	A stream of Dylan objects.	Define subclass	521
`<file-object-stream>`	A stream of Dylan objects that is read from or written to a file.	Make	521
`<handle-object-stream>`	A stream of Dylan objects that is read from or written to a handle.	Make	521
`<handle-string>`	A string in a handle.	Make	639
`<resource-string>`	A string in a resource.	Make	640

Table 10-1 Files, resources, and streams

Class	Description	Use	See Page
<file>	A Macintosh file.	Define subclass	496
<slot-spec>	Internal. A specification for streaming a slot.	—	523
<class-spec>	Internal. A specification for streaming a class.	—	522
<framework-library>	Internal. Manages a list of resources for a Framework library.	—	649

Table 10-2 shows the classes that support error reporting and debugging.

Table 10-2 Error reporting

Class	Description	Use	See Page
<os-error>	An error typically reported by the operating system and handled by the Framework.	Make	643
<framework-error>	An error typically handled by the Framework.	Define subclass	642
<debugger-behavior>	A behavior that provides the Debug menu and actions associated with its menu items.	—	645

Table 10-3 shows the classes that support general geometric operations. For the classes that support QuickDraw geometric operations, see Table 10-8 on page 229.

Table 10-3 Geometry

Class	Description	Use	See Page
<rect>	A rectangle.	Make	635
<region>	A region.	Specialize only	635

Event Handling

Event handlers are objects that can respond and handle events. For complete information about event handling, see page 241. Remember that many user interface elements identified in the section "User Interface Support" on page 228 are also event handlers, as are documents (see page 236). Table 10-4 shows the general event-handling classes.

Table 10-4 General event-handling and dispatching classes

Class	Description	Use	See Page
<event-handler>	An object capable of responding to an event. Event handlers include documents, windows, views, controls, and so on.	Specialize only	263

Table 10-4 General event-handling and dispatching classes

Class	Description	Use	See Page
`<main-handler>`	The event handler responsible for events not handled by other event handlers.	Specialize only	265
`<behavior>`	An object that is capable of responding to an event when it is associated with an event handler,	Define subclass	266
`<idler-reference>`	Internal. A list of event handlers that can execute between event processing.	—	

Events are notifications from the Macintosh OS that the state of the application or the environment has changed. Table 10-5 shows the objects created by the Framework after calling `WaitNextEvent` from the Framework's main event loop.

Table 10-5 Toolbox events

Class	Description	Use	See Page
`<event>`	An event representing a change in the state of the hardware or operating system.	Specialize only	267
`<toolbox-event>`	An event defined by the Macintosh toolbox `EventRecord` data structure.	Specialize only	268
`<window-event>`	A window event, such as an activate, update, or mouse event.	Specialize only	271
`<mouse-event>`	A mouse event, such as a mouse-down, mouse-up, or mouse-moved event.	Specialize only	273
`<generic-mouse -down-event>`	A mouse-down event, whether or not it is in an active window.	Specialize only	273
`<mouse-down-event>`	A mouse-down event within an active window.	Specialize only	273

Table 10-5 Toolbox events

Class	Description	Use	See Page
`<background-mouse-down-event>`	A mouse-down event within an inactive window.	Specialize only	273
`<mouse-up-event>`	A mouse-up event.	Specialize only	273
`<key-event>`	A key event, such as a key-up or key-down event.	Specialize only	270
`<key-down-event>`	A key-down event.	Specialize only	270
`<key-up-event>`	A key-up event.	Specialize only	270
`<activate-event>`	An activate event.	Specialize only	271
`<update-event>`	An update event.	Specialize only	271
`<system-event>`	A system event, such as a suspend or resume event.	Specialize only	271
`<suspend-event>`	A suspend event.	Specialize only	271
`<resume-event>`	A resume event.	Specialize only	271
`<mouse-moved-event>`	A mouse-moved event.	Specialize only	273
`<disk-event>`	A disk event.	Specialize only	270
`<become-target-event>`	An event that notifies event handlers in the target chain of a new target.	Specialize only	274
`<resign-target-event>`	An event that notifies event handlers in the target chain of a change in the target.	Specialize only	275

Introduction

Object model support includes support for scripting and recordability. It also uses and responds to Apple events. Table 10-6 shows the classes involved with object model support.

Table 10-6 Object model support

Class	Description	Use	See Page
`<ae-desc>`	An Apple event descriptor record.	Define subclass	607
`<relative-descriptor>`	An Apple event descriptor record.	Make	611
`<index-descriptor>`	An Apple event descriptor record.	Make	611
`<range-descriptor>`	An Apple event descriptor record.	Make	611
`<logical-descriptor>`	An Apple event descriptor record.	Make	611
`<comparison -descriptor>`	An Apple event descriptor record.	Make	611
`<token>`	An Apple event descriptor record used to resolve an object.	Make	612
`<object-specifier>`	An Apple event data structure that refers to a particular Dylan object in your application.	Make	610
`<scripting-event>`	An Apple event.	Define subclass	612
`<required-event>`	A required suite Apple event.	Define subclass	612
`<open-application -event>`	An Open Application Apple event.	Make	612
`<open-documents -event>`	An Open Documents Apple event.	Make	612
`<print-documents -event>`	A Print Documents Apple event.	Make	612
`<quit-event>`	A Quit Application Apple event.	Make	612
`<core-event>`	A core suite Apple event.	Define subclass	612

Table 10-6 Object model support

Class	Description	Use	See Page
<get-data-event>	A Get Data Apple event.	Make	612
<set-data-event>	A Set Data Apple event.	Make	612
<create-element-event>	A Create Element Apple event.	Make	612
<save-event>	A Save Apple event.	Make	612
<close-event>	A Close Apple event.	Make	612
<delete-event>	A Delete Apple event.	Make	612
<misc-standards-event>	A miscellaneous standards suite Apple event.	Define subclass	612
<select-event>	A Select Apple event.	Make	612
<undo-event>	An Undo Apple event.	Make	612
<redo-event>	A Redo Apple event.	Make	612
<cut-event>	A Cut Apple event.	Make	612
<copy-event>	A Copy Apple event.	Make	612
<paste-event>	A Paste Apple event.	Make	612
<external-reference-table>	Internal. Used for managing references to Dylan objects passed to non-Dylan functions.	—	630

Designators are used to identify a piece of data in data interchange or in an Apple event object; for example, a word inside a paragraph of a document.

Table 10-7 shows these designator classes provided by the Framework.

Table 10-7 Designators

Class	Description	Use	See Page
`<designator>`	A temporary reference used to specify data in an object for object model support.	Define subclass	561
`<selection-designator>`	A designator for the `cSelection` class of Apple event objects.	Make	562
`<text-designator>`	A designator for the `cText` class of Apple event objects.	Make	563
`<character-designator>`	A designator for the `cChar` class of Apple event objects.	Make	563
`<range-designator>`	Designates a range of bytes.	Make	564
`<offset-designator>`	Designates bytes of data offset from a starting point.	Make	564
`<property>`	A designator for the `cProperty` class of Apple event objects.	Define subclass	616
`<document-property>`	A designator for a specific property of the `cDocument` class of Apple event objects.	Make	616
`<window-property>`	A designator for a specific property of the `cWindow` class of Apple event objects.	Make	618
`<file-property>`	A designator for a specific property of the `cFile` class of Apple event objects.	Make	617
`<text-view-property>`	A designator for a specific property of the `cText` class of Apple event objects.	Make	617
`<text-view-selection -property>`	A designator for a specific property of the `cSelection` class of Apple event objects.	Make	618

User Interface Support

User interface support includes the following areas:

- graphics, including QuickDraw and image-related geometry
- windows, including support for scrolling and tracking
- views, including text and grid views
- menus
- adorners
- context-sensitive undo and redo

Table 10-8 shows the classes that support graphics.

Table 10-8 Graphics

Class	Description	Use	See Page
`<frame>`	The location and extent of a window, view, or graphics port.	Specialize only	319
`<graphics-port>`	A port through which an image is rendered.	Specialize only	320
`<text-style>`	A text style, such as font information.	Specialize only	322
`<qd-region>`	A QuickDraw region.	Make	324
`<qd-text-style>`	Font information for QuickDraw text.	Make	322
`<qd-graphics-port>`	A QuickDraw GrafPort.	Specialize only	320
`<qd-graphics -buffer>`	A buffer for graphics based on a graphics world (GWorld).	Make	320
`<qd-graphics -buffer>`	A buffer for graphics based on a graphics world (GWorld).	Make	320
`<rgb-color>`	An RGB color specification.	Make	324

Table 10-9 shows classes related to windows and general-purpose views, including view determiner classes and classes that provide scrolling and tracking support.

Table 10-9 Windows and views

Class	Description	Use	See Page
<window>	A window.	Make	407
<window-context>	An event handler that manages one or more windows.	Define subclass	406
<view>	An area within a window, typically used for drawing or responding to events.	Define subclass	355
<view-determiner>	A rule that determines how the location and size of a view changes in relation to a change in the extent of its superview or subviews.	Define subclass	362
<has-super-view -bounds>	A rule that specifies that a view has the same location and size as its superview.	Make	362
<super-view-horiz -relative-determiner>	A rule that specifies proportional resizing of a view's extent in relation to a horizontal change in its superview.	Make	362
<super-view-vert -relative-determiner>	A rule that specifies proportional resizing of a view's extent in relation to a vertical change in its superview.	Make	362
<super-view-relative -determiner>	A rule that specifies proportional resizing of a view's extent in relation to a change in the extent of its superview.	Make	362
<horiz-scroll-bar -determiner>	A rule that specifies how a horizontal scroll bar control can change in the extent of its superview.	Make	362
<vert-scroll-bar -determiner>	A rule that specifies how a vertical scroll bar control can change in the extent of its superview.	Make	362

Table 10-9 Windows and views

Class	Description	Use	See Page
<grow-icon -determiner>	A rule that determines the position of the size box if its superview's extent changes.	Specialize only	413
<grow-icon>	A size box for a window.	Specialize only	413
<scroller>	A view that represents an area that can be larger than the area of its associated window, and thus be scrollable.	Make	471
<tracker>	An object that provides visual feedback for mouse movements.	Define subclass	473

Table 10-10 shows classes that implement TextEdit features and text entry in dialog boxes.

Table 10-10 Text views

Class	Description	Use	See Page
<text-view>	A view for TextEdit operations.	Make	375
<edit-text>	A view for an editable text field used typically in a dialog box.	Make	378
<number-text>	A view for an editable text field in which the text is constrained to digits used typically in a dialog box.	Make	379
<edit-text-dialog -filter>	A behavior that handles special characters (tab, return, escape, and so on) while editing text in a dialog box.	Specialize only	436

Table 10-11 shows classes that implement grid views.

Table 10-11 Grid views

Class	Description	Use	See Page
<grid-view>	A view that is subdivided into cells.	Define subclass	391
<list-view>	A view that is a single column of cells.	Make	394
<text-grid-view>	A grid view whose cells display text.	Make	394
<text-list-view>	A list view whose cells display text.	Make	395
<grid-arrow-key-behavior>	A behavior that selects cells in a grid by using the arrow keys.	Make	396
<grid-select-behavior>	A behavior that selects cells in a grid for editing.	Make	396

Table 10-12 shows classes for implementing dialog boxes and controls.

Table 10-12 Dialogs and controls

Class	Description	Use	See Page
<tabber>	A behavior that provides tabbing between fields in a dialog box.	Define subclass	435
<dialog-behavior>	A behavior that implements modal dialogs.	Make	433
<static-text>	A view for unchangeable text.	Make	439
<simple-icon>	An icon.	Make	438
<ctl-mgr-control>	A Control Manager control.	Make	446
<ctl-mgr-button>	A Control Manager push button control.	Make	449
<ctl-mgr-check-box>	A Control Manager checkbox control.	Make	449

Table 10-12 Dialogs and controls

Class	Description	Use	See Page
<ctl-mgr-radio>	A Control Manager radio button control.	Make	450
<ctl-mgr-scroll-bar>	A Control Manager scroll bar control.	Make	451
<ctl-mgr-popup>	A Control Manager popup menu control.	Make	452
<cluster>	A view for a group of controls that operate together, such as radio buttons.	Make	437
<control>	A control.	Define subclass	440
<button>	A push button control.	Define subclass	443
<check-box>	A checkbox control.	Define subclass	443
<radio>	A radio button control.	Define subclass	443
<scroll-bar>	A scroll bar control.	Define subclass	444
<popup>	A popup menu control.	Define subclass	445
<control-event>	An event associated with a control.	Specialize only	454
<button-event>	An event associated with a push button control.	Specialize only	454
<check-box-event>	An event associated with a checkbox control.	Specialize only	454
<radio-event>	An event associated with a radio button control.	Specialize only	454

Table 10-12 Dialogs and controls

Class	Description	Use	See Page
<scroll-bar-event>	An event associated with a scroll bar control.	Specialize only	454
<popup-event>	An event associated with a control.	Specialize only	454
<popup-menu>	Internal. A menu inside a Control Manager popup menu control.	—	

Table 10-13 shows classes that implement menus, including menu event handling.

Table 10-13 Menus

Class	Description	Use	See Page
<menu-bar>	Internal. The menu bar.	—	.
<menu-element>	A menu element, such as a menu item or hierarchial menu.	Specialize only	292
<menu>	A menu, such as the File menu.	Make	297
<menu-item>	A menu item, such as the Save item of the File menu.	Make	294
<separator-item>	A divider between menu items.	Make	297
<apple-menu>	The Apple menu.	Specialize only	300
<menu-event>	An event that affects a menu item.	Define subclass	301

Table 10-14 shows classes that implement adorners.

Table 10-14 Adorners

Class	Description	Use	See Page
`<adorner>`	A visible adornment to a view; for example, a border.	Define subclass	360
`<draw-adorner>`	An adorner that controls the order in which other adorners are drawn.	Specialize only	360
`<hilite-adorner>`	An adorner that controls the order in which other adorners are highlighted.	Specialize only	360
`<frame-adorner>`	An outline just inside a rectangular view.	Make	360
`<black-background-adorner>`	A rectangle behind a rectangular view.	Make	360
`<default-button-adorner>`	A rounded rectangle, usually associated with a default button.	Make	455
`<tracker-adorner>`	Internal. Responsible for handling tracking feedback during autoscrolling.	—	474

Table 10-15 shows classes that implement context-sensitive undo and redo.

Table 10-15 Context undo and redo

Class	Description	Use	See Page
`<command>`	A command that handles undo and redo operations.	Define subclass	540
`<command-context>`	Event handlers to which an undo or redo operation applies.	Make	539
`<typing-command>`	A command that handles undo and redo for text.	Specialize only	542

Table 10-15 Context undo and redo

Class	Description	Use	See Page
<text-style-command>	A command that implements undo and redo for text styles.	Define subclass	544
<font-command>	A command that implements undo and redo for the kind of font in a text style.	Make	545
<font-size-command>	A command that implements undo and redo for the font size in a text style.	Make	545
<font-style-command>	A command that implements undo and redo for the font style (bold, italic, and so on) in a text style.	Make	545

Documents and Data Interchange

Documents tie the application's data to windows, and thus views, and to files. You can store a document's data in Dylan collection classes, in a graphics buffer associated with a view, or in your own data model.

Table 10-16 shows the document class.

Table 10-16 Documents

Class	Description	Use	See Page
<document>	An object that represents the relationship between data, views and files.	Define subclass	485

Table 10-17 shows the classes that provide general support for transferring data to and from the clipboard or within or between applications via drag and drop.

Table 10-17 Data interchange

Class	Description	Use	See Page
<typed-data>	Data whose data type, such as text or pict, is specified.	Define subclass	558
<typed-pointer>	Typed data that is referenced by a pointer.	Make	559
<typed-handle>	Typed data that is referenced by a handle.	Make	560
<data-item>	Data that may be represented with several data types.	Make	560

Table 10-18 shows the classes that implement clipboard support.

Table 10-18 Clipboard

Class	Description	Use	See Page
<clipboard-behavior>	A behavior that responds to clipboard events, such as cut, copy, paste, or clear. It is automatically associated with text views.	Define subclass	594
<clipboard-command>	A command to handle a clipboard event.	Specialize only	596
<cut-command>	A command to handle a cut event.	Specialize only	596
<paste-command>	A command to handle a paste event.	Specialize only	596

Table 10-18 Clipboard

Class	Description	Use	See Page
<clear-command>	A command to handle a clear event.	Specialize only	596
<clipboard-flavor>	The kind of data stored in the scrap.	Define subclass	595
<clipboard>	The clipboard.	Specialize only	594

Table 10-19 shows the classes that implement drag and drop support.

Table 10-19 Drag and drop

Class	Description	Use	See Page
<drag>	A dragging operation.	Make	577
<drag-item>	An item selected in a drag operation.	Make	578
<drag-flavor>	The data in a drag item, which is represented in a specific data type.	Make	579
<drag-event>	Internal. A mouse event interpreted as a drag.	—	580
<track-drag-event>	Internal. An event which indicates that items are being dragged through a window.	—	580
<receive-drag-event>	Internal. An event which indicates that an item was dropped on a window.	—	580

CHAPTER 11

Event Handling

Contents

About Event Handling 242
- Kinds of Events 242
- Event Handlers 246
- The Target Chain 248
- Responding to Events 250
  - Event Handling Using the Target Chain 251
  - Mouse Event Handling 252
- Changing the Target 254
- Behaviors and Event Handling 254

Using Event Handling Objects 259
- Adding an Event Handler to the Target Chain 259
- Defining a do-event Method 260
- Implementing a Behavior 260
- Specifying Actions When the Target Changes 263

Event Handling Objects Reference 263
- The <event-handler> Class 263
- The <main-handler> Class 265
- The <behavior> Class 266
- The <event> Class 267
- Toolbox Event Classes 268
  - The <disk-event> Class 270
  - Key Event Classes 270
  - System Event Classes 271
  - Window Event Classes 271
  - Mouse Event Classes 273
  - The <become-target-event> Class 274

The <resign-target-event> Class 275
Functions for Event Handling 276

Event Handling

Event handling is the mechanism that detects events and allows your application to take action in response to them. Events include mouse clicks, key strokes, window activation and update, Apple events, and so on. The event handling mechanism is defined by functions that are called to detect these events, package the events into objects, and dispatch them so that the code you write can respond.

This chapter describes the general event handling mechanism and how your application can respond to events. Events related to menus, controls, and Apple events are described in detail in separate chapters. For specific information about menu events, see “Menus” on page 283. For specific information about menu events, see “Control Events” on page 423. For specific information about Apple events, see “Scripting Support” on page 601.

In addition to describing the general event handling mechanism, this chapter also shows you how to:

- determine the state of an event
- respond to an event
- set up a behavior that responds to an event

About Event Handling

Conceptually, the Framework handles events by waiting for the next one to occur and then dispatching the event to an object that can respond by taking appropriate action. These objects are called event handlers because they can respond to an event as it is dispatched.

Event handlers are objects that represent things within the application such as documents, windows, and views. Thus, a document, window, or view can respond to an event such as a mouse click, a window activate event, and so on. These event handlers are linked together according to the priority each has in responding to an event. This organization is called the target chain.

The Framework also provides behaviors, which are objects that can be attached to event handlers to provide a response or "behavior" when an event occurs. Behaviors are like event handlers except that their only purpose is to represent actions in response to events. Because behaviors only represent actions instead of more tangible things, such as documents, windows, and views, behaviors can be defined once and then be used by several event handlers.

The following sections identify and describe important concepts related to event handling:

- kinds of events
- event handlers
- the target chain
- responding to events
- behaviors and event handling

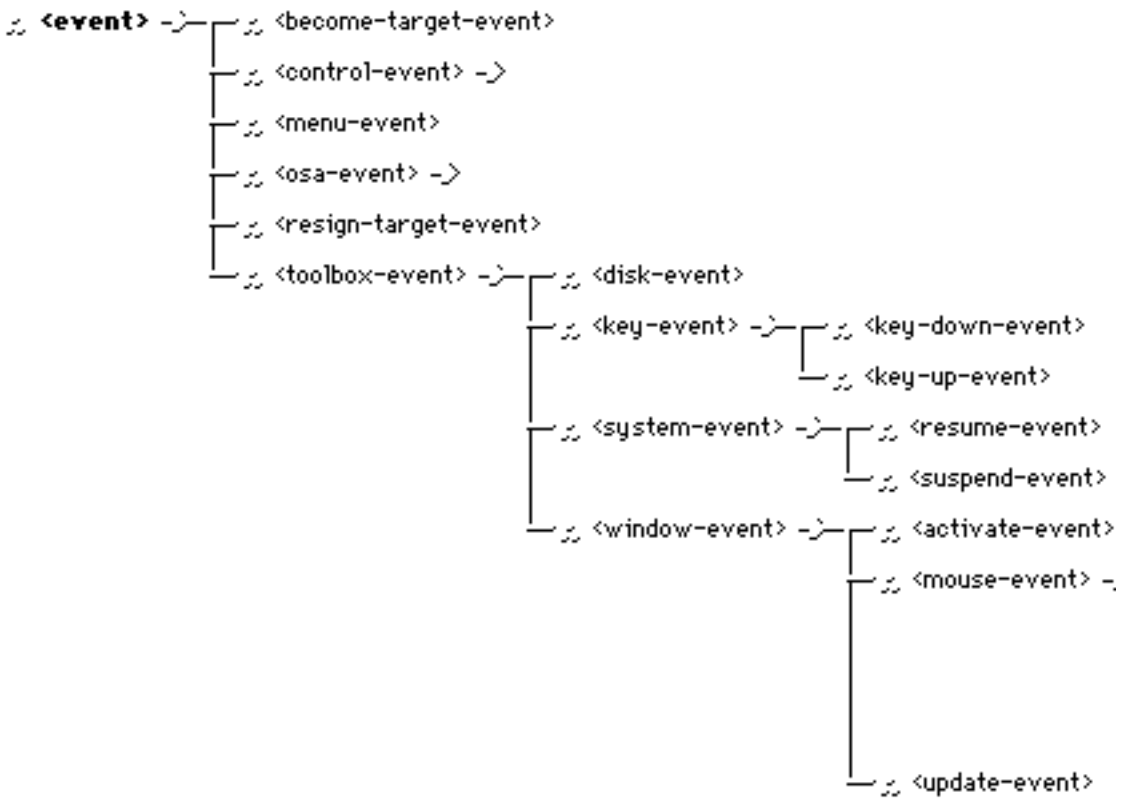
Kinds of Events

An **event** is a notification by the Event Manager about a change in the state of the Macintosh hardware or operating system. For example, the Event Manager notifies the application when the user presses or releases the mouse button or a key on the keyboard, when a window is activated or deactivated, when part of the screen must be redrawn, when an Apple event is sent to the application, and so on. For a complete discussion of the Event Manager and event

notification, see the Event Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

The Framework detects these events for you, packages the events into objects, and dispatches them so that they may be responded to in the appropriate way. In all cases, the detection and packaging of events is completely handled by the Framework. The kinds of objects that the Framework uses to represent events are shown in Figure 11-1 on page 243 and Figure 11-2 on page 245.

Figure 11-1 Event class hierarchy



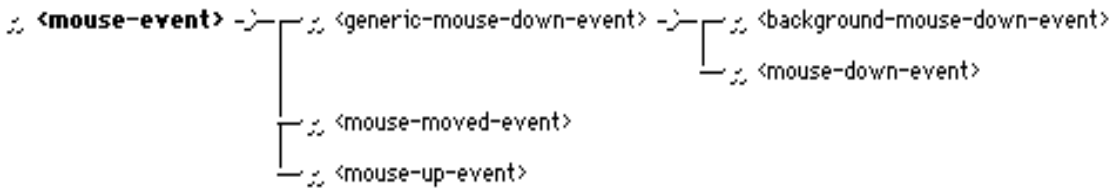
Objects created from the classes in Figure 11-1 represent the following kinds of events:

`<become-target-event>`	Represents a change in the target and specifies the event handler to become the new target. This class is used in conjunction with the `<resign-target-event>` class. The `<become-target-event>` class is discussed on page 274.
`<control-event>`	Represents an event related to a control, such as the pushing a button or clicking a radio button or scroller. For information about the `<control-event>` class, see page 454.
`<menu-event>`	Represents an event in a menu, such as selecting a menu item with the mouse or by a keyboard equivalent. For more information about the `<menu-event>` class, see "The <menu-event> Class" on page 301.
`<scripting-event>`	Represents an event related to the object support model and Open Scripting Architecture (OSA). For more information about the `<scripting-event>` class, see "Scripting Event Classes" on page 612.
`<resign-target-event>`	Represents a change in the target and specifies the event handler to become the new target. This class is used in conjunction with the `<become-target-event>` class. The `<resign-target-event>` class is discussed on page 275.
`<toolbox-event>`	Represents an event defined by the Macintosh toolbox `EventRecord` data structure. The `<toolbox-event>` class is described on page 268.
`<disk-event>`	Represents a disk event.
`<key-event>`	Represents a key event, such as a key-up or key-down event. For more information about this class, see "Key Event Classes" on page 270.
`<key-down-event>`	Represents a key-down event. For more information about this class, see "Key Event Classes" on page 270.
`<key-up-event>`	Represents a key-up event. For more information about this class, see "Key Event Classes" on page 270.

`<system-event>`	Represents a system event, such as a suspend or resume event. For more information about this class, see “System Event Classes” on page 271.
`<resume-event>`	Represents a resume event. For more information about this class, see “System Event Classes” on page 271.
`<suspend-event>`	Represents a suspend event. For more information about this class, see “System Event Classes” on page 271.
`<window-event>`	Represents a window event, such as an activate, update, or mouse event. For more information about this class, see “Window Event Classes” on page 271.
`<activate-event>`	Represents an activate event. For more information about this class, see “Window Event Classes” on page 271.
`<mouse-event>`	Represents a mouse event. For the class hierarchy of `<mouse-event>` subclasses, see Figure 11-2. For information about mouse event classes, see “Mouse Event Classes” on page 273.
`<update-event>`	Represents an update event. For more information about this class, see “Window Event Classes” on page 271.

Figure 11-2 shows the class hierarchy for mouse events.

Figure 11-2 Mouse event class hierarchy



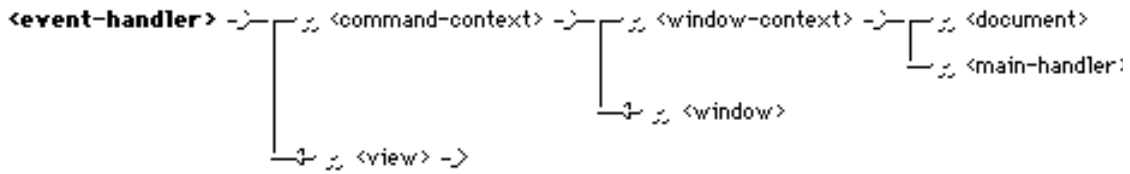
Objects created from the classes in Figure 11-2 represent the following kinds of events:

<mouse-event>	An event caused by a change to the state of the mouse.
<generic-mouse-down-event>	A mouse-down event.
<background-mouse-down-event>	A mouse-down event in an inactive window.
<mouse-down-event>	A mouse-down event in an active window.
<mouse-moved-event>	A mouse-moved event.
<mouse-up-event>	A mouse-up event.

Event Handlers

Objects that can respond to events are called **event handlers**. Objects that represent documents, windows, and views are event handlers because they can respond to, and thus handle, events. Figure 11-3 shows some of the event handler classes provided by the Framework.

Figure 11-3 Event handler class hierarchy



Note

As a subclass of <event-handler>, classes such as <view>, <window>, and <document>, inherit the ability to handle events. The primary purpose of these subclasses is to represent corresponding components of an application; thus, each subclass is described in the chapter for the relevant component. ◆

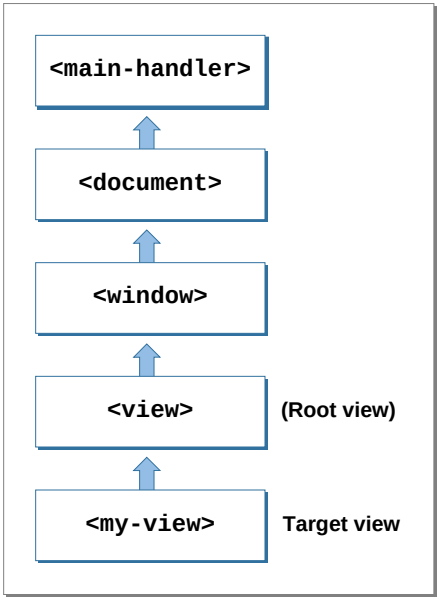
Objects created from the classes in Figure 11-3 represent the following kinds of events:

<event-handler>	Represents the ability to handle events. The <event-handler> class is described on page 263.
<command-context>	Represents the ability to handle undo and redo commands; thus, the class usually handles events associated with these commands. For more information about the <command-context> class, see "The <command-context> Class" on page 539.
<window-context>	Represents information about the state of a window, such as whether it is the main window or whether it should be closed when its document is closed. It typically does not handle events; however its subclasses, <document> and <main-handler>, do handle events. For more information about the <window-context> class, see page 406.
<document>	Represents the container (but not the storage) for the content of a window. Objects of this class typically handle menu events related to documents and files, such as those corresponding to the Save and Save As menu items. For information about menus and menu events, see "About Menus" on page 283. For information about documents, see "About Documents" on page 479.
<main-handler>	Represents the event handler of "last resort" and conceptually represents the application. If no other handler processes an event, such as the event associated with the Quit menu item, the main handler can handle it. The <main-handler> class is described on page 265.
<window>	Represents a window on the screen. The <window> class typically handles window events, including mouse events, which are described beginning with page 271. For more information about windows, see page 407.
<view>	Represents the background on which the content of a window is placed. The <view> class and its subclasses typically handle mouse events in the content area of a window. For more information about views, see "Views" on page 335.

The Target Chain

Because any event handler can respond to events, event handlers are organized in a chain so that each of them has the opportunity to handle the event in a prioritized and consistent order. This chain is called the **target chain** because a target event handler specifies the event handler with the highest priority, and thus, the first opportunity to handle the event. A simple, and typical, target chain consists of a `<main-handler>` object, a `<document>` object, a `<window>` object, and two or more `<view>` objects. These view objects represent the root view for the window and content view. Figure 11-4 shows how these event handlers are organized.

Figure 11-4 Target chain for a document with one window and content view



When the application starts, the Framework creates a `<main-handler>` object, which represents the application itself. Among other things, this object can handle events that no other event handler in the target chain takes

responsibility for. For example, a mouse click outside of a window is handled by the <main-handler> object.

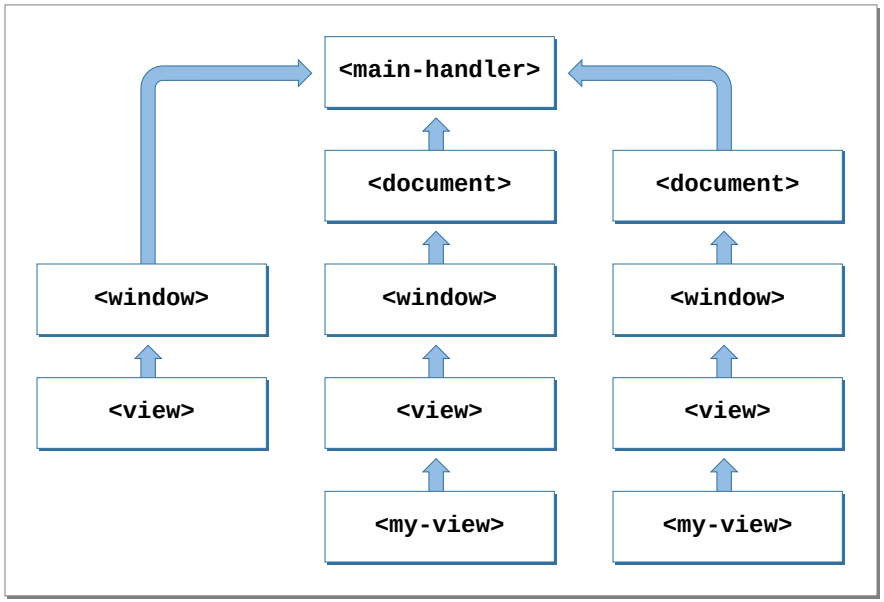
When you create event handlers, you specify the next handler in the target chain. By default, the next handler for a document is the <main-handler> object. When you create a window, you specify the <document> object as its next handler if the window displays the contents of the document; otherwise the Framework sets the next handler for a window to be the <main-handler> object also.

For each window, the Framework always provides a root view whose next handler is the window. The next handler for a view that you create to display the content of the document is typically the root view that the Framework provides.

The **target** specifies the first event handler that can handle an event. In Figure 11-4, for example, your view is allowed to handle an event before the root view, the root view is allowed to handle the event before the window, and so on. You specify the target view when you create a window. For more information about setting the target view, see "Setting the Target View" on page 347.

Note that the target chain changes when the target event handler changes. When a window is activated, for example, the target view for the window becomes the target event handler. Figure 11-5 shows the event handler objects in an application that has two open documents and a window that is not associated with any document. The active window determines the target chain.

Figure 11-5 An application with several event-handling chains



You may, of course, provide additional views and define their interrelationships by specifying the next handler. The interrelationships define a view hierarchy, which is described in the chapter "Views" on page 335.

Responding to Events

The response to an event may be handled completely or partially by the Framework or the response may be left for the application to handle. In many cases, the Framework responds to and completely handles the event for you. For example, if the user clicks on a window that is not active, the Framework detects the event, creates an object that represents it, and then responds to the event by bringing the window to the front and making it active.

In other cases, the Framework may respond partially. For example, when the user clicks the mouse, the Framework determines whether this is another click in a sequence of double or triple clicks or whether it is the first click in a new

sequence. Your application can test the condition (single, double, or triple click) and respond further.

Some events cannot be completely or partially handled because the events drive specific functions in the application. Often events that occur in the content area of a window, such as within a view, or within the menu bar are often left for the application to handle because the response defines the characteristics of the application. For example, a mouse-down and mouse-moved events in a "paint" application may cause the application to respond by drawing a line. The same events in a word-processing application may cause the application to respond by highlighting text.

When an event occurs, the Framework packages the event into the appropriate kind of event object and dispatches it for handling by an event handler—usually the target event handler. Although the Framework may dispatch some events so that they may be handled by other event handlers, the target event handler is the first event handler that is allowed to respond to an event for any event that your application can respond to.

The Framework handles most kinds of events by sending the event to the target event handler and allowing event handlers in the target chain handle event. The Framework, however, handles mouse events slightly differently. The following sections describe how the target chain is used in event handling and the differences for mouse event handling.

Event Handling Using the Target Chain

When an event is dispatched to the target event handler, the event may be handled, or the event may be passed to the next handler up the chain. For example, if the target is the content view, it may handle a resume event. The content view, however, might not handle menu events associated with the document, thus the Framework would allow the next handler in the chain to handle it. When the event handler that represents the document is called, it can respond.

The main handler object is responsible for handling an event if no other event handler takes responsibility. If the main handler does not handle the event either, the Framework calls `event-not-handled`, which you can use specify actions such as reporting an error.

You need not be concerned with the mechanism by which the Framework passes the event to the next handler. You are concerned only with responding to events. To respond to an event, you define a `do-event` method in which you

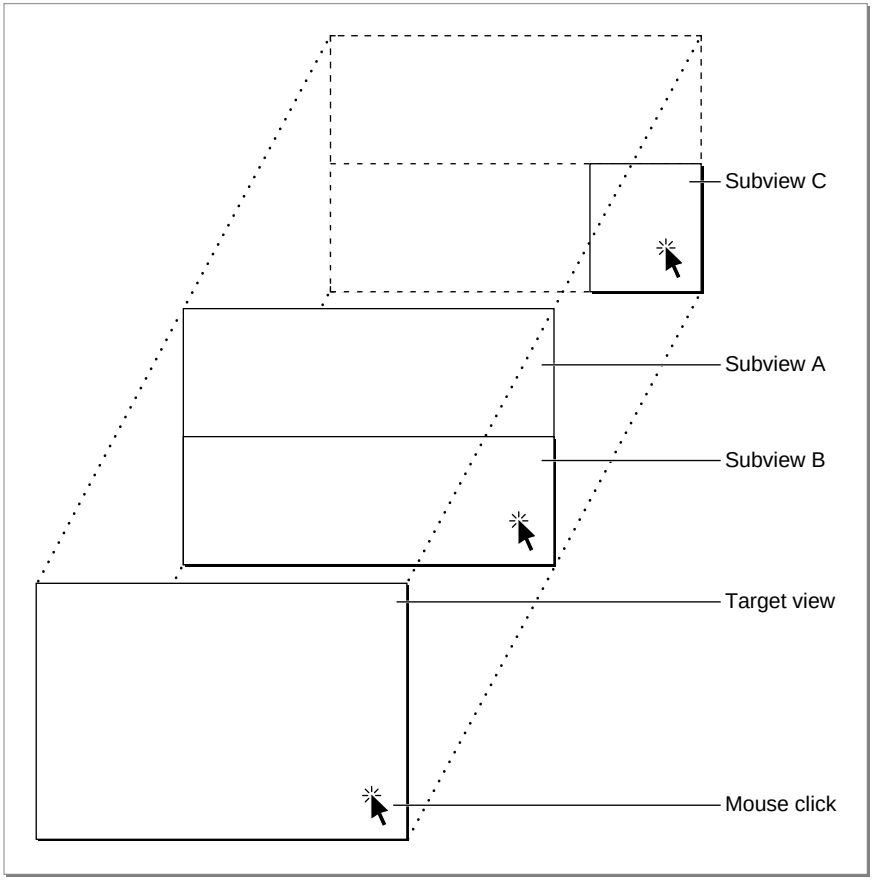
specialize the parameters for the event handler, the event itself, and another object, which may be a singleton that uniquely identifies the object.

For example, to allow a document to respond to a particular menu item, you can define a `do-event` method specialized on your document, event objects from the `<menu-event>` class, and a singleton parameter that specifies the menu item. The Framework will call the `do-event` method when it reaches the document in the target chain. Of course, the event and singleton must also match. For an example of setting up a `do-event` method, see "Defining a do-event Method" on page 260. For information and examples about setting up `do-event` methods for menu events, see "Responding to Menu Events" on page 290.

Mouse Event Handling

Like other events, mouse events are dispatched to the target handler. Unlike other events, however, the first event handler that can respond is the deepest subview in the view hierarchy underneath the location where the mouse event occurred. Consider the view hierarchy in Figure 11-6.

Figure 11-6 A mouse down event in a view hierarchy



The target handler is the content view and the mouse down event could be handled by the content view or its subviews B or C. Subview A does not have the chance to handle the event because the event did not occur within its bounds. The Framework starts with the deepest subview, which is subview C in Figure 11-6. If subview C does not handle the event, the Framework allows the next higher view to handle it and so on.

If the mouse event is not handled by a subview or the target handler, the event is passed up the target chain, as with other events. Unlike other events,

however, mouse events are not passed up the target chain further than the window. If a mouse event is not handled by any of the views in the view hierarchy and is not handled by the window, the event is passed directly to the main handler; event handlers between the window and the main handler, such as a document, are skipped.

Changing the Target

The target event handler changes due to events, such as mouse clicks, or due to executing a call to `set-target`. For example, the user may click on an inactive window to activate it. The Framework automatically deactivates the old target and activates the new one. The Framework executes the following actions to change the target:

1. It creates and dispatches a `<resign-target-event>` event up the target chain starting with the old target.
2. It sets the new target to the specified target, which changes the target chain. In the window activation example, it sets new target to be the `target-view` slot of the `<window>` object that represents the window being activated.
3. It creates and dispatches a `<become-target-event>` event up the target chain starting with the new target.

You can specialize the `event` parameter to `do-event` methods on `<resign-target-event>` and `<become-target-event>` to take additional actions when the target changes. For an example, see "Specifying Actions When the Target Changes" on page 263.

Behaviors and Event Handling

A **behavior** is a relatively self-contained task that you want an event handler, such as a view, to support. Behavior classes define tasks that you can add to and remove from an event handler. You can add the behavior to the list of actions performed by the event-handler and remove it when the behavior is no longer required.

There are two key advantages to using behaviors:

- You do not have to create a subclass of an event handler. For example, if you want to provide a special menu for certain kinds of windows, you do not

need to create a subclass of the window to distinguish it from other windows that do not have the menu. Thus, you do not need subclasses of windows just to add a menu.

- You can reuse the behavior with many event handlers. For example, you might want to provide a behavior to select the contents of a view. Once the selection behavior is defined, it can be added to any view that has selectable contents.

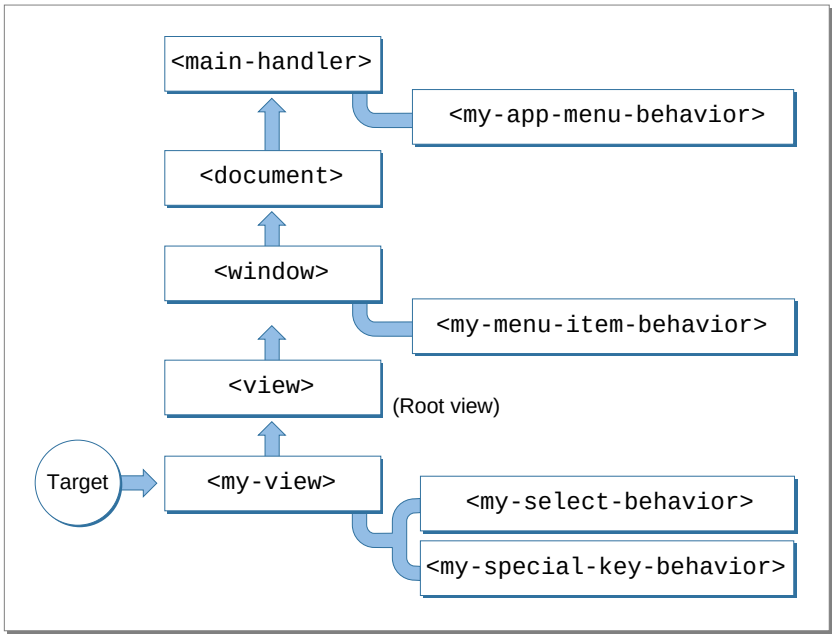
The Framework provides the `<behavior>` class from which you may define subclasses that implement your behaviors. These subclasses define the tasks you want to perform when an event occurs.

To allow a behavior to execute, you must add it to the list of behaviors for an event handler. When the event handler becomes the next handler in the chain eligible to handle an event, the event handler's behaviors are allowed to handle the event first. If a behavior does not handle the event, the event handler is then allowed to handle it.

For example, you may add a selection behavior to a view that supports selectable contents. When an event occurs and the view is in the target chain, the view's behavior is allowed to handle the event just before the view itself. If an event handler has several behaviors, they are allowed to handle an event in the order in which they are present in the list.

Consider the example target chain in Figure 11-7, which shows behaviors associated with an object from the `<main-handler>` class, an object from the `<window>` class, and one from the `<my-view>` subclass of <view>.

Figure 11-7 Target chain with behaviors



The <my-app-menu-behavior> object is associated with the <main-handler> object and allows the application to handle menu items not handled by the other event handlers; for example, the New menu item. If an event reaches the <main-handler> object, the <my-app-menu-behavior> behavior is allowed to handle it first. By adding a behavior to the <main-handler> object, you can provide application-wide support as the <main-handler> class cannot be subclassed.

The <my-menu-item-behavior> object is associated with a <window> object and responds to a menu item unique to a specific window. If ann event reaches the <window> object, the behavior has a chance to handle it first. The behavior only handles the specific menu item; however, adding the behavior to the window object eliminates the need to subclass the <window> object to support the special menu item.

The <my-select-behavior> and <my-special-key-behavior> objects are associated with the <my-view> object. The selection and special key actions are

implemented as behaviors, and not implemented directly by the subclass, because they can be associated with other <view> subclass objects that need to support those kinds of actions.

Because the behaviors are associated with the same object, either behavior may respond to an event. If both behaviors respond to the same event, the first behavior in the <my-view> object's list of behavior would have the chance to respond first, followed by the second behavior in the list. After the behaviors have the chance to respond, the <my-event> object would be given the chance to respond. If the <my-event> object does not respond, the event is passed up the chain.

The Framework also provides subclasses of the <behavior> class, as shown in Figure 11-8.

Figure 11-8 Behavior class hierarchy

Objects created from the classes in Figure 11-8 represent the following behaviors:

<behavior>	Represents the ability to usurp and handle events associated with the event handler to which the behavior is attached. The <behavior> class is described in this section; see page 266.
<clipboard-behavior>	Represents the behavior associated with the clipboard, including the ability to handle the following Edit menu items: Cut, Copy, Paste, and Clear. For more information about the

	<clipboard-behavior> class, see "The <clipboard-behavior> Class" on page 594.
<debugger-behavior>	Represents the behavior of the debugger. For more information about the <debugger-behavior> class, see "The <debugger-behavior> Class" on page 645
<dialog-behavior>	Represents a behavior that implements modal dialogs. For more information about the <dialog-behavior> class, see page 433.
<edit-text-dialog-filter>	Represents a behavior that handles conflicting keys, such as Tab and Return, while entering text in a dialog. For more information about the <edit-text-dialog-filter> class, see page 436.
<grid-select-behavior>	Represents a behavior that selects cells in a grid for editing. For more information about the <grid-select-behavior> class, see page 396.
<grid-arrow-key-behavior>	Represents a behavior that selects cells in a grid by using the arrow keys. For more information about the <grid-arrow-key-behavior> class, see page 396.
<tabber>	Represents a behavior that provides tabbing between fields in a dialog box. For more information about the <tabber> class, see page 435.

Using Event Handling Objects

The following sections show some of the common ways to use event handlers.

Adding an Event Handler to the Target Chain

When you create an event handler object, you can use the `next-handler:` keyword to specify the next event handler up the target chain from the object you are creating. By default, the next handler for documents and windows is the main handler, `*main-handler*`, the next handler for a root view is its window, and the next handler for a non-root view is its superview.

The Framework automatically sets up the next handler for a view whenever it is added as a subview or removed. If you create a window for a document, you need to specify the document as being the window's next handler. Listing 11-1 shows the next handler being specified for a window. The keyword parameters passed to `make-scrolling-window` are used to create a `<window>` object with the `make` function.

Listing 11-1 Specifying the next handler for a window

```
let window = make-scrolling-window(list(view),
                        location: point(100, 100),
                        extent: view.extent + $scroll-bar-size,
                        target-view: view,
                        title: document.title,
                        next-handler: document,
                        main-window: #t,
                        has-go-away: #t,
                        has-zoom: #t,
                        has-grow: #t);
```

Defining a do-event Method

The `do-event` method specifies the actions to take when an event occurs and the event handler is allowed to handle it. Listing 11-2 defines a `do-event` method that executes when a mouse-down event is sent up the target chain and a `<my-view>` object has the chance to respond.

Listing 11-2 Defining a `do-event` method

```
define method do-event (view :: <my-view>,
                                event :: <mouse-down-event>,
                                id :: <object>) => ()
    ignore(event, id);
    let final-point = track-mouse(make(<my-tracker>),
                                  view, event.local-mouse);
    let draw-rect = make(<rect>);
    set-rect-from-points(draw-rect, event.local-mouse, final-point);
    view.the-document.data
        := insert(view.the-document.data, draw-rect, last: #t);
    invalidate(view, draw-rect);
end method;
```

If an event has been handled completely, it is not necessary to pass the event up the target chain. If you want other event handlers or their behaviors to also handle the event, call `next-method` from your `do-event` method.

Note
For examples of `do-event` methods that respond to menu events. See "Responding to Menu Events" on page 290. ◆

Implementing a Behavior

Behaviors are useful for specifying additional actions for an event handler. For example, you typically define a behavior and add it to the main handler so that menu items, such as New, can be handled by the main handler without requiring you to define a subclass of `<main-handler>` and specialize a do-event

method on the subclass. For an example of how a behavior can handle a menu event, see "Responding to Menu Events" on page 290.

To implement a behavior, perform the following steps:

1. Define a subclass of the `<behavior>` class.
2. Define methods whose parameters are specialized on your behavior subclass. For a list of methods, see Table 11-3, "Event-handling functions for dispatching" on page 279.
3. Add the behavior to the appropriate event handler.

Listing 11-3 shows the class definition for a `<tool>` class that changes the cursor and handles mouse tracking within a view.

Listing 11-3 Defining a behavior subclass

```
define class <tool> (<behavior>)
    slot tracker-class :: <class>;
end class;
```

Listing 11-4 shows the `behavior-event` method that specifies the actions to take when the `<paint-view>` event handler is allowed to respond to a mouse-down event. The behavior is allowed to respond before the event handler's `do-event` method is called.

Listing 11-4 Defining a behavior's response to an event

```
define method behavior-event (tool :: <tool>,
                              next :: <list>,
                              view :: <paint-view>,
                              event :: <mouse-down-event>,
                              id :: <keyword>) => ()
    ignore(tool, next, id);
    track-mouse(make(tool.tracker-class, tool: tool),
        view, event.local-mouse);
end method;
```

Listing 11-5 shows the `behavior-set-cursor` method that specifies the actions to take when the `<paint-view>` event handler is allowed to respond to a mouse-moved event. The behavior is allowed to respond before the event handler's `do-set-cursor` method is called.

Listing 11-5 Handling the cursor from a behavior

```
define method behavior-set-cursor (tool :: <tool>,
                                   next :: <list>,
                                   view :: <view>,
                                   mouse :: <point>,
                                   cursor-region :: <qd-region>) => ()
    ignore(next, view, mouse, cursor-region);

    set-cursor($arrow-cursor);
end method;
```

You can add a behavior to the list of behaviors for an event handler at any time and remove a behavior when it is no longer needed. Listing 11-6 shows how to add and remove a behavior.

Listing 11-6 Adding and removing a behavior

```
let tracking-tool = make(<tool>);
add-behavior(my-paint-view,tracking-tool);
...
remove-behavior(my-paint-view,tracking-tool);
```

Behaviors receive a chance to handle and event in the order that they exist in the event handler's list of behaviors. You can specify the `first:`, `before:`, or `after:` keyword parameters in the call to `add-behavior` to respectively place the behavior first in the list, or before or after another behavior. If you do not specify a keyword, the behavior is placed last in the list.

Specifying Actions When the Target Changes

When the user clicks on a window, it is activated and the window's target view becomes the target of the chain of event handlers. At other times, you can call `set-target` to change the target. Whenever a new target is selected, the Framework first dispatches a `<resign-target-event>` followed by a `<become-target-event>`. You can define `do-event` or `behavior-event` methods whose event parameter specializes on these events to handle clean up for the old target or set up of the new one. For example, the Framework uses these events to change highlighting for views.

Event Handling Objects Reference

The following sections describe the classes that implement the Framework's event handling mechanism and identify the functions that you can use with them.

The <event-handler> Class

The `<event-handler>` class provides subclasses with the ability to handle events. It is seldom necessary to create a direct subclass of the `<event-handler>` class. You never need to create `<event-handler>` objects, rather you create objects of its subclasses, such as `<document>`, `<window>`, and `<view>`.

Listing 11-7 shows the class definition for the `<event-handler>` class.

Listing 11-7 The `<event-handler>` class definition

```
define free open class <event-handler> (<object>)
  slot identifier :: <symbol>,
    init-value: $null-identifier,
    init-keyword: identifier:;

  slot behavior-list :: <list>,
    init-value: #(),
    init-keyword: behavior-list:;
```

```
  slot enabled? :: <boolean>,
    init-value: #t,
    init-keyword: enabled:;

  slot next-handler,
     init-value: #f;
end class;
```

Slot descriptions

`identifier`	Identifies the event handler. Set this slot if you need to specialize a method on a singleton event handler. Use the `identifier:` keyword to specify the identifier; otherwise, the slot is set to `$null-identifier`.
`behavior-list`	Specifies the behaviors associated with the event handler. Use the `behavior-list:` keyword to specify the initial list of behaviors; otherwise, the list is empty. You can call `add-behavior` and `remove-behavior` to change the contents of this list. For more information about behaviors, see "Behaviors and Event Handling" on page 254.
`enabled?`	Specifies whether the event handler or its behaviors is prepared to respond to events. By default, event handlers, and thus their behaviors can respond to events. Call `enabled?` to determine if the event handler or its behaviors is prepared to respond to events.
`next-handler`	Specifies the next event handler in the target chain. To access this slot, call `next-handler` and `next-handler-setter`.

Initialization

The `initialize` method for `<event-handler>` objects sets the sets the next handler. Its parameters are defined as follows:

```
define method initialize (event-handler :: <event-handler>,
                          #key next-handler: next-evt-handler) => ()
```

`event-handler`	The event handler.
`next-evt-handler`	The next event handler.

The <main-handler> Class

The <main-handler> class handles the event loop and is the event handler that responds to an event if the event is not handled by another event handler. The Framework creates a single <main-handler> object for you. You never need to create a subclass of <main-handler> or create a <main-handler> object yourself. Listing 11-8 shows the class definition for the <main-handler> class.

Listing 11-8 The <main-handler> class definition

```
define primary open class <main-handler> (<window-context>)
    slot command-queue :: <deque>,
        init-function: curry(make, <deque>);
    slot document-type-list :: <list>,
        init-value: #();  // list of pairs (file type, document class)
    inherited slot close-with-last-window?>,
        init-value: #f;
    // private
    // use target-handler and target-handler-setter to access this slot.
    slot %target-handler :: <event-handler>;
end class
```

Slot descriptions

`command-queue` Internal queue of commands which are waiting to be executed. Not exported.

`document-type-list` A list of pairs of file types and document classes supported by the application, such as `'PICT'`, and the document class that supports the type. Initially the list is empty. Call `add-document-type` to add to the list. To determine which document class supports a document type call `get-document-class`. The `document-type-list` slot is not exported.

`close-with-last-window?`
Specifies whether the children of a context should be closed when the last window associated with the context is closed `(#t)` or whether they should remain open after the

	last window associated with the context is closed (#f). Use the close-with-last-window: keyword and specify true (#t) to allow children of the main handler to be closed; otherwise, the value is false(#f), meaning that children of the context will not be closed. You can use the getter and setter to access this slot. By default, windows close their children and the main handler does not. For more information about window contexts, see "The <window-context> Class" on page 406.
%target-handler	The current target event handler. You can call set-target to set the slot, You can also call the getter and setter functions, target-handler and target-handler-setter. The %target-handler slot is not exported.

Initialization

The initialize method for the <main-handler> object sets the target event handler to itself. Its parameters are defined as follows:

```
define method initialize (main-handler :: <main-handler>, #key)
```

main-handler	The main handler.

The <behavior> Class

The <behavior> class defines the protocol for behaviors that you wish to implement. You create a subclass of <behavior> to define a task that you want an event handler to perform. You create an object of your subclass and add call add-behavior to add it to the list of behaviors for the event handler you specify. For examples of adding and removing behaviors, see Listing 11-6 on page 262.

Listing 11-9 shows the class definition for the <behavior> class.

Listing 11-9 The <behavior> class definition

```
define primary open class <behavior> (<object>)
    slot identifier,
        init-value: $null-identifier,
        init-keyword: identifier:;
end class;
```

Slot descriptions

identifier	Identifies the behavior. Set this slot if you need to specialize a method on a singleton behavior or if you wish to place behaviors in an event handler's list of behaviors based on the position of this behavior in the list.

Initialization

No initialize method is defined for <behavior> objects.

The <event> Class

The <event> class defines the protocol for events. Except for scripting events, the Framework provides event-related objects for you. You do not need to define subclasses of <event> or create <event> objects yourself. Listing 11-10 shows the class definition for the <event> class.

Listing 11-10 The <event> class definition

```
define primary open class <event> (<object>)
  slot identifier :: <symbol>,
    init-value: $null-identifier,
    init-keyword: identifier:;
end class;
```

Slot descriptions

identifier	Identifies the event. Set this slot if you need to specialize a method on a singleton event. Use the identifier: keyword to specify the identifier; otherwise, the slot is set to $null-identifier.

Initialization

No `initialize` method is defined for `<event>` objects.

Toolbox Event Classes

Toolbox event classes define objects that represent Event Manager events. The slots of the `<toolbox-event>` class hold information from an Event Manager event record. The Framework provides subclasses of the `<toolbox-event>` class whose objects represent Event Manager events. You do not need to create subclasses of the `<toolbox-event>` class or create `<toolbox-event>` objects yourself.

Listing 11-11 shows the class definition for the `<toolbox-event>` class.

Listing 11-11 The `<toolbox-event>` class definition

```
define primary sealed class <toolbox-event> (<event>)
  slot event-what :: <integer>;

  slot event-time :: <integer>;

  slot global-mouse :: <point>,
     init-keyword: global-mouse:;

  slot event-message :: <integer>;

  slot modifiers :: <integer>;

  slot shift-key? :: <boolean>;

  slot option-key? :: <boolean>;

  slot command-key? :: <boolean>;

  slot control-key? :: <boolean>;
end class
```

Slot descriptions

`event-what`	The kind of event. It corresponds to the `what` field in the toolbox `EventRecord` data type.
`event-time`	The elapsed ticks since system start up. It corresponds to the `when` field in the toolbox `EventRecord` data type.
`global-mouse`	The location of the cursor, in global coordinates, when the event occurred. It corresponds to the `where` field in the toolbox `EventRecord` data type.
`message`	Additional information associated with the event. It corresponds to the `message` field in the toolbox `EventRecord` data type.
`modifiers`	The state of the modifier keys and mouse button when the event occurred. It corresponds to the `modifier` field in the toolbox `EventRecord` data type. The state of commonly tested keys are specified in the following slots:
`shift-key?`	The state of the shift key. True (`#t`) indicates that it was down when the event occurred.
`option-key?`	The state of the option key. True (`#t`) indicates that it was down when the event occurred.
`command-key?`	The state of the command key. True (`#t`) indicates that it was down when the event occurred.
`control-key?`	The state of the control key. True (`#t`) indicates that it was down when the event occurred.

Initialization

The `initialize` method for `<toolbox-event>` objects replaces the default slot values with those in the specified event record. Its parameters are defined as follows:

```
define method initialize (event :: <toolbox-event>,
                          #key event-record: event-rec)
```

`event`	The toolbox event.
`event-rec`	An event record.

If the `event-record:` keyword is not specified, no values are replaced.

The <disk-event> Class

The <disk-event> class defines objects that represent Event Manager disk events. The Framework creates <disk-event> objects as required. You do not need to define subclasses of <disk-event> or create <disk-event> objects yourself. Listing 11-12 shows the class definition for the <disk-event> class.

Listing 11-12 The <disk-event> class definition

```
define primary sealed class <disk-event> (<toolbox-event>)
end class;
```

Key Event Classes

The <key-event> class defines objects that represent Event Manager key events, which are key-down and key-up events. The Framework provides subclasses for each of these events, <key-down-event> and <key-up-event>, respectively, from which the Framework creates objects as required. You do not need to define subclasses from the <key-event> class or its subclasses, nor do you create objects from these classes yourself. Listing 11-13 shows the class definitions for the <key-event>, <key-down-event>, and <key-up-event> classes.

Listing 11-13 Class definitions for key events

```
define primary sealed class <key-event> (<toolbox-event>)
  slot key-character :: <character>,
    init-keyword: key-character:;
end class;

define primary sealed class <key-down-event> (<key-event>)
end class;

define primary sealed class <key-up-event> (<key-event>)
end class;
```

Slot descriptions

`key-character` Specifies which key is affected by a key-up or key-down event.

System Event Classes

The <system-event> class defines objects that represent Event Manager system events, which are suspend and resume events. The Framework provides subclasses for each of these events, <suspend-event> and <resume-event>, respectively, from which the Framework creates objects as required. You do not need to define subclasses from the <system-event> class or its subclasses, nor do you create objects from these classes yourself. Listing 11-14 shows the class definitions for the <system-event>, <suspend-event>, and <resume-event> classes.

Listing 11-14 Class definitions for system events

```
define primary sealed class <system-event> (<toolbox-event>)
  slot convert-clipboard? :: <boolean>,
     setter: #f,
     required-init-keyword: convert-clipboard:;
end class;

define primary sealed class <suspend-event> (<system-event>)
end class;

define primary sealed class <resume-event> (<system-event>)
end class;
```

Slot descriptions

`convert-clipboard?` Specifies whether the contents of the clipboard must be converted and copied to a private scrap when a resume event occurs and whether the clipboard must be filled from a private scrap when a suspend event occurs. It is true (`#t`) if the clipboard must be converted by copying to and filling from a private scrap; otherwise it is false (`#f`) if these actions are not necessary.

Window Event Classes

Window event classes define objects that represent Event Manager events that affect windows. The slots of the <window-event> class hold information from an Event Manager window record. The Framework provides subclasses of the <window-event> class whose objects represent Event Manager activate, update,

and mouse events. (For information about mouse event subclasses, see the next section, beginning on page 273.) You do not need to define subclasses from the `<window-event>` class or its subclasses, nor do you create objects from these classes yourself.

Listing 11-15 shows the class definition for the `<window-event>` class.

Listing 11-15 The `<window-event>` class definition

```
define primary sealed class <window-event> (<toolbox-event>)
  slot event-window-ptr :: <WindowRecord>,
    required-init-keyword: window-ptr:;
end class;
```

Slot descriptions

`event-window-ptr` Specifies the window associated with the event. The window is defined using the toolbox `WindowRecord` data type. For information about the `WindowRecord` data type, see the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

Listing 11-16 shows the class definition for the `<activate-event>` class.

Listing 11-16 The `<activate-event>` class definition

```
define primary sealed class <activate-event> (<window-event>)
  slot activate-it? :: <boolean>,
    required-init-keyword: activate:
end class;
```

Slot descriptions

`activate-it?` Specifies whether the window should be activated (`#t`) or deactivated (`#f`). The Framework sets this slot from the toolbox event record when an event occurs. You may call the slot's getter method to determine whether the window is being activated or deactivated.

Listing 11-17 shows the class definition for the `<update-event>` class.

Listing 11-17 The <update-event> class definition

```
define primary sealed class <update-event> (<window-event>)
end class;
```

Mouse Event Classes

Mouse event classes define objects that represent Event Manager events caused by the user pressing or releasing the mouse button. The Framework provides subclasses of the <mouse-event> class whose objects represent Event Manager mouse-down, mouse-up events, and mouse-moved events, which are <generic-mouse-down-event>, <mouse-up-event>, and <mouse-moved-event>, respectively.

The Framework also provides subclasses for the <generic-mouse-down-event> class so that you can define methods that execute only when the mouse is clicked in an active window or in the background. These subclasses are <mouse-down-event> and <background-mouse-down-event>, respectively.

You do not need to define subclasses from the <mouse-event> class or its subclasses, nor do you create objects from these classes yourself. Listing 11-18 shows the class definition for the <mouse-event> class and its subclasses, <generic-mouse-down-event>, <mouse-down-event>, <background-mouse-down-event>, <mouse-up-event>, and <mouse-moved-event>.

Listing 11-18 Class definitions for mouse events

```
define primary sealed class <mouse-event> (<window-event>)
  slot local-mouse :: <point>;

  slot part-code :: <integer>,
     setter: #f,
    required-init-keyword: part-code:;
end class;

define primary sealed class <generic-mouse-down-event> (<mouse-event>)
end class;
```

```
define primary sealed
    class <mouse-down-event> (<generic-mouse-down-event>)
end class;

define primary sealed
    class <background-mouse-down-event> (<generic-mouse-down-event>)
end class;

define primary sealed class <mouse-up-event> (<mouse-event>)
end class;

define primary sealed class <mouse-moved-event> (<mouse-event>)
end class;
```

Slot descriptions

`local-mouse`	Specifies the location of the mouse pointer in the local coordinates of a window or view.
`part-code`	Specifies the part of the screen or window the mouse pointer is over; for example, the menu bar or the content area of a window. For information about the part codes, see the codes returned by the toolbox `FindWindow` function in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

The <become-target-event> Class

The `<become-target-event>` class defines objects that the Framework creates when the target event handler changes. You do not need to define subclasses of `<become-target-event>` or create `<become-target-event>` objects yourself. Listing 11-19 shows the class definition for the `<become-target-event>` class.

Listing 11-19 The `<become-target-event>` class definition

```
define primary sealed class <become-target-event> (<event>)
  slot new-target :: <event-handler>,
     setter: #f,
     required-init-keyword: new-target:;

  slot same-window? :: <boolean>,
```

```
    setter: #f,
    required-init-keyword: same-window:;
end class;
```

Slot descriptions

`new-target`	Specifies the event handler to become the new target.
`same-window?`	Specifies whether the window associated with the current target in the event handling chain is the same as the new target. The Framework sets this slot when the target changes; for example, when the user clicks the mouse in a different view. You may call the slot's getter method to determine whether the window changed as a result of changing the target.

The <resign-target-event> Class

The `<resign-target-event>` class defines objects that the Framework creates when the target event handler changes. You do not need to define subclasses of `<resign-target-event>` or create `<resign-target-event>` objects yourself. Listing 11-20 shows the class definition for the `<resign-target-event>` class.

Listing 11-20 The `<resign-target-event>` class definition

```
define primary sealed class <resign-target-event> (<event>)
  slot old-target :: <event-handler>,
    setter: #f,
    required-init-keyword: new-target:;

  slot same-window? :: <boolean>,
    setter: #f,
    required-init-keyword: same-window:;
end class;
```

Slot descriptions

`old-target`	Specifies the event handler that is no longer the target.
`same-window?`	Specifies whether the window associated with the current target in the event handling chain is the same as the new target. The Framework sets this slot when the target

changes; for example, when the user clicks the mouse in a different view. You may call the slot's getter method to determine whether the window changed as a result of changing the target.

Functions for Event Handling

This section identifies methods and generic functions associated with event handling. Getter and setter functions are identified with their associated slots. For slot descriptions, see "Event Handling Objects Reference" on page 263.

Methods and generic functions related to event handling are divided into the following groups:

- general event handling functions
- those that are used for behaviors
- those that are used for streaming behaviors
- those that are used for dispatching

Table 11-1 identifies general event handling functions.

Table 11-1 General event-handling functions

Function	Purpose	Call	Sp.
`open`	Open an event handler, such as a document, window, or view.	√	
`close`	Close an event handler, such as a document, window, or view.	√	
`dispatch-event`	Internal.		
`do-setup-menus`	To enable menu items when the specified behavior is active.		
`handle-event`	Starts event handling at the specified event handler.	√	
`do-event`	Define the actions to take when an event occurs while the specified event handler is active.		√

Table 11-1 General event-handling functions

Function	Purpose	Call	Sp.
`handle-setup-menus`	To enable menu items when the specified behavior is active		
`handle-update`	Internal.		
`do-update`	Specifies actions to take when the a change occurs.		
`set-event-record`	Internal	√	
`event-not-handled`	Specifies actions to take when an event is not handled by any event handler or behavior.		√
`do-content-click`	Internal.		
`do-drag`	Internal.		√
`do-go-away`	Internal.		
`do-zoom`	Internal.		
`double-click?`	Determine if a mouse click is the second one in a sequence.	√	
`triple-click?`	Determine if a mouse click is the third one in a sequence.	√	

Table 11-2 shows event handling fuctions for behaviors.

Table 11-2 Event-handling functions for behaviors

Function	Purpose	Call	Sp.
`add-behavior`	Add the specified behavior to an event handler's list of behaviors	√	
`remove-behavior`	Remove the specified behavior from an event handler's list of behaviors	√	
`behavior-event`	Define the actions to take when an event occurs while the specified behavior is active.		√
`next-behavior`	Defers handling an event until other behaviors have the chance to respond	√	
`find-behavior`	Finds a behavior in the list of behaviors	√	
`behavior-update`	Specifies actions to take when a change occurs.		√
`behavior-setup-menus`	To enable menu items when the specified behavior is active.		√
`behavior-idle`	Specifies actions to take during idle time.		√
`behavior-open`	Specifies actions to take when the event handler is opened.		√
`behavior-close`	Specifies actions to take when the event handler is closed.		√
`behavior-set-cursor`	Specifies actions to take when the mouse moves into the behavior's view.		√

Table 11-3 shows event handling fuctions for dispatching.

Table 11-3 Event-handling functions for dispatching

Function	Purpose	Call	Sp.
`event-loop`	Internal.		
`poll-event`	Internal.		
`make-mouse-down -event`	Internal.		
`make-toolbox -event`	Internal.		
`create-menu -event`	Internal.		

Table 11-4 shows functions that are used for streaming.

Table 11-4 Event-handling functions for streaming

Function	Purpose	Call	Sp.
`clonable?`	To determine whether the behavior may be cloned	√	
`streamable?`	To determine whether the behavior may be streamed	√	
`streamable -behaviors`	Internal.	√	
`cloneable -behaviors`	Internal.	√	
`clone-behavior -list`	Internal.	√	

CHAPTER 11

Event Handling

CHAPTER 12

Menus

Contents

About Menus 283
- Menus and Menu Items 283
- Menu Event Handling 284
- Menu Classes 285

Using Menu-Related Objects 287
- Creating Menus and Menu Items 287
- Setting Up Menus 288
- Responding to Menu Events 290
- Setting up the About Menu Item 292

Menu Class Reference 292
- The <menu-element> Class 292
- The <menu-item> Class 294
- The <separator-item> Class 297
- The <menu> Class 297
- The <apple-menu> Class 300
- The <menu-event> Class 301
- Functions for Menus 302

The Framework sets up your application's menu bar and adds the Apple menu, the Help menu and the Application menu for you automatically. The Framework provides you with the ability to

- set up menus and menu items
- install, access, and remove your own menus
- add items to a menu
- enable and disable menu items
- handle menu events

This chapter describes the classes that the Framework provides for supporting menus and events related to menus and the concepts for their use. The chapter also shows you how to use objects from these classes to set up menus and respond to menu events.

The Framework's event handling system is used to set up menus and respond to menu events. You should be familiar with event handling, as described in the chapter "Introduction" on page 219. For general information about event handlers and behaviors, see the chapter "Event Handling" on page 241.

About Menus

This section describes the concepts and classes related to menus. For general information about Macintosh menus and the Menu Manager that the Framework uses to implement them, see the Menu Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

Menus and Menu Items

The Framework provides several classes that represent the various data structures required by the Menu Manager. These classes represent a menu bar, general-purpose menus and specific ones such as the Apple menu, and menu items.

You seldom need to be concerned with the menu bar or the Apple menu. Most of the work centers around customizing the various menus, such as the File, Edit, and other menus, and the elements that go into them.

Menu items are associated with event handlers. The Framework provides two methods specialized on event handler, `do-setup-menus` and `behavior-setup-menus`, in which you can enable and mark menu items. Thus, an event handler, such as a document or view, or its behavior can enable menu items when it is in the target chain. The `<main-handler>` object representing the application or its behaviors can control when application-wide menu items, such as New, Open, or Quit are available. For an example of how to set up menus, see "Setting Up Menus" on page 288.

Entire menus can also be installed and removed using the same event handling mechanism and methods (`do-setup-menus` and `behavior-setup-menus`) that allow menu items can be enabled.

Note
The Framework maintains state information about menus and menu items. The Framework only calls the Menu Manager when it is necessary to synchronize its state with the state of the Menu Manager, which reduces the number of system calls. For example, `do-setup-menus` may be called several times to change the state of menu items before an event occurs that actually makes it necessary to display a menu. The Framework brings the Menu Manager up-to-date just before the menu is displayed. ◆

Menu Event Handling

Menu items are associated with menu events. Menu events are also represented by a class in the Framework. When you create a menu item, the Framework creates an event that corresponds to the item unless you specify an explicit association between an existing event object and the item.

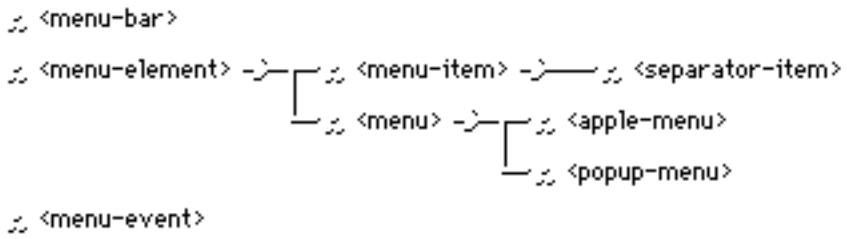
Menu events, not menu items, are handled by the Framework. For example, if the user selects New from the File menu, the Framework dispatches the event associated with the selection. You specialize a `do-event` or `behavior-event` method to handle the event.

You need not be concerned with the menu item object after you establish the correspondence between it and the event. For example, when you enable a menu item, you specify the menu event, not the menu item. Although this specification may seem somewhat indirect, it allows several menu items to be processed with the same event.

Menu Classes

Figure 12-1 shows the menu-related classes provided by the Framework.

Figure 12-1 Menu-related classes



`<menu-bar>`	Represents the application's menu bar. The menu bar is internal to the Framework.
`<menu-element>`	Represents information that is used both by menu items and by menus themselves. For more information about this class, see "The <menu-element> Class" on page 292.
`<menu-item>`	Represents an item in a menu, such as the New item in the File menu. For more information about this class, see "The <menu-item> Class" on page 294.
`<separator-item>`	Represents a divider, such as the one before the Quit menu item in the File menu. For more information about this class, see "The <separator-item> Class" on page 297.
`<menu>`	Represents a menu, such as the Apple, File, or Edit menus. It is also used to represent popup menus. For more information about this class, see "The <menu> Class" on page 297.
`<apple-menu>`	Represents the Apple menu, which is the menu in which you add the application's About item. For more information about this class, see "The <apple-menu> Class" on page 300.

`<popup-menu>`	Represents a pop-up menu. A popup menu is actually a control. For more information about this class and about controls, see the chapter "Dialogs and Controls" on page 419.
`<menu-event>`	Represents an event, such as a keyup event in the menu bar or a keyboard equivalent. For more information about this class, see "The <menu-event> Class" on page 301.

Using Menu-Related Objects

The following sections provide examples of how to set up and use menu-related objects. These sections show you how to

- create menus and menu items
- set up menus
- set up events to respond to menu items
- set up the About menu item in the Apple menu

Creating Menus and Menu Items

You typically create menus as part of the initialization sequence for your application. For example, if you define a method, `init-my-app`, to initialize your application, you may create menus in the method. Alternatively, if you create a behavior to represent your application, you can create menus in its `initialize` method.

Listing 12-1 shows how to create a File menu with the New, Close, and Quit items. The menu is installed, however, the items are not yet enabled.

Listing 12-1 Making a menu and menu items

```
make(<menu>,
    title: "File",
    install: #t,
    items: list(make(<menu-item>,
                    title: "New",
                    identifier: #"new",
                    command-key: 'N'),
                make(<separator-item>),
                make(<menu-item>,
                    title: "Close",
                    identifier: #"close",
                    command-key: 'W'),
```

```
                make(<separator-item>),
                make(<menu-item>,
                        title: "Quit",
                        identifier: #"quit",
                        command-key: 'Q')));
```

A title, event, and keyboard equivalent is defined for each menu item. The Framework creates a `<menu-item>` object whose identifier is specified by the `identifier:` keyword.

Listing 12-2 shows how to create menu items by specifying an actual event object using the `event:` keyword instead of the identifier.

Listing 12-2 Specifying an event object for a menu item

```
make(<menu>, title: "Size",
    install: #t,
    items: list(make(<menu-item>,
                title: "8",
                event: make(<tile-size-event>, tile-size: 8)),
            make(<menu-item>,
                title: "16",
                event: make(<tile-size-event>, tile-size: 16)),
...
```

The `event:` keyword allows you to specify a subclass of <menu-event>. The `make` function for the `<tile-size-event>` creates an object from a subclass to associate with the menu item.

Setting Up Menus

When the Framework dispatches a menu event, it also updates the menu bar. It clears its internal menu state and calls `do-setup-menus` for each event handler in the target chain. It also calls `behavior-setup-menus` for each behavior of an event handler before calling the event handler's `do-setup-menus`.

Note
The Framework does not actually call the Menu Manager routines to set up menus until it is necessary to display a menu. ◆

You provide specializations of `do-setup-menus` and `behavior-setup-menus` to reenable menu items and to mark reenabled items. Listing 12-3 shows the menu items that are enabled when a behavior is added to an event handler.

Listing 12-3 Setting up menus for a behavior

```
define method behavior-setup-menus
                                    (behavior :: <test-behavior>,
                                    next-behaviors :: <list>,
                                    main-handler :: <main-handler>)
    ignore(next-behaviors, main-handler);
    next-method();

    enable-item(#"new");
    enable-item(#"float-test");
    enable-item(#"dialog-test");
    enable-item(#"grid-test");

    if (*has-quickdraw-gx*)
        enable-item(#"gx-test");
    end if;

    enable-item(#"benchmark");
end method;
```

Listing 12-4 shows the menu items that are enabled and marked when an `<O-tile-view>` object is in the target chain.

Listing 12-4 Setting up menus for an event handler

```
define method do-setup-menus (view :: <O-tile-view>)
    next-method();
```

```
    enable-item(#"tile-type");
    mark-item(#"tile-type", tile-type: view.tile-type);
    ...
    enable-item(#"tile-show-dual");
    mark-item(#"tile-show-dual", mark: view.show-dual);
end method;
```

The menu items to be enabled or marked are identified by the menu event; not the menu item. If several menu items are associated with an event, all of them are enabled or marked.

The actual way that events are enabled or marked is determined by specializations of `do-enabling` and `do-marking` methods. These methods are called by `enable-item` and `mark-item`, respectively. Listing 12-5 shows a specialization of the `do-marking` method on `<tile-type-event>` objects. This specialization of `do-marking` only marks tiles if the type of the tile matches the type specified in the call to `mark-item` in Listing 12-4.

Listing 12-5 Specializing the `do-marking` method's parameters

```
define method do-marking (event :: <tile-type-event>,
                                  #key tile-type: the-type)
    if (the-type = event.tile-type)
        event.menu-item.mark := #t;
    else
        next-method();
    end if;
end method;
```

The technique of specializing the `do-marking` method rather than the `mark-item` method hides much of the logic that would be required to test each menu item against a slot in its event before determining whether to mark the item.

Responding to Menu Events

When a menu event is dispatched, event handlers and their behaviors may handle the event. For general information about event handling, see the chapter "Event Handling" on page 241. You specialize `do-event` methods on an event handler or specialize `behavior-event` methods for behaviors associated

with an event handler. These methods specify the application logic to execute when the event is dispatched.

Listing 12-6 shows a `do-event` method that is specialized on an event handler and on a singleton identifier of a menu event.

Listing 12-6 An event handler's response to a menu event

```
define method do-event (view :: <0-tile-view>,
                                event :: <menu-event>,
                                id == #"tile-show-dual")
    ignore(event, id);

    view.show-dual := ~ view.show-dual;
    invalidate-all(view);
end method;
```

If you specialize a menu event by its class, it may not be necessary to specialize on its identifier. Listing 12-7 shows the `do-event` method that responds to any `tile-type` menu event.

Listing 12-7 An event handler's response to a subclassed menu event

```
define method do-event (view :: <0-tile-view>,
                                event :: <tile-type-event>,
                                id :: <keyword>)
    ignore(id);
    reset-tile-type(view, event.tile-type);
end method;
```

Listing 12-8 shows a behavior's response to a menu event.

Listing 12-8 A behavior's response to a menu event

```
define method behavior-event (behavior :: <test-behavior>,
                                    next-behaviors :: <list>,
                                    main-handler :: <main-handler>,
```

```
                                    event :: <menu-event>,
                                    id == #"new")
    ignore(behavior, next-behaviors, event, id);

     post-command(main-handler,
            make(<new-window-command>,context: main-handler));
end method;
```

Setting up the About Menu Item

The Framework sets up the About menu item for you. All you need do is assign the title of the item. For example, you can use the following statement to set the title after you call `init-framework`:

```
*apple-menu*.about-item.title := "About MyApp…";
```

You must provide a `do-event` method specialized on the identifier `#"about-box"`.

Menu Class Reference

The following sections describe the exported classes that are intended to be used by application developers to support menus.

The <menu-element> Class

The `<menu-element>` class defines objects that maintain information that are used both by menu items and by menus themselves. You do not use `<menu-element>` objects directly; use `<menu>` and `<menu-item>` objects instead. The `<menu-element>` class is exported so that you can specialize methods on the `<menu-element>` instead of specializing methods with identical implementations on `<menu>` and `<menu-item>`.

Listing 12-9 shows the `<menu-element>` class definition.

Listing 12-9 The <menu-element> class definition

```
define primary open class <menu-element> (<object>)
  slot parent-menu,
    init-value: #f,
     init-keyword: parent-menu:;

  slot index :: <integer>,
    init-keyword: index:;

  // the following indicates the state of the toolbox menu
    slot toolbox-enabled? :: <boolean>, init-value: #f;

  // private
  // use title, title-setter, enabled? and enabled?-setter to access
    // these
  slot %title :: <string>,
     init-value: "",
     init-keyword: title:;

  slot %enabled? :: <boolean>,
     init-value: #f;

end class;
```

Slot descriptions

`menu-bar` Specifies the menu bar to which a menu or menu item is attached. The Framework sets up the association with the application's menu bar for you and stores the association in the `*menu-bar*` variable. If you want to associate a menu or item with another menu bar, you can use the `menu-bar:` keyword; otherwise menus and elements are associated with the menu bar represented by the `*menu-bar*` variable. The `menu-bar` slot is not exported. Changing the association of this slot with the `*menu-bar*` variable requires modifying the Framework.

`parent-menu` Specifies the menu associated with a menu item or the parent menu item associated with a submenu in a hierarchial menu. Typically, the parent menu is the `<menu>` object to which a list of items are specified in the `make`

	function for the menu or it is the `<menu>` object specified as the first required parameter in a call to `add-menu-item`. If the `parent-menu` slot is not set in one of these ways, use the `parent-menu`: keyword to specify the owning menu for a menu item or submenu; otherwise, the value of the `parent-menu` slot is false (`#f`). The `parent-menu` slot is not exported.
`index`	Specifies the position of a menu item in a menu. Typically, the value of the `index` slot is determined by the order that items are specified in the `make` function for a `<menu>` object. The position can also be specified in a call to `add-menu-item`. If the `index` slot is not set in one of these ways, use the `index:` keyword to specify the position of an item in a menu starting from 1; otherwise, the value of the `index` slot is false (`#f`). The `index` slot is not exported.
`toolbox-enabled?`	Internal; not exported.
`%title`	Specifies the title of a menu or menu item. Use the `title:` keyword to specify the title; otherwise the title is empty when you create a menu or menu item. Call `title` and `title-setter`, respectively, to get or set the title of a menu or menu item. The `%title` slot is not exported.
`%enabled?`	Specifies whether the menu or menu item is enabled. You can call `enabled?` and `enabled?-setter`, respectively, to get or set the value of this slot. The `%enabled?` slot is not exported.

Initialization

No `initialize` method is defined for `<menu-element>` objects.

The <menu-item> Class

The `<menu-item>` class defines objects that represent the menu items in a menu. You typically create these objects when you create the menu that contains them. For an example of making menus and menu items, see "Creating Menus and Menu Items" on page 287.

Listing 12-10 shows the `<menu-item>` class definition.

Listing 12-10 The <menu-item> class definition

```
define primary sealed class <menu-item> (<menu-element>)
  // the following indicates the state of the toolbox menu
  slot toolbox-mark :: <integer>,
     init-value: 0;

  // private
  // use generic functions event, mark, icon, style, command-key,
  // and their setters to access these.
  slot %event :: <object>,
     init-value: #f;

  slot %mark :: <integer>,
     init-value: $noMark,
     init-keyword: mark:;

  slot %icon :: <integer>,
     init-value: 0,
     init-keyword: icon:;

  slot %style :: <integer>,
     init-value: 0,
     init-keyword: style:;

  slot %command-key,
     init-value: #f,
     init-keyword: command-key:;
end class;
```

Slot descriptions

`toolbox-mark` Internal; not exported.

`%event` Specifies the event object associated with the menu item. Use the `event:` or `identifier:` keywords to specify the event object, as described under "Initialization;" otherwise, no event object is specified when you create the menu item. You can call `event` and `event-setter`, respectively, to get or set the value of this slot. The `%event` slot is not exported.

%mark	Specifies the mark to the left of the title; for example, a check mark. Use the mark: keyword to specify the mark; otherwise, no mark is present when you create the menu item. You can call mark and mark-setter, respectively, to get or set the value of this slot. The %mark slot is not exported.
%icon	Specifies the icon associated with the menu item. Use the icon: keyword to specify the icon; otherwise, no icon is specified. You can call icon and icon-setter, respectively, to get or set the value of this slot. The %icon slot is not exported.
%style	Specifies the style associated with the menu item. Use the style: keyword to specify the style; otherwise, the style is Plain. You can call style and style-setter, respectively, to get or set the value of this slot. The %style slot is not exported.
%command-key	Specifies the keyboard equivalent for the menu item. Use the command-key: keyword to specify the keyboard equivalent; otherwise, no keyboard equivalent is defined. You can call command-key and command-key-setter, respectively, to get or set the value of this slot. . The %command-key slot is not exported.

Initialization

The initialize method for <menu-item> objects sets up the association between the <menu-item> object and a <menu-event> object. In your make function, use the event: keyword if the <menu-event> object already exists. Use the identifier: keyword if you want to make an object that is subclassed from <menu-event>. The initialize method's parameters defined as follows:

```
define method initialize (item :: <menu-item>,
                                  #key event: menu-event,
                                  identifier: id)
```

item	The menu item.
menu-event	The event to associate with the menu item.
id	The ID of the event to associate with the menu item.

If the event: keyword is specified, the object specified by the menu-event parameter becomes the event associated with the menu item. If the identifier:

keyword is specified, the Framework creates a <menu-event> object whose identifier slot contains the value specified by the id parameter.

The <separator-item> Class

The <separator-item> class defines the divider lines used to separate related menu items. You typically create <separator-item> objects when you create a <menu> object. For an example of making a <separator-item> object, see "Creating Menus and Menu Items" on page 287.

Listing 12-11 shows the <separator-item> class definition.

Listing 12-11 The <separator-item> class definition

```
define primary sealed class <separator-item> (<menu-item>)
    inherited slot %title,
        init-value: "-";
end class;
```

Slot descriptions

%title	Internal; not exported. Do not change the title of a <separator-item> object.

Initialization

No initialize method is specified for <separator-item> objects.

The <menu> Class

The <menu> class defines objects that represent the menus. You typically create <menu> objects immediately after calling init-framework to initialize the Framework, although you can create a menu at any time after the Framework is initialized. For an example of making menus, see "Creating Menus and Menu Items" on page 287.

You may define subclasses of <menu>, although it is seldom necessary to do so. For example, the Framework defines the Apple menu as a subclass because it must handle the situation where items in the Apple menu are added

dynamically,. This situation requires additional information to be maintained for the Apple menu than for other menus.

Listing 12-12 shows the <menu> class definition.

Listing 12-12 The <menu> class definition

```
define primary sealed class <menu> (<menu-element>)
  slot items :: <stretchy-object-vector>,
     init-function: method ()
                      make(<stretchy-object-vector>, size: 0)
                    end;

  slot id :: <integer>;

  slot menu-resource-id :: <integer>,
     init-value: 0,
     init-keyword: resource-id:;

  slot toolbox-menu :: <MenuHandle>,
     init-value: as(<MenuHandle>, 0);

  slot installed? :: <boolean>,
     init-value: #f;

  // set this to #t if you want to bypass the menu setup code
  // and have this menu and all its items enabled at all times
  slot always-enabled? :: <boolean>,
     init-value: #f;
end class;
```

Slot descriptions

`items` Specifies the menu items associated with the menu. For information about menu items, see "The <menu-item> Class" on page 294. To add a menu item to a menu, call `add-menu-item`. Only the getter method is defined for this slot.

ID	Specifies the menu's ID, starting from 128. The initial value, false (#f), is changed by the initialize method, described below.
toolbox-menu	Specifies the Menu Manager menu record associated with the object. The Framework sets this slot. Only the getter method is defined for this slot.
installed?	Specifies whether the menu is installed. To install a menu, call install-menu. To determine whether a menu is installed, call installed-menus.
always-enabled?	Specifies whether you want the menu item to always be enabled (#t) or enabled each time by the do-setup-menus function (#f). By default, the do-setup-menus function controls how the menu item is enabled.

Initialization

The initialize method for <menu> objects sets up a <menu> object and associates menu items with the menu and, optionally, installs the menu in the menu bar. The <menu> object can be created from a resource. The initialize method's parameters are defined as follows:

```
define method initialize (menu :: <menu>,
                                #key
                                   resource-ID: resource-ID,
                                   title: t ("Menu"),
                                   items: menu-items,
                                   install: install)
```

menu	The menu.
resource-ID	The resource ID from which to load the menu.
t	The title.
menu-items	The list of menu items to associate with the menu.
install	Specifies whether to install the menu as soon as it is made.

If the resource-ID: keyword is specified, the values of the object's slots are obtained from the specified resource. If the resource-ID: keyword is not specified, the Framework creates a new menu with the next sequential ID. In this case, you should specify the title: keyword; otherwise, the title is "Menu."

If you specify true (#t) for the install: keyword, the menu is installed in the menu bar immediately. If you do not specify the install: keyword or if you specify false (#f) for it, you must call install-menu when you want to add the menu to the menu bar.

The <apple-menu> Class

The <apple-menu> class represents the Apple menu. The Framework creates an <apple-menu> object for you and assigns it to the *apple-menu* variable. You can use this variable to access the menu and the menu item that corresponds to the About dialog box. For an example of setting up this menu item, see "Setting up the About Menu Item" on page 292.

Listing 12-13 shows the <apple-menu> class definition.

Listing 12-13 The <apple-menu> class definition

```
define primary sealed class <apple-menu> (<menu>)
    slot about-item,
        init-value: #f;
    slot last-user-item :: <integer>,
        init-value: 0;
end class;
```

Slot descriptions

about-item	Specifies the object that corresponds to the About menu item. You can set the slots of the <menu-item> object stored in the about-item slot. For more information about <menu-item> objects, see "The <menu-item> Class" on page 294.
last-user-item	Internal; not exported.

Initialization

The initialize method for the <apple-menu> object sets up the Apple menu and the object that represents the About menu item. The initialize method's parameters are defined as follows:

```
define method initialize (menu :: <apple-menu>, #key)
```

`menu` The Apple menu object.

If you do not change the title or style of the `<menu-item>` object stored in the `about-item` slot, the title of the menu item is "About Dylan Framework…" in plain text.

The <menu-event> Class

The `<menu-event>` class defines menu events, which are used to represent the user's action of selecting a menu item. The menu event object that corresponds to a menu item is created when you create a `<menu-item>` object. If you specify the `identifier:` keyword when you create a menu item object, you do not need to create a corresponding `<menu-event>` object; the Framework does this for you. For information about `<menu-item>` objects and how they relate to `<menu-event>` objects, see "The <menu-item> Class" on page 294.

You can define subclasses of the `<menu-event>` class. For example, you may want your event to include additional slots. If you subclass `<menu-event>`, you must create an object of your subclass and specify it as the event corresponding to a menu item. For an example of how to define a subclass of `<menu-event>`, see "Responding to Menu Events" on page 290.

Listing 12-14 shows the `<menu-event>` class definition.

Listing 12-14 The `<menu-event>` class definition

```
define primary sealed class <menu-event> (<event>)
    slot menu-item :: <menu-item>,
        init-keyword: menu-item:;
end class;
```

Slot descriptions

`menu-item` Specifies the menu items associated with this event. Use the `menu-item:` keyword to specify a menu item; otherwise, the event is never dispatched.

Initialization

The `initialize` method for `<menu-event>` objects associates a menu event with a menu item. The `initialize` method's parameters are defined as follows:

```
define method initialize (event :: <menu-event>, #key)
```

Functions for Menus

This section identifies methods and generic functions associated with menus and menu handling. Getter and setter functions are identified with their associated slots. For slot descriptions, see "About Menus" on page 283.

Methods and generic functions related to menus are divided into the following groups:

- those that manipulate entire menus
- those that manipulate menu items
- those that are associated with menu events

Table 12-1 identifies the functions that operate on entire menus.

Table 12-1 Functions for manipulating entire menus

Function	Purpose	Call	Sp.
`install-menu`	Install a menu into the application's menu bar.	√	
`installed-menus`	Retrieve the list of menus that are currently installed in the application's menu bar.	√	
`remove-menu`	Remove a menu from the application's menu bar.	√	
`get-menu`	Retrieve a menu from the application's menu bar.	√	

Table 12-2 identifies the functions that operate on menu items.

Table 12-2 Functions for manipulating menu items

Function	Purpose	Call	Sp
`enable-item`	Enable menu items.	√	
`mark-item`	Mark menu items.	√	
`do-enabling`	Perform actions to enable menu items.		√
`do-marking`	Perform actions to enable menu items.		√
`add-menu-item`	Add a menu or menu item to a menu.	√	
`find-menu-item`	Find a menu item.	√	
`add-menu-item -from-toolbox -menu`	Internal.		

Table 12-3 identifies the functions that operate on menu events.

Table 12-3 Functions for handling menu events

Function	Purpose	Call	Sp.
`do-setup-menus`	Install menus and enable menu items.		√
`behavior-setup-menus`	Install menus and enable menu items when the specified behavior is active.		√
`do-event`	Define the actions to take when an event occurs.		√
`behavior-event`	Define the actions to take when an event occurs while the specified behavior is active.		√

Menus

CHAPTER 13

Graphics

Contents

About Graphics 307
- About Graphics Ports and Buffers 309
- About Graphics Caches 310
- About Frames 310
- Frame Coordinate Systems 313
- About Regions 314
- About Text Styles 315
- About Color 316

Using Graphics Objects 317
- Changing an Object's Location and Extent 317
- Manipulating Regions and Rectangles 317
- Defining Text Styles 318
- Defining Colors 318

Graphics Objects Reference 319
- The <frame> Class 319
- Graphics Ports Classes 320
- Text Style Classes 322
- The QuickDraw Region Class 324
- RGB Color Class 324
- Graphics-Related Functions 325

Graphics classes provide support for drawing in views. This chapter describes classes that handle graphics, including those that represent

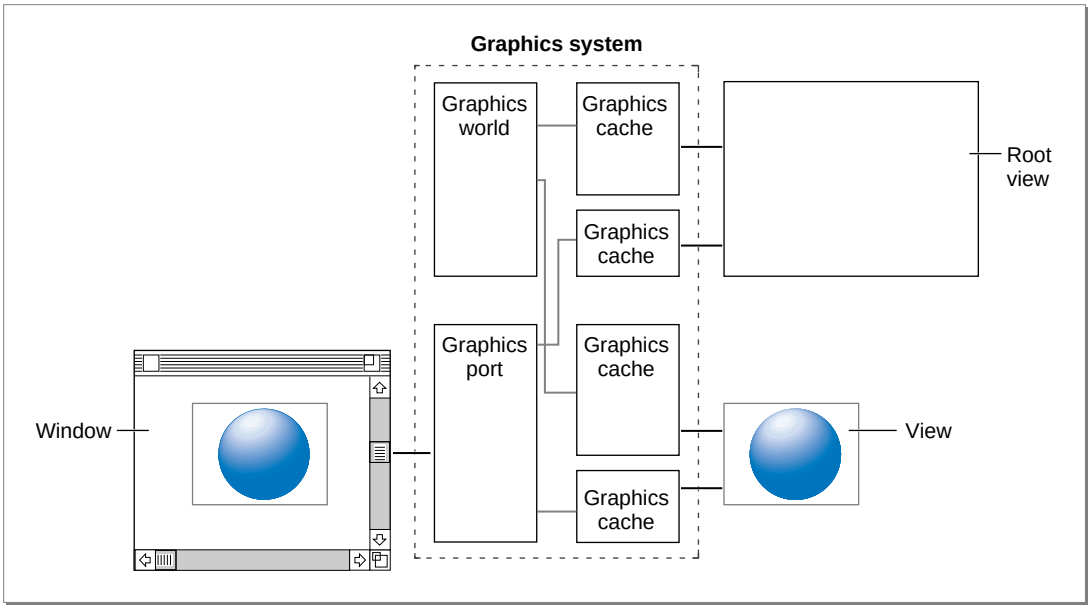
- frames, which define the location and extent of a window, view, or graphics port
- graphics ports, which define a drawing environment
- graphics buffers, which are associated with graphics ports
- graphics caches, in which clipping and focus information reside
- color and text style characteristics
- specific QuickDraw implementations of ports, graphics systems, caches, buffers, regions, and styles

About Graphics

Graphics concepts include topics that relate to the mechanism that manages views and windows so that they can be displayed onscreen or manipulated offscreen, and topics that related to geometric areas. The remainder of this section introduces the graphics mechanism. For information about the geometric aspects of graphics, see "About Frames" on page 310 and "About Regions" on page 314.

The graphics mechanism consists of several objects that operate within a **graphics system**, such as QuickDraw. The Framework provides one graphics system that supports both QuickDraw and QuickDraw graphics worlds (GWorlds). Figure 13-1 shows the mechanism.

Figure 13-1 Graphics mechanism



The graphics mechanism consists of three major kinds of objects:

- graphics ports
- graphics worlds
- graphics caches (internal; not a class)

A **graphics port** represents a port from a window on to the screen, such as a GrafPort in QuickDraw. For each window there is only one graphics port. The graphics port translates the memory structure of a window to a structure that can be displayed onscreen.

The Framework also provides a graphics port subclass that represents a **graphics world** (GWorld). This subclass includes additional slots to hold information associated with the graphics world as well as a buffer to hold the contents of the window offscreen. The Framework also provides a method to copy the graphics world buffer to the graphics port associated with the window, thus allowing you to draw the contents of the buffer onscreen.

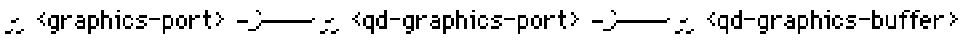
A **graphics cache** translates information about a view into a form that can be handled by a graphics port. For example, a graphics cache can store information about how to draw the view into the port, clipping and translation information, and other information that is relevant to a graphics port. A view has one graphics cache for each graphics port that it can draw into. Thus, if a view can appear in several windows, it has a cache for each of them.

In general, you need not be concerned with graphics ports and graphics caches. When you create a window, the Framework creates a graphics port for you. [In the future, you may need to specify the kind of graphics port.] When you add a view to a window, or a subview to a view that is already associated with a window, the Framework automatically sets up a cache to match the requirements of the window's graphics port.

About Graphics Ports and Buffers

Graphics port classes represent the port, such as the QuickDraw GrafPort, used to draw the contents of a window. The Framework provides classes that handle QuickDraw, QuickDraw graphic worlds, and buffered views based on graphic worlds. Figure 13-2 shows the graphics port classes.

Figure 13-2 Graphic port and buffer classes



Objects created from the classes in Figure 13-2 represent the following kinds of graphic systems:

`<graphics-port>`	Represents an abstract class for a graphics port.
`<qd-graphics-port>`	Represents a QuickDraw graphics port.
`<qd-graphics-buffer>`	Represents a QuickDraw graphics world, which can be used for offscreen drawing.

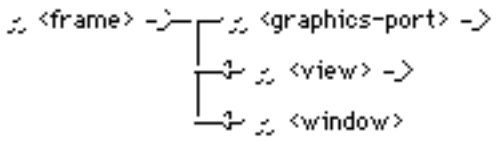
About Graphics Caches

Graphics caches holds information to translate a view's characteristics into a form that can be used by a graphics port. The Framework handles setting up the correct graphics cache for you when you add a view as a subview to one that is already associated with a window; for example, when you specify a subview list while creating the window or when you later add a view as a subview to a view in the list.

About Frames

Frames are inherited by classes whose objects need to maintain their location and extent in relation to some other object or on a screen or within an offscreen buffer. These classes are shown in Figure 13-3.

Figure 13-3 Frame classes



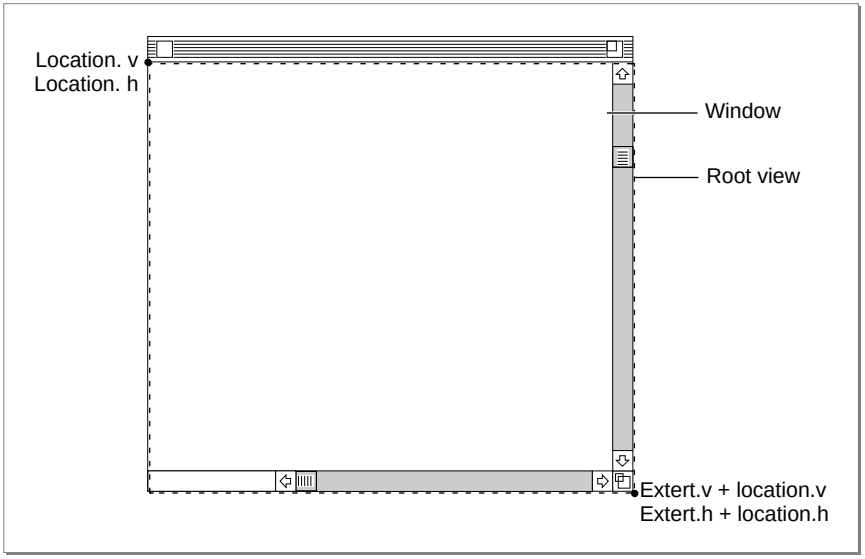
The classes that inherit from the <frame> class are identified as follows:

`<graphics-port>`	Represents a graphics cache from which you may create a subclass.
`<view>`	Represents an area within a window typically used for drawing.
`<window>`	Represents a Window Manager window.

Note
Windows and views are also event handlers, thus they also inherit (directly or indirectly) from the `<event-handler>` class. ◆

The `<frame>` class allows an area to be defined by the objects of its subclasses. Figure 13-4 shows the area defined by a window and the window's root view.

Figure 13-4 Location and extent for windows and root views



The **location** is a point that identifies the top-left corner of the area. Points are defined by their vertical (`v`) and horizontal (`h`) slots.

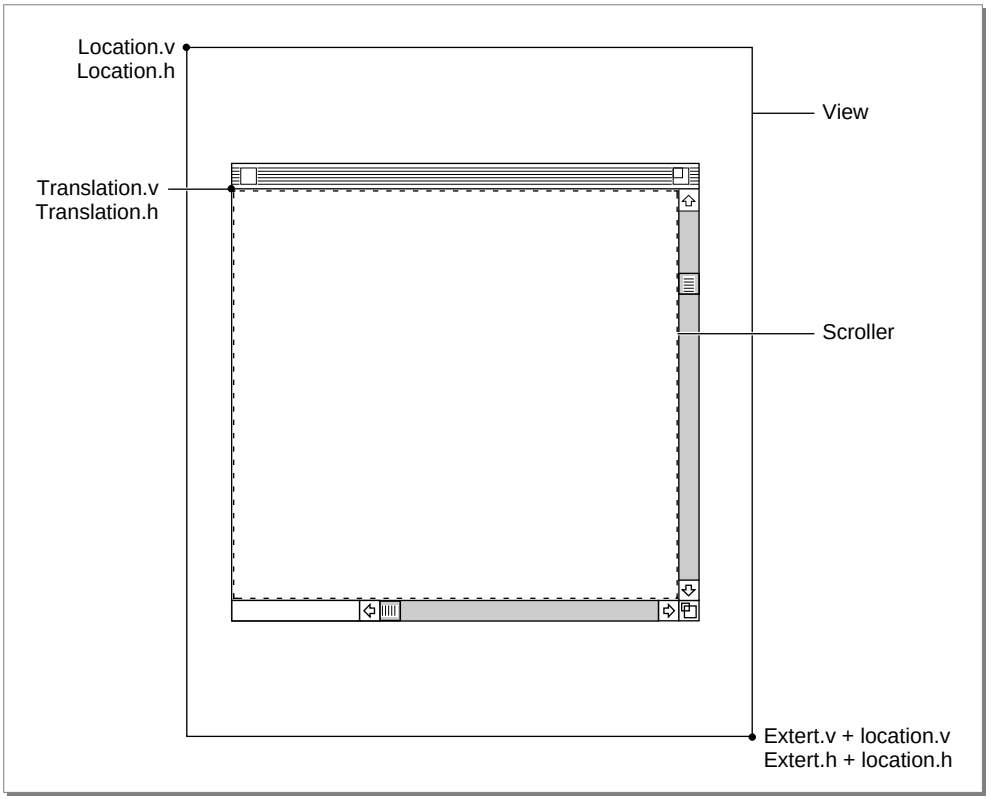
The window's location defines the top-left corner of the content area of the window in relation to the top-left corner of the main monitor. The root view's location defines its location in relation to the window's location. Because, the Framework always defines the root view to be the same area as the window, the root view's location is always `$zero-point`, which is point (0,0), meaning that if the window moves, its root view also moves to the same location.

The location of a subview is relative to its superview. Thus, if a subview is defined with its v slot as 20 and its h slot as 30, the subview's location is 20 pixels down and thirty pixel's to the right of its superview's location. If the superview is the root view, then the subview is 20 pixels down and thirty pixel's to the right of the top-left corner of the window's content area.

The **extent** is a point that identifies the size of the area. The botton-right corner of an area is a point representing the sum of the respective `h` and `v` slots in the object's location and extent. The root view's extent is the same as the window's extent, as shown in Figure 13-4.

A `<frame>` object also provides a slot that represents the translation between a scroller and its subviews. Because a view in a window is relative to its superview, its location does not change. If a view is scrolled in a window, however, the relation between that view and the scroller view can change, allowing different parts of the view to show through the scroller. Figure 13-5 shows a scroller view with one subview.

Figure 13-5 Translation between scroller views and its subviews.



The **translation** is a point that defines how the location would appear to change if the subview is scrolled in the window. For example, if the view is scrolled down to show more of its bottom content, which means that the view moves up in relation to the scroller, the vertical translation increases to accommodate the greater apparent distance between the location of the subview and the location of the scroller view. The translation is handled by the Framework for you when you call appropriate methods, such as `local-bounds` to determine the bounding rectangle for a view. If you access the `location` and `extent` slots of a scroller view's subview directly, you may need to handle the translation yourself.

Frame Coordinate Systems

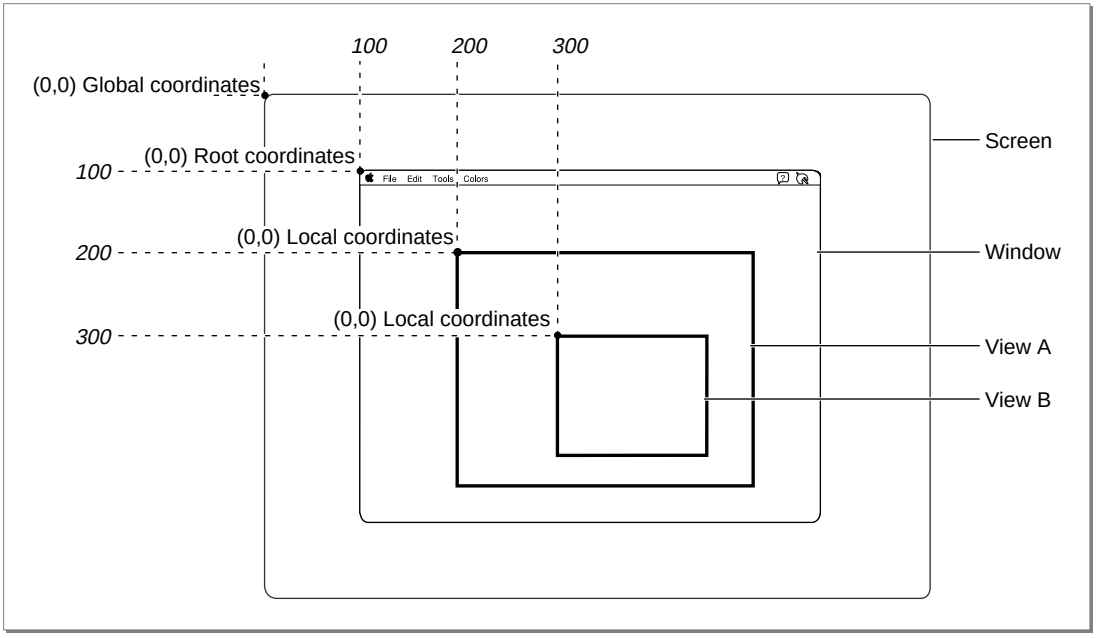
The Framework supports conversions between four coordinate systems. These coordinate systems are:

- **local**, in which coordinates are relative to the object. For example, a view's location and extent are defined in local coordinates, which are the coordinates in the context of the view.
- **super**, in which a view's local coordinates are defined in the coordinates of its superview.
- **root**, in which coordinates are defined in the coordinates of the root view. Root coordinates are the same as local coordinates in the Macintosh Window Manager. The top-left coordinate of a root view is always point (0,0), corresponding to the top-left corner of the window.
- **global**, in which coordinates are mapped into the coordinates determined by top-left corner of the main monitor. Global coordinates are the same as global (screen) coordinates in the Macintosh Window Manager. The location and extent of a window is always defined in global coordinates.

In general, the Framework allows you to work with local coordinates, which are coordinates that are relative to the object. For example, the Framework handles conversion of the location of a mouse-down event in a window and returns the location in the local coordinates of the view. Thus, a do-event method whose parameter is specialized on a view, does not need to convert the location from another coordinate system.

Figure 13-6 provides an example of the coordinate systems for which conversion is supported by the Framework.

Figure 13-6 Framework coordinate systems



Assume that translation is `$zero-point`, thus not a factor in the conversion. If view B is a subview of view A, the location of view B in its superview's coordinate system is at point (100, 100). The location of view B in the root coordinate system is at point (200, 200). The location of the view B in global coordinates is at point (300, 300).

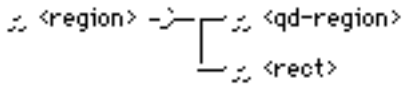
About Regions

Regions are classes that represent an area. The Framework provides the `<region>` abstract class and two subclasses:

- QuickDraw regions, `<qd-region>`
- Rectangles, `<rect>`

You can define subclasses of the <region> class and create objects from them. If you use the QuickDraw graphics system, you typically create objects from the <qd-region> and <rect> classes. Figure 13-7 shows the region classes.

Figure 13-7 Region classes



Objects created from the classes in Figure 13-7 represent the following kinds of geometric structures:

<region>	Represents an abstract region.
<qd-region>	Represents a region associated with a QuickDraw region handle.
<rect>	Represents a rectangle.

About Text Styles

Text styles define the font, size, and style used in a type face. The Framework provides the <text-style> abstract class and a subclass that represents QuickDraw text styles. You can define subclasses of the <text-style> class and create objects from these subclasses. If you use the QuickDraw graphics system, however, you typically create objects from the <qd-text-style> class. Figure 13-8 shows the text style classes.

Figure 13-8 Text style classes

Objects created from the classes in Figure 13-8 represent the following kinds of text styles:

`<text-style>`	Represents a text style from which you can define subclasses to create objects.
`<qd-text-style>`	Represents a text style for QuickDraw text.

The Framework provides the following constants for QuickDraw text styles, as shown in Table 13-1:

Table 13-1 QuickDraw text style constants

Text Style	Description
`$Chicago12`	Chicago font, 12-point regular.
`$Geneva9`	Geneva font, 9-point regular.
`$Geneva12`	Geneva font, 12-point regular.

About Color

The RGB color scheme is used to represent color information. You create objects from the `<rgb-color>` class. The Framework provides the following color constants for RGB colors, as shown in Table 13-2:

Table 13-2 RGB color constants

Color	Description
`$no-color`	No color; used for “transparent” drawing.
`$black-color`	Black
`$white-color`	White
`$red-color`	Red
`$green-color`	Green
`$blue-color`	Blue

Using Graphics Objects

The following sections show some of the common tasks for which you need frames, regions and rectangles, text styles, and colors. For examples of using graphics buffers, see "Setting Up a View for a Graphics World" on page 348.

Changing an Object's Location and Extent

You can determine the location and extent of an object by calling `local-bounds`. If you want the location and extent of a view's superview, you can call `super-bounds`. Listing 13-1 shows an example of setting the location and extent of a view to the match the location and extent of its superview.

Listing 13-1 Changing a view's location and extent

```
let new-bounds = view.super-bounds;
view.location := point(new-bounds.left, new-bounds.top);
view.extent := point(new-bounds.right, new-bounds.bottom);
```

Note
The syntax `view.super-bounds` is equivalent to `super-bounds(view)`. ◆

Manipulating Regions and Rectangles

The Framework provides several methods for performing operations on regions and rectangles, including union, intersection, difference, xor, determining if a point exists in one, conversion between them, and so on. For a list of these methods, see Table 13-6, "Functions for QuickDraw regions" on page 328.

Listing 13-2 shows how to define a rectangle from two points and create a QuickDraw region from the union of two rectangles.

Listing 13-2 Manipulating regions and rectangles

```
let old-draw-rect = make(<rect>);
set-rect-from-points (old-draw-rect,
                          tracker.start-point,
                          last-point);
let redraw-region = union-region(old-draw-rect, draw-rect);
```

Defining Text Styles

Text styles define the font, size, and style of a typeface. The Framework provides constants for several QuickDraw typefaces used in windows and dialog boxes. For information about these constants, see "About Text Styles" on page 315.

You can define constants for your own typefaces. Listing 13-3 shows the definition of a text-style constant for Helvetica, 10-point bold-italic.

Listing 13-3 Defining a text style

```
define constant Helv10-bold-italic = make (<qd-text-style>,
                                           font: "Helvetica",
                                           size: 10,
                                           bold: #t,
                                           italic: #t);
```

Defining Colors

RGB colors specify a red, green, and blue component, each of which can have a value from 0 to 65535, inclusive. If each component is 0, the color is black. If a component is 65535, it is red, green, or blue, depending on the component. If all components are 65535, the color is white.

The Framework provides color constants for red, green, blue, white, and black. It also provides the constant `$no-color`; when used, no drawing occurs. For information about color constants, see "About Color" on page 316.

The Framework provides an `rgb-color` method that you can use to create color objects. Listing 13-4 shows the definition of a color constant for gray.

Listing 13-4 Defining a color constant

```
define constant $gray-color = rgb-color (40000, 40000, 40000);
```

Graphics Objects Reference

The following sections identify the graphics-related classes in the Framework and describe their slots.

The <frame> Class

The `<frame>` class allows an area to be defined for an object. The Framework defines graphic ports, windows and views as frames. You do not need to define subclasses from the `<frame>` object, nor create <frame> objects directly. For more information about frames, see "About Frames" on page 310.

Listing 13-5 shows the `<frame>` class definition.

Listing 13-5 The `<frame>` class

```
define primary open class <frame> (<object>)
    slot location :: <point>,
     init-value: $zero-point,
     init-keyword: location:;
    slot extent :: <point>,
     init-value: $zero-point,
     init-keyword: extent:;
end class;
```

Slot descriptions

`location` The top-left corner of the frame.

`extent` The size of the frame.

Initialization

No `initialize` method is defined for `<frame>` objects.

Graphics Ports Classes

Graphics ports classes represent a port through which an image can be rendered. The Framework provides an abstract class, `<graphics-port>`, and two subclasses, `<qd-graphics-port>` and `<qd-graphics-buffer>`.

You do not need to define subclasses for graphics ports. The Framework creates a `<qd-graphics-port>` object when you create a window. You must create a `<qd-graphics-buffer>` object when you want to use a graphics world (GWorld) for offscreen drawing.

Listing 13-6 shows the class definition for `<graphics-port>` objects.

Listing 13-6 The `<graphics-port>` class definition

```
define primary open class <graphics-port> (<frame>)
end class;
```

Listing 13-7 shows the class definition for `<qd-graphics-port>` objects.

Listing 13-7 The `<qd-graphics-port>` class definition

```
define primary sealed class <qd-graphics-port> (<graphics-port>)
  slot graf-port :: <GrafPort>,
    init-value: as(<GrafPort>, 0),
    init-keyword: graf-port:;
  slot owning-window,
    init-value: #f,
    init-keyword: window:;

  slot port-is-window? :: <boolean>,
```

```
    init-value: #f,
    init-keyword: is-window:;
end class;
```

Slot descriptions

`graf-port`	The Window Manager GrafPort data structure.
`owning-window`	The window associated with the graphics port.
`port-is-window?`	Specifies whether the graphics port associated with a window (`#t`), or if it is associated with an offscreen buffer (`#f`).

Initialization

No `initialize` method is defined for `<qd-graphics-port>` objects.

Listing 13-8 shows the class definition for `<qd-graphics-buffer>` objects.

Listing 13-8 The `<qd-graphics-buffer>` class definition

```
define primary sealed class <qd-graphics-buffer> (<qd-graphics-port>)
  slot world :: <CGrafPort>,
     init-value: as(<CGrafPort>, 0);
  slot pix-map :: <PixMapHandle>,
     init-value: as(<PixMapHandle>, 0);
  slot color-table :: <CTabHandle>,
     init-value: as(<CTabHandle>, 0),
     init-keyword: color-table:;
  slot pixels :: <machine-pointer>,
     init-value: $null-machine-pointer;
  slot lock-count :: <integer>,
  init-value: 0;

  slot depth :: <integer>,
     required-init-keyword: depth:;
end class;
```

Slot descriptions

`world`	A color GrafPort.

pix-map	A handle to a pixel map.
color-table	A handle to a color table.
pixels	A buffer that contains the image you wish to render.
lock-count	The current number of caches associated with the buffer.
depth	The depth of the buffer, in bits.

Initialization

The initialize method for <qd-graphics-buffer> objects sets up the toolbox data structures and calls NewGWorld. For information about the NewGWorld function, see *Inside Macintosh: Imaging With QuickDraw*. The initialize method's parameters are defined as follows:

```
define method initialize (buffer :: <qd-graphics-buffer>,
                          #key gworld-flags: flags (0)) => ()
```

buffer	The graphics buffer.
flags	Flags passed to the NewGWorld function.

If the gworld-flags: keyword is not specified, 0 is used.

Text Style Classes

Text style class objects represent character sets for display of text. The Framework provides an abstract class, <text-style>, and a class that represents character sets for use with QuickDraw, <qd-text-style>. Listing 13-9 shows the definition of these classes.

Listing 13-9 The <text-style> and <qd-text-style> classes

```
define free open class <text-style> (<object>)
end class;

define primary sealed class <qd-text-style> (<text-style>)
  slot font :: <string>, init-value: #f,
     init-keyword: font:;
  slot font-number :: <integer>,
     init-value: 0;
```

```
  slot font-size :: <integer>,
     init-value: 12,
     init-keyword: size:;
  slot font-style :: <integer>,
     init-value: 0,
     init-keyword: style:;
end class;
```

Slot descriptions

`font`	The font name, such as Geneva or Chicago.
`font-number`	The font number.
`font-size`	The font size.
`font-style`	The font style.

Initialization

No `initialize` method is defined for `<text-style>` objects. The `initialize` method for `<qd-text-style>` objects determines the font name from the font number if a number is specified and allows you to specify the style, The method's parameters are defined follows:

```
define method initialize ( style :: <qd-text-style>,
                           #key
                             font-number: font-num,
                             bold: bold,
                             italic: italic,
                             underline: underline,
                             outline: outline,
                             shadow: shadow,
                             condense: condense,
                             extend: extend) => ()
```

`style`	The style.
`font-num`	The font number.
`bold`	True (`#t`) if the characters are bold face.
`italic`	True (`#t`) if the characters are italic.
`underline`	True (`#t`) if the characters are underlined.
`outline`	True (`#t`) if the characters are outlined.

`condense` True (`#t`) if the characters are condensed.

`extend` True (`#t`) if the characters are extended.

If both the `font:` and `font-number:` keywords are specified, the font number is used to determine the font.

The QuickDraw Region Class

The Framework provides a QuickDraw region class that you may use to store QuickDraw data. Listing 13-10 shows the class definition.

Listing 13-10 The `<qd-region>` class

```
define primary sealed class <qd-region> (<region>)
  slot rgn-handle :: <RgnHandle>,
     init-function: method() new-region-handle(); end;
end class;
```

Slot descriptions

`rgn-handle` A handle to a QuickDraw region.

Initialization

The `initialize` method for `<qd-region>` objects registers the region handle for garbage collection when the region is no longer in use. The `initialize` method's parameters are defined as follows:

```
define method initialize (rgn :: <qd-region>, #key) => ()
```

`rgn` The region.

RGB Color Class

The Framework provides an `<rgb-color>` class that represents RGB colors and supports conversion between objects of this class and the color QuickDraw `RGBColor` data structure. Listing 13-11 shows the definition of the `<rgb-color>` class.

Listing 13-11 The `<rgb-color>` class

```
define class <rgb-color> (<color>)
  slot red :: <integer>,
    setter: #f,
    required-init-keyword: red:;

  slot green :: <integer>,
    setter: #f,
    required-init-keyword: green:;

  slot blue :: <integer>,
    setter: #f,
    required-init-keyword: blue:;
end class;
```

Slot descriptions

`red`	The red component of an RGB color. Use the `red:` keyword to define the component. You cannot change the component after it has been is defined.
`green`	The green component of an RGB color. Use the `green:` keyword to define the component. You cannot change the component after it has been is defined.
`blue`	The blue component of an RGB color. Use the `blue:` keyword to define the component. You cannot change the component after it has been is defined.

Initialization

The `initialize` method is defined for color objects.

Graphics-Related Functions

Graphics-related functions are grouped in the following categories:

- functions that operate on frames to handle conversion between coordinate systems
- functions for determining or setting the characteristics of graphics ports

- functions for manipulating a graphic buffer associated with a graphics port
- functions that manipulate a QuickDraw region
- functions for manipulating text styles
- functions for manipulating colors

The following tables identify the functions in each category and describe their purpose and use.

Table 13-3 shows functions for frames, which include windows, views, and graphics ports.

Table 13-3 Functions for frames

Function	Purpose	Call	Sp.
`super-frame`	Return the frame of the next event handler in the target chain.	√	
`set-bounds`	Set the location and extent of a frame.	√	
`local-bounds`	Determine the location and extent of a frame in local coordinates.	√	
`super-bounds`	Determine the location and extent of a frame in the coordinates of its next handler.	√	
`offset-in-super`	Determine the offset of a frame in its next event handler's frame.	√	
`local-to-super`	Convert a point or region from local coordinates to the coordinates of its next event handler.	√	
`super-to-local`	Convert a point or region from the coordinates of its next event handler to the local coordinates of the specified frame.	√	
`local-to-global`	Convert a point or region from local coordinates to global coordinates.	√	
`global-to-local`	Convert a point or region from global coordinates to the local coordinates of the specified frame.	√	

Table 13-4 identifies functions for determining or setting the characteristics of graphics ports.

Table 13-4 Functions for graphics ports

Function	Purpose	Call	Sp.
`close`	Close a graphics port.	√	
`visible?`	Determines whether part of a graphics port's window is visible. Returns (`#t`) if any part of the window is visible or the graphics port does not own a window.	√	
`set-origin`	Set the origin of the graphics port to the specified global coordinates.	√	

Table 13-5 identifies the functions for manipulating the buffer associated with a graphics port.

Table 13-5 Functions for graphics buffers

Function	Purpose	Call	Sp.
`dispose`	Dispose of a buffer.	√	
`lock-buffer`	Lock a buffer.	√	
`unlock-buffer`	Unlock a buffer.	√	
`set-buffer-port`	Set up a graphics port for a buffer.	√	
`restore-buffer-port`	Restore a graphics port. (Used with `set-buffer-port`.)	√	
`draw-buffer`	Draw the contents of a buffer to the active graphics port.	√	
`make-pict`	Create a picture handle from the contents of a buffer.	√	

Table 13-5 Functions for graphics buffers

Function	Purpose	Call	Sp.
`clear-buffer`	Clear a buffer.	√	
`copy-buffer`	Make a copy of a buffer.	√	
`set-source-bounds`	Retrieve the extent of a buffer into a QuickDraw rectangle.	√	

Table 13-6 identifies the functions that manipulate a QuickDraw region.

Table 13-6 Functions for QuickDraw regions

Function	Purpose	Call	Sp.
`make-region`	Create a region object for use with the specified graphics system.	√	
`get-region`	Create a region with the contents of a QuickDraw region handle.	√	
`terminate`	Dispose of a region.	√	
`clone`	Create a copy of a region.	√	
`set-rect-as-region`	Change the structure of a region to that of a rectangle.	√	
`as-rect`	Retrieve the contents of a region as a rectangle.	√	
`as-qd-region`	Retrieve the contents of a rectangle as a region.	√	
`empty-region?`	Determine whether the region is empty.	√	
`set-empty-region`	Clear the contents of a region.	√	
`point-in-region?`	Determine whether a point is in a region.	√	
`inset-region`	Shrink or expand a region.	√	
`offset-region`	Move a region.	√	

Table 13-6 Functions for QuickDraw regions

Function	Purpose	Call	Sp.
`union-region`	Create a region from the union of two regions.	√	
`intersect-region`	Create a region from the intersection of two regions.	√	
`regions-intersect?`	Determine whether the area of two regions intersect.	√	
`difference-region`	Create a region from the difference between two regions.	√	
`xor-region`	Create a region from the non-intersecting areas of two regions.	√	
`add-to-region`	Add a region to an existing region.	√	
`subtract-from-region`	Subtract a region from an existing region	√	
`intersect-with-region`	Determine the intersection between two regions.	√	
`frame-region`	Draw a border just inside a region's bounds.	√	
`paint-region`	Paint a region using the pen and pattern from the current graphics port.	√	
`erase-region`	Erase the region by redrawing it in the background color of the graphics port.	√	
`invert-region`	Reverse the bits for each pixel in the region.	√	
`fill-region`	Fill a region with the specified pattern.	√	

Table 13-7 identifies the functions for manipulating text styles.

Table 13-7 Functions for QuickDraw text styles

Function	Purpose	Call	Sp.
`set-port-style`	Set the current graphics port's text font, type face, and size.	√	
`normal?`	Get or set whether the type face is in normal style.	√	
`bold?`	Get or set whether the type face style is bold.	√	
`italic?`	Get or set whether the type face style is italic.	√	
`underline?`	Get or set whether the type face style is underlined.	√	
`outline?`	Get or set whether the type face style is outlined.	√	
`shadow?`	Get or set whether the type face style is shadowed.	√	
`condense?`	Get or set whether the type face style is condensed.	√	
`extend?`	Get or set whether the type face style is extended.	√	

Table 13-8 identifies functions for manipulating colors.

Table 13-8 Functions for colors

Function	Purpose	Call	Sp.
`rgb-color`	Create an `<rgb-color>` object.	√	
`choose-color`	Invoke the Color Picker dialog box.	√	
`get-color`	Create a `<rgb-color>` object from a toolbox `RGBColor` data structure.	√	
`set-color`	Set up an `RGBColor` data structure from an `<rgb-color>` object.	√	
`set-fore-color`	Set the foreground color of the current graphics port.	√	
`set-back-color`	Set the background color of the current graphics port.	√	

CHAPTER 13

Graphics

CHAPTER 14

Views

Contents

About Views and Supporting Classes 335
- View Hierarchies 337
- Adorners 339
- View Determiners 341
- Drawing in Views 342
- Highlighting 343
- Cursor Manipulation 344

Using View Objects 345
- Defining a View Class 345
- Setting Up Views 346
- Setting the Target View 347
- Drawing in Views 348
- Setting Up a View for a Graphics World 348
- Drawing With Buffered Views 349
- Setting Up Adorners 350
- Implementing a View Determiner 352

View Objects Reference 354
- The <view> Class 355
- Adorner Classes 360
- View Determiner Classes 362
- View-Related Functions 364
  - Functions for Graphics Support 367
  - Functions for Adorners 368
  - Functions for View Determiners 369

Views have two major purposes in the Framework—they handle drawing and they can respond to events. This chapter describes view classes and classes that support them:

- adorners, which allow borders or other adornments to be drawn as part of a view
- determiners, which specify rules for how a view can be associated with its superview.

For information about text views, see "Text Views" on page 373. For information about grid and list views, see "Grid and List Views" on page 387.

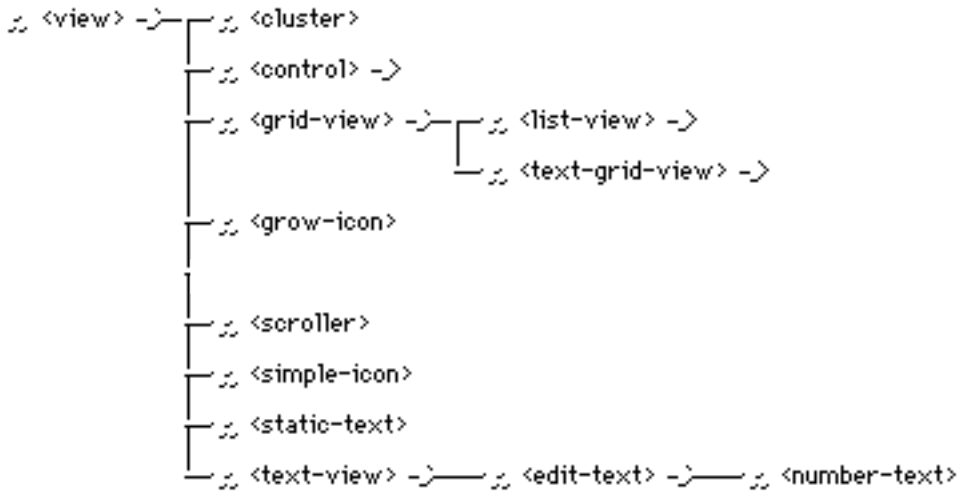
About Views and Supporting Classes

Views are objects that handle drawing and respond to events. The `<view>` class is a subclass of both the `<event-handler>` and `<frame>` classes. As a subclass of `<event-handler>`, a view can respond to events, such as a key-down or mouse-down event. For information about event handling, see "About Event Handling" on page 242. As a subclass of `<frame>`, the view has an extent that defines its size and its location in relation to its superview, which the Framework translates into a position withing a GrafPort. For more information about frames, see "About Frames" on page 310.

Note
The Framework provides several kinds of views to implement dialogs. These classes represent controls, groupings of controls (called clusters), static text, and icons. These classes are discussed in the chapter "Dialogs and Controls" on page 419. ◆

Figure 14-1 shows the subclasses of the `<view>` class.

Figure 14-1 View classes



Objects created from the classes in Figure 14-1 represent the following kinds of views:

`<cluster>`	Represents a group of views associated with a control. For information about the `<cluster>` class, see page 437.
`<control>`	Represents a control. For information about the `<control>` class, see page 440.
`<grid-view>`	Represents a view that is divided into cells. For information about the `<grid-view>` class, see page 391.
`<list-view>`	Represents a view that is a single column of cells. For information about the `<list-view>` class, see page 394.
`<text-grid-view>`	Represents a grid view that handles editable text. For information about the `<text-grid-view>` class, see page 395.
`<grow-icon>`	Represents a size box in a window. For information about the `<grow-icon>` class, see page 414.

<scroller>	Represents an area that can be larger than the area of its window, and thus represents an area that is scrollable. For information about the <scroller> class, see page 471.
<simple-icon>	Represents an icon. For information about the <simple-icon> class, see page 438.
<static-text>	Represents unchangeable text, such as text found in dialog boxes. For information about the <static-text> class, see page 439.
<text-view>	Represents a view that supports TextEdit operations. For information about the <text-view> class, see page 375.
<edit-text>	Represents a view that allows text to be changed. For information about the <edit-text> class, see page 378.
<number-text>	Represents editable text that is restricted to the digits 0 through 9. For information about the <number-text> class, see page 379.

View Hierarchies

Views are organized into hierarchies. A view hierarchy establishes the relationship between the views within it and defines the order in which events can be handled. When you create a view, you can specify its subviews. You can also add subviews to a view after it is created.

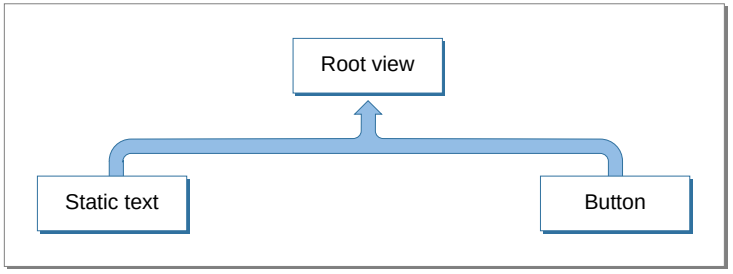
When you create a window, the Framework creates a view that is associated with the window. This view is the window's **root view**. Other views within the window must descend directly or indirectly from the root view.

Typically, you create a view that displays your content, called a **content view**, as a direct or indirect subview of the root view. If the window does not allow scrolling, you can make the view a subview of the root view directly. Consider the dialog box in Figure 14-2.

Figure 14-2 Views for a simple dialog

The view hierarchy consists of the root view and two subviews, one that contains static text and one that represents a button. Figure 14-3 shows the view hierarchy.

Figure 14-3 The view hierarchy for a simple dialog



Both the static text and button views represent the content of the dialog and are direct subviews of the root view. For information about dialogs, see the chapter "Dialogs and Controls" on page 419.

Figure 14-4 shows views that support scrolling in a window.

Figure 14-4 Views that support scrolling

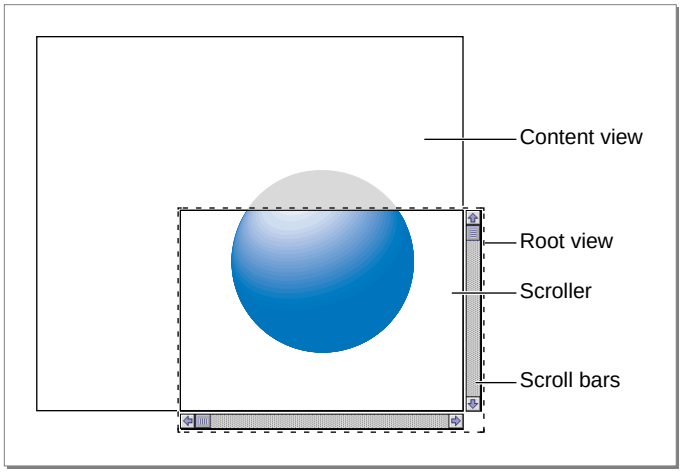
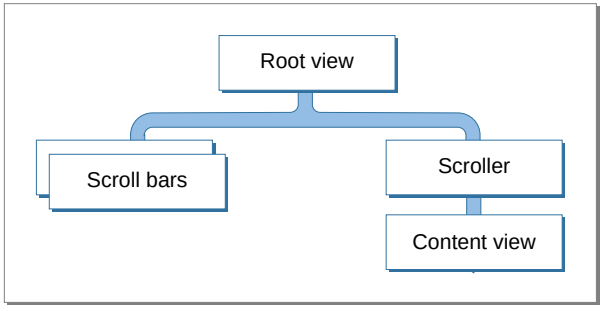


Figure 14-5 shows the view hierarchy for the scrolling window.

Figure 14-5 The view hierarchy for a scrolling window



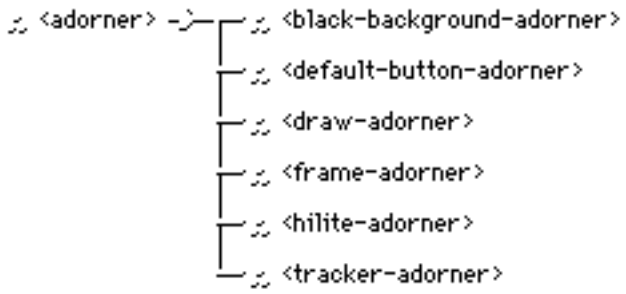
Adorners

Adorners represent reusable modifications to views, which are drawn when the view itself is drawn. For example, the Framework provides an adorner,

called a frame adorner, that draws an outline just inside the border of a view. You can add a frame adorner to any view that needs to be outlined without having to create a subclass of the view.

Figure 14-6 shows the subclasses of the `<adorner>` class.

Figure 14-6 Adorner classes



Objects created from the classes in Figure 14-1 represent the following kinds of views:

`<black-background-adorner>`
: Provides a a black background behind a view. For information about the `<black-background-adorner>` class, see “Adorner Classes” on page 360.

`<default-button-adorner>`
: Provides a border associated with a default button control. For information about the `<default-button-adorner>` class, see “The <default-button-adorner> Class” on page 455.

`<draw-adorner>`
: Specifies drawing the view itself. For information about the `<draw-adorner>` class, see “Drawing in Views” on page 342.

`<frame-adorner>`
: Provides a frame that outlines a view. For information about the `<frame-adorner>` class, see “Adorner Classes” on page 360.

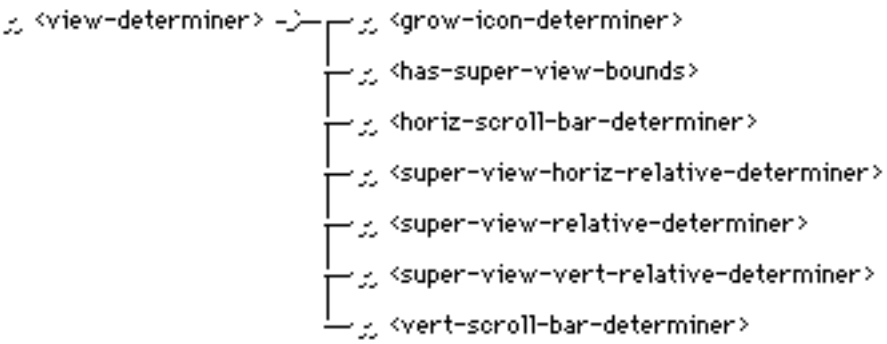
<hilite-adorner>	Specifies highlighting for the view itself. For information about the <hilite-adorner> class, see "Drawing in Views" on page 342.
<tracker-adorner>	An internal adorner used for mouse tracking.

View Determiners

View determiners represent the rules for changing a view when its superview changes. The Framework provides view determiner classes that represent common rules, such as keeping view within a superview's bounds or maintaining a view's size in proportion to its superview's size. The Framework also provides view determiner classes that handle resizing of scroll bars and moving a window's grow box.

Figure 14-7 shows the subclasses of the <view-determiner> class.

Figure 14-7 View determiner classes



Objects created from the classes in Figure 14-1 represent the following kinds of views:

<grow-icon-determiner>	Represents the rules for moving a <grow-icon> object when its superview changes. For information about the <grow-icon-determiner> class, seepage 414.

`<has-super-view-bounds>` Represents rules that specify using the superview's bounds. For information about the `<has-super-view-bounds>` class, see "View Determiner Classes" on page 362.

`<horiz-scroll-bar-determiner>`
Represents the rules for moving and resizing a horizontal scroll bar when its superview changes. For information about the `<horiz-scroll-bar-determiner>` class, see "View Determiner Classes" on page 362.

`<super-view-horiz-relative-determiner>`
Represents the rules that specify proportional changes to a view when its superview changes horizontally. For information about the `<super-view-horiz-relative-determiner>` class, see "View Determiner Classes" on page 362.

`<super-view-relative-determiner>`
Represents the rules that specify proportional changes to a view when its superview changes. For information about the `<super-view-relative-determiner>` class, see "View Determiner Classes" on page 362.

`<super-view-vert-relative-determiner>`
Represents the rules that specify proportional changes to a view when its superview changes vertically. For information about the `<super-view-vert-relative-determiner>` class, see "View Determiner Classes" on page 362.

`<vert-scroll-bar-determiner>`
Represents the rules for moving and resizing a vertical scroll bar when its superview changes. For information about the `<vert-scroll-bar-determiner>` class, see "View Determiner Classes" on page 362.

Drawing in Views

The Framework draws the contents of all views when an update event occurs. The `draw` method associated with the view defines its contents. For example,

the `draw` method could contain code to draw a series of shapes, such as lines or rectangles, pictures, or any other visual content.

The Framework only draws the visible part of each view that has been invalidated, which meaning the part that is in the window's update region. The Framework provides methods that allow you to invalidate or validate all or part of a view.

Drawing starts with the root view. After its contents are drawn, the contents of its subviews are drawn. If a subview also has subviews, they are also drawn before drawing the next subview of the root, and so on. Thus, drawing of subviews proceeds until the deepest subview of a view has been drawn and all subviews at the same level in a branch of a hierarchy are drawn before drawing for the next branch starts.

For each view, adorners control the order in which the adorner and the view itself is drawn. The Framework draws adorners in the order they occur within the view's list of adorners. The view always contains at least two adorners, a draw adorner, which is a `<draw-adorner>` object and a highlight adorner, which is a `<hilite-adorner>` object. The draw adorner causes the view's draw method to execute, thus drawing the contents of the view. The hilite adorner causes the view to be highlighted (see the following section for more information).

Note
The Framework provides ways for you to draw without waiting for an update event by calling `redraw` or `redraw-all`. ◆

Highlighting

The Framework changes the highlighting for views and adorners after they are drawn by calling the `hilite` method of the view or adorner. If you do not define a `hilite` method, nothing happens when the Framework attempts to highlight the view or adorner.

Higlighting for views is performed in the reverse order of drawing. Thus, highlighting for a subview is performed before highlighting for its superview. In other words, drawing occurs on the way down the view hierarchy; highlighting occurs on the way back up the hierarchy.

The order in which a view and its adorners are highlighted depend on the specific transition between highlight states (described below) and the order

that adorners are placed in the view's list of adorners. The Framework either highlights the adorners in a forward order or in reverse order as they are found in the list. When the highlight adorner is found, the view itself is highlighted.

The order of iteration, depends on the current highlight state for a view and the new state. The highlight states are either on, off, or dim. Table 14-1 shows the state transitions and the iteration order.

Table 14-1 Highlight state transitions and iteration orders

Current State:	New State: Off	Dim	On
Off	—	Forward	Forward
Dim	Reverse	—	Forward
On	Reverse	Reverse	—

Cursor Manipulation

The Framework provides a mechanism that allows each view to set the kind of cursor. The kind of cursor can also be constrained while in a view by specifying a cursor region that has less area than the view itself.

Initially, the cursor region is defined as the sleep area, which is the combined area of all screens less the area of active windows. The Framework restricts the cursor region of a view within a window to be the area of the view less the area of its subviews. This has the effect of allowing the cursor to be controlled by the deepest subview over which the cursor is positioned.

The normal event-handling protocol determines which view will set the cursor. The Framework updates the mouse position in response to mouse-moved events. When a mouse-moved event occurs, the behaviors of the target event handler have the opportunity to set the cursor, followed by the event handler (typically a view) itself. Once the cursor is set, there is no need to pass the event up the target chain. If the cursor is not set, the event is passed up the chain and other event handlers and their behaviors can set the cursor. If the event reaches

the main handler, the cursor is changed to an arrow cursor and the sleep region is reset to the area for all screens less the area of active windows.

The Framework provides the following cursors, as shown in Table 14-2:

Table 14-2 Cursors

Cursor	Description
`$arrow-cursor`	Arrow cursor
`$iBeamCursor`	I-beam cursor

Most of these cursors are defined as resources in the Framework's library. You can set the cursor in a view attached to a behavior or by the view itself. By default, the cursor is an arrow cursor.

Using View Objects

The following sections show some of the common tasks for which you need views, adorners, and view determiners.

Defining a View Class

You often define a view class to add slots that associate the view with other objects. Listing 14-1 shows how to define a class that associates a view with a document.

Listing 14-1 Defining a view class

```
define class <my-view> (<view>)
  slot the-document :: <my-document>,
    init-keyword: document:;
end class;
```

You can then create objects of the `<my-view>` class after you have a document object. For example, the following example shows one way to create the view:

```
let view = make(<my-view>,
                  location: point(0, 0),
                  extent:   point(1000, 1000),
                  document: a-document);
```

Note
Always define an extent for your view; otherwise its size is `$zero-point`, which will never be visible. ◆

Setting Up Views

You typically set up views when you create a window. If the window is associated with a document, the window and its views are created in the document's `make-windows` method.

After you define a view class, you can create an object from it and add it as a subview to another view. If you want to add the view to a window, you add the view as a subview of the root view.

If you want to associate the view with a new window, you can follow this procedure:

1. Create your view.
2. Create your window.
3. Add your view as a subview of the window's root view.

Listing 14-2 shows an example.

Listing 14-2 Setting up view for a window

```
let my-content = make(<view>,
                          extent: point(350, 150));

let the-window = make(<window>,
                      location: point(50, 50),
                      extent: view.extent + $scroll-bar-size,
```

```
                        has-go-away: #t,
                        moveable-modal: #t,
                        title: "Untitled",
                        target-view: my-content);

add-sub-view(the-window.root-view, my-content);
```

If you want to add a view to a new scrolling window, call `make-scrolling-window` as follows:

```
let window = make-scrolling-window(list(my-content),
                    location: point(50, 50),
                    extent: point(300, 200),
                    title: get-string($hello-window-title-id),
                    closeable: #t,
                    zoomable: #t,
                    resizable: #t,
                    target-view: my-content);
```

The `make-scrolling-window` method adds each view in a list of views to the root view.

If you want to add a view to an existing window, you must be able to access the window object. For example, if you know any view in the window, you call `get-window` and specify the view as the parameter. For more information about windows, see "About Windows" on page 401.

Setting the Target View

When you create a window, the target view for event handling is the root view of the window. Typically you need to specify another view such as your content or a text exit field as the target. When you create the window, you can specify the `target-view:` keyword to set the target view. When the window becomes active, the target becomes the specified view and the target chain includes all views from the target to the window's root view. After the window has been created, you can call `set-target` to change the target view.

Drawing in Views

The `draw` method specifies how to draw the content of the view. Listing 14-3 shows a `draw` method that draws the contents of a document's data as rectangles.

Listing 14-3 Defining a `draw` method

```
define method draw (view :: <my-view>,
                    draw-region :: <region>) => ()
  ignore(draw-region);
  for (a_rect in view.the-document.data)
    frame-region(a_rect);
  end for;
end method;
```

Setting Up a View for a Graphics World

A buffered view is a view that has a `<qd-graphics-buffer>` object, called a graphics buffer, as one of its slots. The graphics buffer is associated with a graphics world, thus you can store data in the view itself, manipulate the data, and draw it onscreen. Drawing from a buffered view is described in the next section.

Listing 14-4 shows the definition for a buffered view class.

Listing 14-4 Defining a view class with a graphics buffer

```
define class <buffered-view> (<view>)
    slot buffer :: <qd-graphics-buffer>,
        init-keyword: buffer:;
end class;
```

When you use a graphics buffer, you are responsible for creating it, initializing it, and disposing of it when it is no longer needed. Listing 14-4 shows an example of creating a graphics buffer and the view that uses it.

Listing 14-5 Creating the buffer and view

```
let buffer = make(<qd-graphics-buffer>,
                  extent: point(width-in-pix, height-in-pix),
                  depth: depth-in-pix);

clear-buffer(buffer); // erase contents

let buffered-view = make (<buffered-view>, buffer: buffer)
```

To dispose of the graphics buffer, call dispose, as in the following example:

```
dispose(buffer);
```

Note
Although the Dylan language will dispose of an object when it is no longer used, you may wish to dispose of large objects yourself earlier to guarantee that other large objects will not be created before memory for unneeded ones is released. ◆

Drawing With Buffered Views

Drawing into a buffered view requires you to focus the view, which sets up the port associated with the graphics buffer, and do what ever drawing actions you wish. The `with-focused-view` macro focuses the view in the port and restores the focus after drawing is complete, Listing 14-6 shows an example of using the macro.

Listing 14-6 Drawing into an offscreen view buffer

```
with-focused-view (buffered-view)
    // your QuickDraw drawing calls go here
end;
```

Your `draw` method should take the contents of the buffer and render it onscreen. The Framework provides the `draw-buffer` method to handle this task, as shown in Listing 14-7.

Listing 14-7 Drawing the contents of a buffer onscreen

```
define method draw (view :: <buffered-view>,
                        draw-region :: <region>) => ()
    draw-buffer(view.buffer, draw-region, draw-region);
end method;
```

Note
The `draw-buffer` method is equivalent to `CopyBits` in the Macintosh Toolbox. ◆

Setting Up Adorners

Adorners are reusable objects that can be attached to a view. When the view is drawn, the view's adorners are also drawn. To set up an adorner, you must follow these steps:

1. Define a subclass of the adorner class.
2. Define an `adorner-draw` method that specifies the drawing actions for the adorner.
3. Optionally, define one or more `adorner-hilite` methods to specify how the adorner is to be highlighted for each state transition.
4. Add the adorner to any views.

Listing 14-8 shows the definition of an adorner subclass that draws a border around the view to which it is attached so that the adorner's view is clearly visible within the view's superview. The area outside the view but inside its superview is drawn in the color specified by the adorner's `color` slot.

Listing 14-8 Defining an adorner subclass

```
define class <paint-visible-edges-adorner> (<adorner>)
    slot color :: <rgb-color>,
        required-init-keyword: color:;
end class;
```

Listing 14-9 shows the `adorner-draw` method that specifies the drawing actions.

Listing 14-9 Defining an `adorner-draw` method

```
define method adorner-draw
                    (adorner :: <paint-visible-edges-adorner>,
                     view :: <view>,
                     draw-region :: <region>) => ()
    let region = clone(draw-region);
    for (sub-view in view.sub-views)
        let sub-view-bounds = sub-view.super-bounds;
        subtract-from-region(region, sub-view-bounds);
        dispose(sub-view-bounds);
    end for;

    set-fore-color(adorner.color);
    paint-region(region);
    dispose(region);
    set-fore-color(view.fore-color);
end method;
```

Listing 14-10 shows the `adorner-hilite` methods for the adorner.

Listing 14-10 Defining `adorner-hilite` methods

```
define method adorner-hilite (adorner :: <hilite-adorner>,
                              view :: <view>,
                              hilite-region :: <region>,
                              old-hilite :: <symbol>,
                              new-hilite :: <symbol>) => ()
```

```
  ignore(adorner);

  if (old-hilite ~= new-hilite)
    hilite(view, hilite-region, old-hilite, new-hilite);
  end if;
end method;
```

You can add and remove an adorner as needed. Listing 14-11 shows an example of creating and adding the adorner to the scroller view of a scrolling window.

Listing 14-11 Adding an adorner to a view

```
let scroller = get-scroller(view);
add-adorner(scroller, make(<paint-visible-edges-adorner>,
                color: rgb-color(40000, 40000, 40000)));
```

You can control the order in which adorners are added to the view's list of adorner by using the `first:`, `before:`, and `after:` keywords. If you do not specify one of these keywords, the adorner is added last. For example, if you want the adorner to draw before the view is drawn, you could specify before: `$draw-adorner` when you call `add-adorner`. For information about the order in which a view and its adorners are drawn, see "Drawing in Views" on page 342.

Implementing a View Determiner

A view determiner specifies how a view should change in response to a change in a superview or subview. The Framework provides several view determiners for you. For a list, see "View Determiners" on page 341. For example, Listing 14-12 shows how to use several view determiners to implement scrolling in views. The `determiner:` keyword specifies the view determiner.

Listing 14-12 Setting up view determiners for scrolling

```
let scroller = make(<scroller>,
    identifier: #"scroller",
    extent: point (root.extent.h - $scroll-bar-size,
                   root.extent.v - $scroll-bar-size),
    determiner: make(<super-view-relative-determiner>));

let h-scroll = make(<ctl-mgr-scroll-bar>,
    identifier: #"h-bar",
    location: point(-1, root.extent.v - $scroll-bar-size),
    extent: point(root.extent.h - $scroll-bar-size + 2, 16),
    scroll-increment: 32,
    determiner: make(<horiz-scroll-bar-determiner>));

let v-scroll = make(<ctl-mgr-scroll-bar>,
    identifier: #"v-bar",
    location: point(root.extent.h - $scroll-bar-size, -1),
    extent: point(16, root.extent.v - $scroll-bar-size + 2),
    scroll-increment: 32,
    determiner: make(<vert-scroll-bar-determiner>));
```

To implement your own view determiner, you must define a subclass of the `<view-determiner>` class and define one or more methods whose view determiner parameter specialize on your subclass. These methods are defined as follows:

Function	Specialize to
`super-view-extent-changed`	Specify the actions to take when the extent of the view's superview changes.
`sub-view-extent-changed`	Specify the actions to take when the extent of a view's subview changes.
`super-view-location-changed`	Specify the actions to take when the location of the view's superview changes.
`sub-view-location-changed`	Specify the actions to take when the location of a view's subview changes.

Listing 14-13 shows the class definition of a subclass of the `<view-determiner>` class.

Listing 14-13 Defining a view determiner subclass

```
define class <center-in-super-determiner> (<view-determiner>)
end class;
```

Listing 14-14 defines a `super-view-extent-changed` method that specifies how to change the view's location when the superview's extent changes. In this example, the view is centered in the superview.

Listing 14-14 Defining a `super-view-extent-changed` method

```
define method super-view-extent-changed
                    (determiner :: <center-in-super-determiner>,
                                        view :: <view>,
                                        super-view :: <view>,
                                        old-extent :: <point>,
                                        new-extent :: <point>) => ()
    ignore(determiner, old-extent, new-extent);

    view.location := point(
        max(0, floor/((super-view.extent.h - view.extent.h), 2)),
        max(0, floor/((super-view.extent.v - view.extent.v), 2)));
end method;
```

After you define the class you can add your view determiner to a view by using the `determiner:` keyword as shown in Listing 14-12, or you can assign it to the view, as in the following statement:

```
view.determiner := make(<center-in-super-determiner>);
```

View Objects Reference

The following sections describe the classes that implement the Framework's view mechanism and identify the functions that you can use with them.

The <view> Class

The `<view>` class is a class that supports drawing and responds to events. You can create objects directly from a `<view>` class or you can create subclasses and create objects from the subclasses. For example, views that are not related to documents, such as those used in dialogs, typically do not require you to define a subclass. However, if you have a document that contains data that your view is responsible for drawing, the reference to the document is usually maintained as a slot in your view subclass.

Listing 14-15 shows the class definition for the `<view>` class.

Listing 14-15 The `<view>` class definition

```
define primary open class <view> (<event-handler>, <frame>)

  slot visible-bounds :: <rect>,
     init-function: curry(make, <rect>);

  slot sub-views :: <list>,
     init-value: #();

  slot adorner-list :: <list>,
     init-value: #(),
     init-keyword: adorners:;

  slot determiner-list :: <list>,
     init-value: #(),
     init-keyword: determiners:;

  slot active? :: <boolean>,
     init-value: #f; // this should only be set by activate(<view>)

  slot wants-to-be-target? :: <boolean>,
     init-value: #f;

  // possible hilites are #"off", #"on" or #"dim"
  slot inactive-hilite :: <symbol>,
     init-value: #"dim",
     init-keyword: inactive-hilite:;
```

```
slot active-hilite :: <symbol>,
   init-value: #"on",
   init-keyword: active-hilite:;

slot fore-color :: <rgb-color>,
   init-value: $black-color,
   init-keyword: fore-color:;

slot back-color :: <rgb-color>,
   init-value: $white-color,
   init-keyword: back-color:;

slot locked? :: <boolean>,
   init-value: #f,
   init-keyword: locked:;

// if transparent? is true, don't draw the back-color before drawing
slot transparent? :: <boolean>,
   init-value: #f,
   init-keyword: transparent:;

slot text-style :: <text-style>,
   init-value: $Geneva12,
   init-keyword: text-style:;

slot inset :: <rect>,
   init-function: curry(make, <rect>),
   init-keyword: inset:;

slot invalidate-all-on-resize? :: <boolean>,
   init-value: #f,
   init-keyword: invalidate-all-on-resize:;

// private
// use generic functions super-view, visible?
// and their setters to access  these slots.

slot %super-view,
   init-value: #f,
```

```
     init-keyword: super-view:;

  slot %visible? :: <boolean>,
     init-value: #t,
     init-keyword: visible:;

  slot cached-local-bounds :: <rect>;

   slot graphics-port,
    init-value: #f,
     init-keyword: graphics-port:;

  slot view-to-port-offset :: <point>,
    init-value: $zero-point;
  slot qd-origin :: <point>,
     init-value: $zero-point;

  slot clip-region :: <qd-region>,
     init-function: curry(make, <qd-region>);

end class;
```

Slot descriptions

`visible-bounds` The location and extent of the visible part of the view. You can determine the visible bounds by calling `visible-bounds`. If you only need to know if the view is visible, call `visible?`. To set this slot, you must call `compute-visible-bounds`. Typically, you do not need to set the value of this slot; the Framework does it for you.

`sub-views` A list of views that are directly subordinate to this view in the view hierarchy. Use the `sub-views:` keyword to specify the list; otherwise, the list is empty. The Framework provides functions that allow you to manipulate this list. They are `add-sub-view`, `find-sub-view`, and `remove-sub-view`.

`adorner-list` A list of adorners that draw before the view is drawn. By default, the Framework adds the drawing adorner and the highlight adorner (see "Drawing in Views" on page 342) to the list for you. To add an adorner to the list, call `add-adorner`. To remove an adorner, call `remove-adorner`.

`determiner-list` A list of view determiners that specify how the location and size of the view changes in relation to a change in the extent of its superview or subviews. Use the `determiners:` keyword to specify the contents of the list; otherwise, the list contains `$default-determiner` to specify that no change occurs. You can use the getter and setter to access this slot.

`active?` Specifies whether or not the view is active. When a view's window is activated, the window's root view and all views in the hierarchy become active. Call `active?` to determine if the view is active. Call `activate` to this slot.

`wants-to-be-target?` Specifies whether or not the view can become the target if the user tabs into the view. By default, the value is false (`#f`) for non-text views. You can use the getter and setter to access this slot. For more information about tabbing behavior, see "Adding a Tabbing Behavior" on page 432.

`inactive-hilite` The highlighting for the view when it is inactive. By default, it is `#"dim"`. Possible choices are `#"on"`, `#"off"`, or `#"dim"`. You can use the getter and setter to access this slot. You can call `current-hilite` to determine or change the current highlighting for the view.

`active-hilite` The highlighting for the view when it is active. By default, it is `#"on"`. Possible choices are `#"on"`, `#"off"`, or `#"dim"`. You can use the getter and setter to access this slot. You can call `current-hilite` to determine or change the current highlighting for the view.

`fore-color` The foreground color for the view. Use the `fore-color:` keyword to specify color; otherwise, the color is black. You can use the getter and setter to access this slot.

`back-color` The foreground color for the view. Use the `back-color:` keyword to specify color; otherwise, the color is white. You can use the getter and setter to access this slot.

`locked?` Whether or not the view is locked.

`transparent?` Specifies whether or not content drawn in the view allows overlapping content to be seen (`#t`) or whether the content being drawn obscures overlapping content (`#f`). Use the `transparent:` keyword to specify whether you want transparent drawing; otherwise, the value of this slot is

	false (`#f`), meaning that drawing in a view obscures what has been drawn already. You can use the getter and setter to access this slot.
`text-style`	The style for text displayed in the view. Use the `text-style:` keyword to specify style; otherwise, the text style is 12-point Geneva. You can use the getter and setter to access this slot.
`inset`	A rectangle that defines the inset for the view. Use the `inset:` keyword to specify the inset; otherwise, the top, left, bottom, and right coordinates for the rectangle are 0, meaning no inset. Call `inset-bounds` to set the bounds. You can use the getter to access this slot.
`invalidate-all-on-resize?`	Specifies whether or not the entire view is invalidated when the view's size changes (`#t`), or not, (`#f`). Use the `invalidate-all-on-resize:` keyword to specify true (`#t`) if you want to redraw the entire view; otherwise, the value of this slot is false (`#f`), meaning that only the invalidated area is redrawn whenever the view's size changes. You can use the getter and setter to access this slot.
`%super-view`	The view's parent view in the view hierarchy, or false (`#f`) if the view is at the root of its hierarchy. Use the `super-view:` keyword to specify the superview; otherwise, it is true (`#f`). Use the `super-view` and `super-view-setter` functions to access this slot. The `%super-view` slot is not exported.
`%visible?`	Specifies whether any part of the view is visible onscreen. Use the `visible:` keyword to specify whether the view is visible; otherwise, it is true (`#t`). Use the `visible` and `visible-setter` functions to access this slot. The `%visible?` slot is not exported.
`cached-local-bounds`	Internal. Call `local-bounds` or `super-bounds` to determine the bounds of a view.
`graphics-port`	Internal. The port through which an image can be rendered.

`view-to-port-offset`
: The translation between local coordinates and root coordinates.

`clip-region`
: The clipping region that specifies the visible part of the view .

Initialization

The `initialize` method for `<view>` objects allows you to specify subviews of a view. The `initialize` method's parameters are defined as follows:

```
define method initialize (view :: <view>,
                    #key        sub-views: sub-view-list ( #() ),
                                dependents: dependent-list ) => ()
```

`view`
: The view.

`sub-view-list`
: The list of subviews directly subordinate to the view in the view hierarchy.

`dependent-list`
: Internal. Used only by the streaming mechanism.

In addition, the `initialize` method sets up the view's bounds from its location and extent, creates a graphics cache for the graphics system specified in `*qd-graphics-system*`, and adds the `$draw-adorner` and `$hilite-adorner` to the view's list of adorners.

Adorner Classes

The `<adorner>` class is a class that specifies items to draw along with the view itself. You do not create objects directly from the `<adorner>` class, rather you create subclasses and create objects from the subclasses. You may also create objects from subclasses provided by the Framework. For example, if you want to outline your view, you can create a `<frame-adorner>` object and add it to the list of adorners for the view. The frame is drawn before the view. For an example of setting up an adorner and adding it to a view, see "Setting Up Adorners" on page 350.

Listing 14-16 shows the class definition for the `<adorner>` class and subclasses provided by the Framework. The subclasses are `<frame-adorner>`, `<black-background-adorner>`, `<draw-adorner>`, and `<hilite-adorner>`.

Listing 14-16 Adorner class definitions

```
define primary open class <adorner> (<object>)
  slot identifier :: <symbol>,
     init-value: $null-identifier,
     init-keyword: identifier:;
end class;

define primary sealed class <draw-adorner> (<adorner>)
  inherited slot identifier,
     init-value: #"draw";
end class;

define primary sealed class <hilite-adorner> (<adorner>)
 inherited slot identifier,
     init-value: #"hilite";
end class;


define primary sealed class <frame-adorner> (<adorner>)
 inherited slot identifier,
     init-value: #"frame";
end class;

define primary sealed class <black-background-adorner> (<adorner>)
  inherited slot identifier,
     init-value: #"black-background";
end class;

define primary sealed class <default-button-adorner> (<adorner>)
  inherited slot identifier,
     init-value: #"default-button";
end class;
```

Slot descriptions

`identifier` Identifies the adorner. Set this slot if you need to find or remove the adorner or add it relative to some other adorner. Use the `identifier:` keyword to specify the identifier; otherwise, the slot is set to `$null-identifier`.

Initialization

No `initialize` method is defined for adorner objects.

View Determiner Classes

The `<view-determiner>` class is a class that determines how the location and size of a view changes in relation to a change in the extent of its superview or subviews. You do not create objects directly from the `<view-determiner>` class, rather you create subclasses and create objects from the subclasses. You may also create objects from subclasses provided by the Framework.

For example, if you want your subview to change in proportion to its superview, you can create a `<super-view-relative-determiner>` object and add it to the list of determiners for the view. (See "Implementing a View Determiner" on page 352 for an example.) The view is then resized in proportion to its superview whenever the superview changes.

Listing 14-17 shows the class definition for the `<view-determiner>` class and its subclasses.

Listing 14-17 View determiner class definitions

```
define class <view-determiner> (<object>)
    slot identifier :: <symbol>,
        init-value: $null-identifier,
        init-keyword: identifier:;
end class;

define primary sealed class <has-super-view-bounds> (<view-determiner>)
end class;


define primary sealed class <super-view-relative-determiner>
                            (<view-determiner>)
end class;

define primary sealed class <horiz-size-super-view-determiner>
                            (<view-determiner>)
end class;
```

```
define primary sealed class <super-view-horiz-relative-determiner>
                            (<view-determiner>)
end class;

define primary sealed class <vert-size-super-view-determiner>
                            (<view-determiner>)
end class;

define primary sealed class <super-view-vert-relative-determiner>
                            (<view-determiner>)
end class;

define primary sealed class <horiz-scroll-bar-determiner>
                            (<view-determiner>)
end class;

define primary sealed class <vert-scroll-bar-determiner>
                            (<view-determiner>)
end class;
```

Slot descriptions

`identifier` Identifies the view determiner. Set this slot if you need to specialize a method on a specific view determiner. Use the `identifier:` keyword to specify the identifier; otherwise, the slot is set to `$null-identifier`.

Initialization

No `initialize` method is defined for view determiner objects.

View-Related Functions

Table 14-3 shows the functions you can use with views.

Table 14-3 Functions for views

Function	Purpose	Call	Sp.
`get-window`	Determine the window associated with a view.	√	
`local-to-root`	Convert a point or region from local coordinates to root coordinates	√	
`root-to-local`	Convert a point or region from root coordinates to the local coordinates of the view.	√	
`open`	Open the view.	√	
`close`	Close the view.	√	
`invalidate-location -changed`	Invalidate part of screen because the view's location changed.		
`invalidate-extent -changed`	Invalidate part of screen because the view's extent changed. Can only be used if new extent is larger than the old extent.		
`invalidate-extent -changed-all`	Invalidate part of screen because the view's extent changed.	√	
`add-sub-view`	Add a subview to the specified view.	√	
`remove-sub-view`	Remove a subview from a view.	√	
`find-sub-view`	Return the first subview that matches the specified identifier in the view's subviews.	√	
`do-each-sub-view`	Execute the specified function on each subview.	√	
`do-view-and-subviews`	Execute the specified function on a view and each of its subviews.	√	

Table 14-3 Functions for views

Function	Purpose	Call	Sp.
`is-sub-view-of?`	Returns whether a view is a subview of a specified view.	√	
`handle-draw`	Internal.		
`draw`	Specify the actions to take to draw the contents of the view.		√
`draw-adorners`	Internal.		
`hilite-adorners`	Internal.		
`hilite`	Specify the actions to take to highlight the view.		√
`current-hilite`	Get or set the current highlight state.	√	
`handle-set-cursor`	Internal.		
`behavior-set -cursor`	Set the cursor for a view from a behavior.		√
`do-set-cursor`	Set the cursor for a view.		√
`get-default -cursor-region`	Internal.		
`get-graphics -cache`	Internal.		
`add-graphics -cache`	Internal.		
`remove-graphics -cache`	Internal.		
`notifiy-super-view -location-changed`	Internal.		
`notify-sub-view-locatio n-changed`	Internal.		
`notifiy-super-view-exte nt-changed`	Internal.		

Table 14-3 Functions for views

Function	Purpose	Call	Sp.
`notify-sub-view -extent-changed`	Internal.		
`location-changed`	Internal. (Call location-setter.)		
`extent-changed`	Internal. (Call extent-setter.)		
`get-mouse`	Retrieve the point under the mouse in the local coordinates of the specified view.		
`clip-to-super -view`	Internal.		
`prepare-to-draw -region`	Internal.		
`prepare-to-draw`	Internal.		
`done-drawing -region`	Internal.		
`done-drawing`	Internal.		
`invalidate`	Invalidate the specified part of a view.		
`invalidate-all`	Invalidate the visible part of a view.		
`validate`	Validate the specified part of a view		
`validate-all`	Validate the visible part of a view.	√	
`redraw`	Immediately redraw the specified part of a view.	√	
`redraw-all`	Immediately redraw the view.	√	
`compute-visible-bounds`	Internal. Use the `visible-bounds` slot.		
`activate`	Internal.		
`set-target -selection`	Scroll the view so it is visible in the window.	√	√
`adjust-extent`	Change the extent to allow for a change in the view's content.	√	√

Table 14-3 Functions for views

Function	Purpose	Call	Sp.
`selection-rect`			
`reveal-selection`	Specify how to increase a view's extent when the view can no longer accomdate what is being drawn.	√	√
`invalidate-cursor-region`	Reset the sleep region to be an empty region.	√	

Functions for Graphics Support

Table 14-4 shows the functions for graphics support.

Table 14-4 Functions for graphics support

Function	Purpose	Call	Sp.
`update-coordinates`	Internal.		
`enable-buffering`	Internal.		
`disable-buffering`	Internal.		
`pre-draw`	Internal.		
`post-draw`	Internal.		
`add-view-to-port`	Internal.		
`remove-view-from-port`	Internal.		
`make-graphics-cache`	Internal.		
`invalidate-caches`	Internal.		
`update-origin`	Internal.		
`compute-clip`	Internal.		
`get-clip-region`	Retrieve the view's clip region.	√	
`view-to-port`	Internal.		

Table 14-4 Functions for graphics support

Function	Purpose	Call	Sp.
`quickdraw-port`	Internal.		
`do-invalidation`	Internal.		
`do-validation`	Internal.		

Functions for Adorners

Table 14-5 shows the functions you can use with adorners.

Table 14-5 Functions for adorners

Function	Purpose	Call	Sp.
`adorner-draw`	Specify how to draw the adorner.		√
`adorner-hilite`	Specify how to highlight the adorner.		√
`add-adorner`	Add an adorner to a view's list of adorners.	√	
`remove-adorner`	Remove an adorner to a view's list of adorners	√	
`find-adorner`	Retrieve an adorner to a view's list of adorners	√	

Functions for View Determiners

Table 14-3 shows the functions you can use with view determiners.

Table 14-6 Functions for view determiners

Function	Purpose	Call	Sp.
`super-view-extent -changed`	Specify the actions to take when the extent of the view's superview changes.		√
`sub-view-extent -changed`	Specify the actions to take when the extent of a view's subview changes.		√
`super-view-location -changed`	Specify the actions to take when the location of the view's superview changes.		√
`sub-view-location -changed`	Specify the actions to take when the location of a view's subview changes.		√

CHAPTER 15

Text Views

Contents

About Text View Classes 373
Using Text View Objects 374
Creating a Text View 374
Getting and Setting Font Styles 374
Growing a Text View 375
Text View Objects Reference 375
The <text-view> Class 375
The <edit-text> Class 378
The <number-text> Class 379
Text View Functions 381

Text views are views that use the Macintosh TextEdit facility to display and manipulate text. TextEdit, and thus the Framework, provide the following capabilities:

- entering new text
- deleting characters that are backspaced over
- translating mouse activity into text selection
- replacing or deleting text

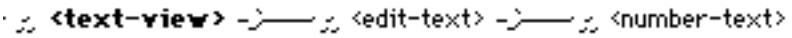
The Framework also provides undoable and redoable commands, supports cut, copy, and paste of text to the clipboard, and scripting.

About Text View Classes

The Framework provides a general purpose text view class that displays and manipulates text. It also provides two subclasses, one that represents "preselected" text, which is text placed in the view and highlighted, and another that restricts the text to numbers. Objects created from these subclasses are often used in dialog boxes where it is useful to provide highlighted fields whose contents may either be confirmed or changed and, for numeric fields, to restrict values to digits.

Figure 15-1 shows the subclasses of the `<text-view>` class.

Figure 15-1 Text View classes



Objects created from the classes in Figure 15-1 represent the following kinds of views:

`<text-view>`	Represents a view that supports TextEdit operations.
`<edit-text>`	Represents a view that sets up and highlights the text in a view. It is primarily intended for use in dialog boxes.

`<number-text>`	Represents editable text that is restricted to the digits 0 through 9 and associated punctuation.

Using Text View Objects

When you create a text view, the Framework does most of the work of translating the user's actions into TextEdit-supported operations for you. You typically only need to create the text view and provide menu items that allow the user to customize text in the view. For information about setting up menus, see "Setting Up Menus" on page 288.

The following sections show how to create a text view and how to get and set the font.

Creating a Text View

Listing 15-4 shows code that creates a text view. The view uses styles and the style of the initial text is 12-point Geneva regular.

Listing 15-1 Creating a `<text-view>` object

```
let view = make(<text-view>,
                extent: point(700, 300),
                use-styles: #t,
                text-style: $geneva12);
```

Getting and Setting Font Styles

Listing 15-4 shows code that determines the font style of the currently selected text in a view and sets it to the same style as the menu item's style.

Listing 15-2 Determining and setting the style of selected text.

```
let old-style = get-selection-font-style(view);
let style = event.menu-item.style;
set-selection-font-style(view, style, toggle: #t)
```

The `toggle: #t` keyword specifies that, if the selection is already set, it is unset.

Note
If the `toggle:` keyword is false (`#f`), the style would remain set if it is already set. ◆

Growing a Text View

The Framework calls `reveal-selection` method to allow characters to be displayed when entering a line of text causes the characters to exceed the height of the view. You must provide a `reveal-selection` method specialized on the the view parameter that allows characters to be displayed; otherwise, they will not be visible as they are being typed.

Text View Objects Reference

The following sections describe the classes that implement the Framework's text view mechanism and identify the functions that you can use with them.

The <text-view> Class

The `<text-view>` class is a class that provides Macintosh Text Edit capabilities. You typically create `<text-view>` objects although you can also create subclasses of the `<text-view>` class if you need to specialize parameters on the class or provide additional slots.

Listing 15-3 shows the class definition for the `<text-view>` class.

Listing 15-3 The <text-view> class definition

```
define primary open class <text-view> (<view>)
  slot te-handle :: <TEHandle>,
     init-value: as(<TEHandle>, 0);
  slot typing-command,
     init-value: #f;
   slot word-wrap? :: <boolean>,
     init-value: #t,
     init-keyword: word-wrap:;
  slot view-only? :: <boolean>,
     init-value: #f,
     init-keyword: view-only:;
  slot use-styles? :: <boolean>,
     init-value: #f,
     init-keyword: use-styles:;
  slot max-text-length :: <integer>,
     init-value: #x7FFF,
     init-keyword: max-text-length:;

  slot record-editing? :: <boolean>,
     init-value: #t;

  inherited slot wants-to-be-target?, init-value: #t;

  inherited slot inset,
     init-function: curry(rect, 2, 2, 2, 2);

end class;
```

Slot descriptions

`te-handle`	A handle to a Text Edit record. You can use the getter and setter to access this slot.
`typing-command`	A command that contains the current text.
`word-wrap?`	Specifies whether you want the text to wrap without breaking a word across lines (`#t`) or whether a word may start and end on different lines (`#f`). Use the `word-wrap:` keyword to specify the value; otherwise, the value is true

	(#t), meaning that words are not split between lines. You can use the getter to access this slot.
view-only?	Specifies whether the text can only be viewed(#t) or whether the text is also editable(#f). Use the view-only: keyword to specify the value; otherwise, the value is false(#f), meaning that the text in the view can be changed.
use-styles?	Specifies whether the text can have multiple styles (#t) or whether the text can have only a single style (#f). Use the use-styles: keyword to specify the value; otherwise, the value is false(#f), meaning that the text in the view is monostyled. You can use the getter and setter to access this slot.
max-text-length	The maximum length for text within the view. Use the max-text-length: keyword to specify the value; otherwise, the value is 32,767. You can use the getter and setter to access this slot.
record-editing?	Specifies whether adding or changing the text is recorded for Apple events(#t) or whether the text is not recorded(#f). By default, the value is true(#t), meaning that changes to the text are recorded. You can use the getter and setter to access this slot.
wants-to-be-target?	Specifies whether or not the view can become the target if the user tabs into the view. By default, the value is true (#t) for text views. You can use the getter and setter to access this slot. For more information about tabbing behavior, see "Adding a Tabbing Behavior" on page 432.
inset	A rectangle that defines the inset for the view. Use the inset: keyword to specify the inset; otherwise, the top, left, bottom, and right coordinates for the rectangle are 2, meaning a 2-pixel inset on each side. You can use the getter and setter to access this slot.

Initialization

The initialize method for <text-view> objects initializes a Macintosh TextEdit record and allows you to specify the initial text and style for text in the view. The initialize method's parameters are defined as follows:

```
define method   initialize  (view :: <text-view>,
                             #key text: text-string,
                             text-style: style) => ()
```

`view`	The view.
`text-string`	The text to associate with the view. Use the `text:` keyword to specify the text; otherwise the view contains no text initially.
`style`	The text style. Use the `text-style:` keyword to specify the style; otherwise the style is 12-point regular Chicago.

The <edit-text> Class

The `<edit-text>` class is a class that allows you to specify the text to be highlighted initially and can provide special character handling associated with dialogs. The special handling allows the tab and shift-tab characters to be interpreted as move to next or previous field, respectively, and the return and cancel characters to be invoke the okay and cancel buttons, respectively. You typically create `<edit-text>` objects for dialogs although you can also create subclasses of the `<edit-text>` class if you need to specialize parameters on the class or provide additional slots.

Listing 15-4 shows the class definition for the `<edit-text>` class.

Listing 15-4 The `<edit-text>` class definition

```
define primary open class <edit-text> (<text-view>)
    inherited slot record-editing?,
        init-value: #f;
    inherited slot word-wrap? :: <boolean>,
        init-value: #t,
end class;
```

Slot descriptions

`record-editing?`	Specifies whether adding or changing the text is recorded for Apple events`(#t)` or whether the text is not recorded`(#f)`. By default, the value is false `(#f)`, meaning that changes to the text are not recorded. You can use the getter and setter to access this slot.

`word-wrap?`	Specifies whether you want the text to wrap without breaking a word across lines (`#t`) or whether a word may start and end on different lines (`#f`). Use the `word-wrap:` keyword to specify the value; otherwise, the value is true (`#t`), meaning that words are not split between lines. You can use the getter to access this slot.

Initialization

The `initialize` method for `<edit-text>` objects allows you to specify the initial text and whether you want to filter characters used to control the operation of a dialog box. The `initialize` method's parameters are defined as follows:

```
define method initialize (view :: <edit-text>,
    #key    text: str,
            filter-control-chars: filter (#t)) => ()
```

`view`	The view.
`str`	The initial text for the view. The initial text is highlighted.
`filter`	Specifies whether to use the tab, shift-tab, escape, and return characters (`#t`) to control the operation of a dialog box, or to not filter these characters and treat them as text (`#f`). Use the `filter-control-chars:` keyword to specify a value; otherwise, the value is true (`#t`), meaning that the characters will be used to control the dialog box and not appear as part of the text.

The <number-text> Class

The `<number-text>` class is a class that specifies a range of values allowed as text in the view. You typically create `<number-text>` objects for dialogs although you can also create subclasses of the `<number-text>` class if you need to specialize parameters on the class or provide additional slots.

Listing 15-5 shows the class definition for the `<number-text>` class.

Listing 15-5 The `<number-text>` class definition

```
define primary open class <number-text> (<edit-text>)
  slot min-value :: <integer>,
     init-value: 0,
     init-keyword: min:;
  slot max-value :: <integer>,
     init-value: 1000000,
     init-keyword: max:;
end class;
```

Slot descriptions

`min-value`	The minimum value. Use the `min:` keyword to specify the value; otherwise, the slot is set to 0.
`max-value`	The maximum value. Use the `max:` keyword to specify the value; otherwise, the slot is set to 1000000.

Initialization

The `initialize` method for `<number-text>` objects allows you to specify the initial value, which is right-justfied in the view. The `initialize` method's parameters are defined as follows:

```
define method initialize (text :: <number-text>,
        #key current-value: val) => ()
```

`text`	The view.
`val`	The number to store as text. Use the `current-value:` keyword to specify the value; otherwise, the text representing the current value is blank.

Text View Functions

Table 15-1 shows functions that manipulate text views.

Table 15-1 Functions for text views

Function	Purpose	Call	Sp.
`adjust-te-rects`	Adjust the TextEdit record's text rectangle to match the size of the view.	√	
`justification`	Determine or specify the justification of the text.		
`make-te-handle`	Create a TextEdit record handle.		
`click-loop`	Specifies how to handle scrolling within a text view.		√
`get-text-handle`	Retrieve the handle containing the view's text.	√	
`get-text`	Retrieve text from a handle.	√	
`set-text`	Set text into a handle.	√	
`append-text`	Append text to a handle.	√	
`recalc-text`	Calculate the beginning of each line in the view.	√	
`adjust-extent`	Change the view's extent to accommodate the exact number of lines in the text.	√	
`text-length`	Determine the text length, in bytes.	√	
`get-selection`	Create a designator object that specifies the selection range.	√	
`set-selection`	Select text using a designator object.	√	
`set-selection-data`	Insert and select text from a `<data-item>` object.	√	
`delete-data`	Not implemented. Use `delete-selection`.		
`delete-selection`	Remove the selected text from the view.	√	

Table 15-1 Functions for text views

Function	Purpose	Call	Sp.
`get-selection-range`	Determine the start and stop characters in a selection.	√	
`set-selection-range`	Select the characters between the start and stop bytes.	√	
`get-selected-text`	Retrieve the selected text as a string of bytes.	√	
`set-selected-text`	Replace the selected text with the specified string.	√	
`get-data`	Retrieve text as a `<data-item>` object.	√	
`make-typing-command`	Internal.		
`done-typing`	Internal. (Notifies that something other than an alphanumeric key was pressed.)		
`can-receive-data?`	Specifies whether the view can accept the data which might be passed into it.		√

Table 15-2 shows functions that operate on text styles.

Table 15-2 Functions for text styles

Function	Purpose	Call	Sp.
`get-selection-fonts`	Retrieve the fonts in the selected text.	√	
`get-selection-font`	Retrieve the font of the selected text.	√	
`set-selection-font`	Set the font of the selected text.	√	
`get-selection-font-sizes`	Retrieve the font sizes in the selected text.	√	
`get-selection-font-size`	Retrieve the font size of the selected text.	√	

Table 15-2 Functions for text styles

Function	Purpose	Call	Sp.
`set-selection -font-size`	Set the font size of the selected text.	√	
`get-selection -font-styles`	Retrieve the character styles in the selected text.	√	
`get-selection -font-style`	Retrieve the character style of the selected text.	√	
`set-selection -font-style`	Set the style of characters in the selected text.	√	
`get-style -handle`	Retrieve the specified part of the style scrap handle for the view.	√	
`set-style -handle`	Specify the part of a style scrap handle to use for a view.	√	
`get-selection -font-style -handle`	Retrieve the style scrap handle for the view.	√	
`set-selection -font-style -handle`	Specify the style scrap handle to use for a view.	√	

CHAPTER 16

Grid and List Views

Contents

About Grid and List Views 387
- Grid and List View Classes 387
- Grid View Behaviors 388

Using Grid and List View Objects 388
- Setting Up Grid and List Views 389
- Drawing in Views 389
- Selecting Cells 390
- Highlighting Cells in a View 390
- Handling Variable Column Widths 390
- Handling Variable Row Heights 391

Grid and List View Objects Reference 391
- The <grid-view> Class 391
- The <text-grid-view> Class 394
- The <list-view> Class 394
- The <text-list-view> Class 395
- The <grid-select-behavior> Class 396
- The <grid-arrow-key-behavior> Class 396
- Grid and List View Related Functions 397

The Framework provides a grid view class that represents rows and columns of cells, such as those contained in a spreadsheet, and subclasses to handle lists (one-column grids) and text within grids or lists.

About Grid and List Views

The Framework provides a general purpose grid view class and subclasses that represent a single-column grids, called lists, and grids and lists that support text. You use grid view classes in the same way as you use other view classes, except that drawing and highlighting are specified on a cell-by-cell basis, rather than for the entire view.

Text grid and list views are not `<text-view>` objects, thus they do not inherit TextEdit capabilities directly. You are responsible for retrieving the text to be displayed or manipulated in the grid.

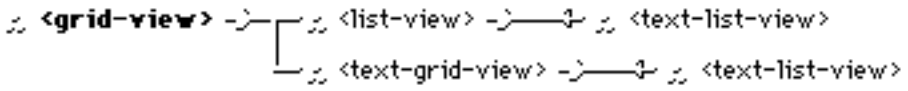
Because grid views are designed to be manipulated cell-by-cell, the Framework provides two behaviors that allow cells to be selected. The `<grid-select-behavior>` class supports cell selection using the mouse. The `<grid-arrow-key-behavior>` class handles cell selection when the user presses an arrow key.

The following sections introduce the grid and list view classes and their supporting behaviors.

Grid and List View Classes

Figure 16-1 shows the subclasses of the `<grid-view>` class.

Figure 16-1 Grid and list view classes



Objects created from the classes in Figure 16-1 represent the following kinds of views:

`<grid-view>`	Represents a view that is divided into cells.
`<text-grid-view>`	Represents a grid view whose cells display text.
`<list-view>`	Represents a view that is a single column of cells.
`<text-list-view>`	Represents a list view whose cells display text.

Grid View Behaviors

The Framework provides a behavior that handles grid selections:

`<grid-select-behavior>`	Represents a behavior that selects cells in a grid for editing. For more information about the `<grid-select-behavior>` class, see "The <grid-select-behavior> Class" on page 396.
`<grid-arrow-key-behavior>`	Represents a behavior that selects cells in a grid by using the arrow keys. For more information about the `<grid-arrow-key-behavior>` class, see "The <grid-arrow-key-behavior> Class" on page 396.

Using Grid and List View Objects

The following sections show examples of some of the tasks you may need to do when working with grid and list view objects. These tasks are as follows:

- set up a grid or list view
- specify how to draw in the view
- how to select cells
- how to highlight cells in a view
- how to handle variable column widths and row heights

Setting Up Grid and List Views

Figure 16-2 shows how to define a subclass of <grid-view> class and create an object from the subclass.

Figure 16-2 Setting up a grid view

```
define class <my-icon-grid-view> (<grid-view>)
end class;

let tool-grid = make(<my-icon-grid-view>,
                     columns: 2, rows: 4,
                     default-column-width: 34, default-row-height: 34,
                     draw-cell-border: #t);
add-behavior(tool-grid, make(<grid-select-behavior>));
add-behavior(tool-grid, make(<grid-arrow-key-behavior>));
```

The behaviors added to the grid view allow automatic selection of a cell by clicking the mouse and moving the arrow key. For more information about selecting cells, see “Selecting Cells” on page 390.

Drawing in Views

The draw method for a grid view calls the draw-cell method for each cell in the view. You can specialize the draw-cell method on the view, the row, or the column, or on some combination of these parameters.

With the exception of the <text-grid-view> objects, if a cell is not specific to the parameters of a draw-cell method that you provide, such as its row or column not being listed as a specialized parameter, the contents of the cell is not drawn.

The draw method for a list view calls the draw-item method for each item in the list. The draw-item method, in turn, calls draw-cell for each row in the first column, which is column 0. Thus, you can either specialize the draw-item method itself or specialize the column parameter for column 0 in your draw-cell method to draw the contents of a list view.

To draw <text-grid-view> and <text-list-view> objects, you must provide get-cell-text or get-item-text methods, respectively, that retrieves the text from the cell or item. The Framework draws the string that is returned by these functions; otherwise, nothing is drawn.

Selecting Cells

You can call select-cell, deselect-cell, and cell-selected? to handle selection of individual cells. You can add the <grid-select-behavior> and <grid-arrow-key-behavior> objects to your grid or list view to handle selection using the mouse and arrow-keys. For an example of setting up these behaviors, see Figure 16-2 on page 389.

Highlighting Cells in a View

The hilite methods for grid views and list views behave similarly to those for drawing. Specialize the parameters for the hilite-cell or hilite-item methods.

Handling Variable Column Widths

When you create a grid view, you can specify whether each column's width is fixed, which is the default, or whether the columns' widths may vary. Specify variable-column-widths: #t when you create the object to allow the column widths to vary. In addition to specifying true (#f) for the variable-column-widths: keyword, you must also provide a column-width method to determine the size of a column, because the method provided by the Framework only returns the default width. (The default is sufficient for fixed-width columns.)

The Framework provides an offset-to-column and offset-to-column methods to determine the position of a column in the view and determine the column's offset in a view, respectively. When variable columns can be used, the method calls column-width on each column to the left of the column whose position you want to determine. For views with many rows, you may wish to provide a more efficient implementation of the offset-to-column and offset-to-column methods.

Handling Variable Row Heights

When you create a grid or list view, you can specify whether each row's height is fixed, which is the default, or whether the rows' heights may vary. Specify `variable-row-heights: #t` when you create the object to allow the row heights to vary. In addition to specifying true (`#f`) for the `variable-row-heights:` keyword, you must also provide a `row-height` method to determine the height of a row, because the method provided by the Framework only returns the default height. (The default is sufficient for fixed-height rows.)

The Framework provides an `offset-to-row` and `offset-to-row` methods to determine the position of a row in the view and determine the row's offset in a view, respectively. When variable rows can be used, these methods call `row-height` on each row above the row whose position you want to determine. For views with many rows, you may wish to provide a more efficient implementation of the `offset-to-row` and `offset-to-row` methods.

Grid and List View Objects Reference

The following sections describe the classes that implement the Framework's view mechanism and identify the functions that you can use with them.

The <grid-view> Class

The `<grid-view>` class divides a view into cells that can be selected individually. You can create objects directly from a `<grid-view>` class or you can create subclasses and create objects from the subclasses. Listing 16-1 shows the class definition for the `<grid-view>` class.

Listing 16-1 The `<grid-view>` class definition

```
define primary open class <grid-view> (<view>)
  slot %rows :: <integer>,
    init-value: 1,
    init-keyword: rows:;
```

```
  slot %columns :: <integer>,
    init-value: 1,
    init-keyword: columns:;

  slot default-row-height :: <integer>,
    init-value: 0,
    init-keyword: default-row-height:;

  slot default-column-width :: <integer>,
    init-value: 0,
    init-keyword: default-column-width:;

  slot variable-row-heights? :: <boolean>,
    init-value: #f,
    init-keyword: variable-row-heights:;

  slot variable-column-widths? :: <boolean>,
    init-value: #f,
     init-keyword: variable-column-widths:;

  slot draw-cell-border? :: <boolean>,
     init-value: #f,
     init-keyword: draw-cell-border:;

  slot selected-cells :: <list>,
    init-value: #();
end class;
```

Slot descriptions

`%rows`	The number of rows in the view. Use the `rows:` keyword to specify the initial number of rows; otherwise, the view is created with only one row. You can use the `rows` method to determine the number of rows and the `rows-setter` method to change the number of rows.
`%columns`	The number of columns in the view. Use the `columns:` keyword to specify the initial number of columns; otherwise, the view is created with only one column. You can use the `columns` method to determine the number of columns and the `columns-setter` method to change the number of columns.

`default-row-height` The initial height of rows in the view. Use the `default-row-height:` keyword to specify the row height; otherwise, the height is 0. You can use the getter and setter to access this slot.

`default-column-width`
The initial width of columns in the view. Use the `default-column-width:` keyword to specify the column width; otherwise, the width is 0. You can use the getter and setter to access this slot.

`variable-row-heights?`
Specifies whether the height of rows in the view can vary `(#t)` or whether the height of each row is the same `(#f)`. Use the `variable-row-heights:` keyword to specify true `(#t)` to allow rows to have different heights; otherwise, the value is true`(#f)`, meaning that the height of each row is the same. You can use the getter and setter to access this slot.

`variable-column-widths?`
Specifies whether the width of columns in the view can vary `(#t)` or whether the width of each column is the same `(#f)`. Use the `variable-column-widths:` keyword to specify true `(#t)` to allow columns to have different widths; otherwise, the value is false`(#f)`, meaning that the width of each column is the same size. You can use the getter and setter to access this slot.

`draw-cell-border?` Specifies whether a border is to be drawn around each cell `(#t)` or whether each cell is to appear without a border `(#f)`. Use the `draw-cell-border:` keyword and specify false `(#f)` to prevent borders from being drawn; otherwise, the value is true`(#t)`, meaning that borders are drawn around cells. You can use the getter and setter to access this slot.

`selected-cells` A list of cells that are currently selected. Initially, the list is empty. You do not need to add or remove items from this list yourself; the `<grid-select-behavior>` object associated with the grid view manipulates this list for you.

Initialization

The `initialize` method for `<grid-view>` objects sets the view's extent to contain the number of rows and columns specified initially. The `initialize` method's parameters are defined as follows:

```
define method initialize (view :: <grid-view>, #key) => ()
```

The <text-grid-view> Class

The `<text-grid-view>` class handles drawing text in grid view cells. You create subclasses from the `<text-grid-view>` class and create objects from your subclasses. Your subclass should inherit the `text-getter` slot and you should provide an init function that retrieves the text.

Listing 16-2 shows the class definition for the `<text-grid-view>` class.

Listing 16-2 The `<text-grid-view>` class definition

```
define free open class <text-grid-view> (<grid-view>)
  slot text-getter :: <function>,
     init-value: default-<text-grid-view>-text-getter,
     init-keyword: text-getter:;
end class;
```

Slot descriptions

`text-getter` specifies the function that is called to get the text. By default, the `default-<text-grid-view>-text-getter` method is used, which concatenates the row and column number into a string, such as "1,2", where 1 is the row and 2 is the column.

Initialization

No `initialize` method is defined for `<text-grid-view>` objects.

The <list-view> Class

The `<list-view>` class is a grid view with a single column. You can create objects directly from a `<list-view>` class or you can create subclasses and create

objects from the subclasses. Listing 16-3 shows the class definition for the <list-view> class.

Listing 16-3 The <list-view> class definition

```
define primary open class <list-view> (<grid-view>)
    inherited slot %columns,
    init-value: 1;
end class;
```

Slot descriptions

columns	Specifies the number of columns to be 1.

Initialization

No initialize method is defined for <list-view> objects.

The <text-list-view> Class

The <text-list-view> class supports drawing text in a list view. You create subclasses from the <text-list-view> class and create objects from your subclasses. Your subclass should inherit the text-getter slot and you should provide an init function that retrieves the text.

Listing 16-4 shows the class definition for the <text-list-view> class.

Listing 16-4 The <text-list-view> class definition

```
define primary open class <text-list-view>
                                        (<list-view>, <text-grid-view>)
  inherited slot text-getter,
     init-value: default-<text-list-view>-text-getter;
end class;
```

Slot descriptions

text-getter	specifies the function that is called to get the text. By default, the default-<text-list-view>-text-getter method is used, which returns the row number as text.

Initialization

No `initialize` method is defined for `<text-list-view>` objects.

The <grid-select-behavior> Class

The `<grid-select-behavior>` class handles cell selection when the user clicks the mouse on a cell in a grid view. You must add the behavior to your grid view class to enable this selection behavior.

Listing 16-5 shows the class definition for the `<grid-select-behavior>` class.

Listing 16-5 The `<grid-select-behavior>` class definition

```
define primary open class <grid-select-behavior> (<behavior>)
    slot multiple-selection? :: <boolean>,
        init-value: #f,
        init-keyword: multiple-selection:;
end class;
```

Slot descriptions

`multiple-selection?`

Specifies whether you want to keep previous selections and adding the cell to the selection when the user clicks on a cell while depressing the shift key. Use the `multiple-selection:` keyword to specify true (`#t`) for multiple-selection support; otherwise, previous selections are lost when a cell is selected—the use of the shift key has no effect.

Initialization

No `initialize` method is defined for `<grid-select-behavior>` objects.

The <grid-arrow-key-behavior> Class

The `<grid-arrow-key-behavior>` class handles cell selection when the user presses the arrow key to move to a cell in a grid view. You must add the behavior to your grid view class to enable this selection behavior.

Listing 16-6 shows the class definition for the `<grid-arrow-key-behavior>` class.

Listing 16-6 The `<grid-arrow-key-behavior>` class definition

```
define primary open class <grid-arrow-key-behavior> (<behavior>)
end class;
```

Initialization

No `initialize` method is defined for `<grid-arrow-key-behavior>` objects.

Grid and List View Related Functions

You can use any of the view-related functions, identified in Table 14-3 on page 364, to manipulate grid and list views. Table 16-1 shows additional functions you can use for grid views.

Table 16-1 Functions for grid views

Function	Purpose	Call	Sp.
`row-heigth`	Determine the row's height.	√	√
`column-width`	Determine the column's width.	√	√
`point-to-cell`	Determine the row and column specified by a point.	√	
`offset-to-row`	Determine the row specified by a vertical coordinate.	√	√
`offset-to-column`	Determine the column specified by a horizontal coordinate.	√	√
`row-to-offset`	Determine the vertical coordinate of a point at the bottom of the specified row.	√	√
`column-to-offset`	Determine the horizontal coordinate of a point at the beginning of a column.	√	√
`cell-bounds`	Determine the bounding rectangle of a cell.	√	

Table 16-1 Functions for grid views

Function	Purpose	Call	Sp.
`cell-in-rect?`	Determine if a cell is in the specified rectangle.	√	
`draw-cell`	Draw the specified cell in the view.		√
`hilite-cell`	Draw the specified cell in the view.		√
`draw-cell-border`	Draw a border around the specified cell.	√	
`invalidate-cell`	Invalidate the area occupied by a cell.	√	
`validate-cell`	Validate the area occupied by a cell.	√	
`select-cell`	Select the specified cell.	√	
`deselect-cell`	Deselect the specified cell.	√	
`cell-selected?`	Determine if the cell is currently selected.	√	

Although you can use the functions for grid views to manipulate list views, the Framework provides several additional functions to make manipulation of list views easier. Table 16-2 identifies these functions.

Table 16-2 Functions for list views

Function	Purpose	Call	Sp.
`item-bounds`	Determine the bounding rectangle of an item.	√	
`draw-item`	Draw the specified item in the view.		√
`hilite-item`	Draw the specified item in the view.		√
`invalidate-item`	Invalidate the area occupied by an item.	√	
`validate-item`	Validate the area occupied by an item.	√	

CHAPTER 17

Windows

Contents

About Windows 401
Using Window Objects 402
  Creating a Window 402
  Creating a Window for a Dialog Box 404
  Creating a Floating Window 404
Window Objects Reference 406
  The <window-context> Class 406
  The <window> Class 407
  The Grow Icon Classes 413
    The <grow-icon> Class 414
    The <grow-icon-determiner> Class 414
  Functions for Windows 415

This chapter discusses windows, which represent Macintosh Window Manager windows and window contexts, which provide the association between an event handler, such as a document, and a window, which is the document's context onscreen.

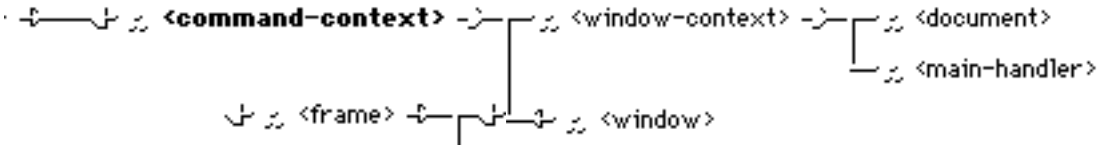
About Windows

You can create the same kinds of windows with the Framework as you can with the Window Manager. The `<window>` class represents Window Manager windows. The `<window>` object is a frame and an event handler. The frame defines a window's size and location. The event handler allows the window to respond to events. If you do not specify otherwise, a window's next handler is the main handler. Thus, you must specify a document as the window's next handler if the window displays a document's contents.

Associated with each window is a root view provided by the Framework, which is at the top of a view hierarchy. In general, you do not need to manipulate windows because most events in windows are handled by either by the Framework or passed to the root view to handle. If an event is in the content area of the window, the root view allows subviews, which are your content views, to handle the event. For more information about views and the view hierarchy, see "About Views and Supporting Classes" on page 335.

Windows and window contexts are subclasses of the `<event-handler>` class and the `<command-context>` class. The window class is also a subclass of the `<frame>` class. Figure 17-1 shows the subclasses of the `<command-context>` class.

Figure 17-1 The `<window>` and `<window-context>` classes



Classes shown in Figure 17-1 represent the following kinds of objects:

`<command-context>`	Represents the context for an undoable and redoable action, which are represented by commands. For more information about commands and their contexts, see "The <command-context> Class" on page 539.
`<window-context>`	Represents a command context for one or more windows. Fore more information about the `<window-context>` class, see "The <window-context> Class" on page 406.
`<document>`	Represents a document. For information about the `<document>` class, see "The <document> Class" on page 485.
`<main-handler>`	Represents the event handler of "last resort," which is typically the application. For information about documents, see "The <main-handler> Class" on page 265.
`<frame>`	Represents an object that maintains a location and size. For information about the `<frame>` class, see "The <document> Class" on page 485.
`<window>`	Represents a window. Fore more information about the `<window>` class, see "The <frame> Class" on page 319.

Using Window Objects

The following sections show examples of creating windows.

Creating a Window

The Framework provides a method, `make-scrolling-window`, that creates a window and sets up the views required for scrolling. For information about setting up a window for scrolling, see "Setting up a Scrolling View" on page 465. Listing 17-1 shows how to create a window for a document. The code shows a call to `make-scrolling-window`, which is used as the parameter to the `open` function that opens the window once it has been created.

Listing 17-1 Creating a scrolling window

```
open(make-scrolling-window(list(view), title: document.title,
                           target-view: view,
                           next-handler: document,
                           main-window: #t,
                           location: point(100, 100),
                           extent: point(400, 300),
                           closable: #t, zoomable: #t, resizable: #t));
```

With the exception of the view list, the parameters to `make-scrolling-window` are used by the window's `make` method. The following items are specified:

- the `title:` keyword specifies the title; in this case, it is the title of the document.
- the `target-view:` keyword specifies the target view, which is the first view in the list of views.
- the `next-handler:` keyword specifies the next event handler, which is the document.
- the `main-window:` keyword specifies whether it is the main window; because it is the first window opened by the document, it is true (#t).
- the `location:` keyword specifies where to place the window in relation to the main monitor; in this case, it is 100 pixels down and 100 pixels to the right of the upper-left corner of the monitor.
- the `extent:` keyword specifies the size of the content area of the window; in this case it is 400 pixels high and 300 pixels wide.
- the `closeable:` keyword specifies whether the window has a close box; it does.
- the `zoomable:` keyword specifies whether the window has a zoom box; it does.
- the `resizable:` keyword specifies whether the window has a resize box and can be resized; it does.

Listing 17-2 shows how part of the `make-scrolling-window` method is implemented.

Listing 17-2 Creating a window

```
define constant make-scrolling-window =
     method (view-list :: <list>,
             #rest window-args) => result :: <window>;
       let the-window = apply(make, <window>, window-args);
let root = the-window.root-view;
…
```

The `apply` function creates a call to `make` for the `<window>` class, using the `#rest` parameters passed into the `make-scrolling-window` method. One of the things that the `initialize` method does is create the root view, which is then available for use as the root of the view hierarchy associated with the window.

Creating a Window for a Dialog Box

A window for a dialog box is modal; however, it does not have a close box and typically cannot be grown or resized. Listing 17-3 shows how to create a dialog box window. For more information about dialog boxes, see "About Dialogs and Controls" on page 419.

Listing 17-3 Creating a dialog box window

```
let the-window = make(<window>,
                      location: point(50, 50),
                      extent: point(350, 150),
                      closable: #f,
                      modal: #t,
                      title: "Dialog");
```

Creating a Floating Window

You create a floating window in the same way that you create other windows. The only difference is that you specify true (`#t`) for the `float:` keyword. Listing 17-4 shows an example of creating a floating window from the Framework's `make-palette-window` method.

Listing 17-4 Creating a Floating Window

```
define constant make-palette-window =
     method (view-list :: <list>,#key
                                   bounds: window-bounds,
                                   title: window-title = "",
                                   target: target) => result :: <window>;
  let bounds = if (window-bounds)
                 window-bounds;
               else
                 make(<rect>, top: 40, left: 10, bottom: 250,
                                                          right: 120);
               end;

  let the-window = make(<window>,
                        title: window-title,
                        location: top-left(bounds),
                        extent: point(bounds.right - bounds.left,
                                      bounds.bottom - bounds.top),
                        closable: #t,
                        floats: #t,
                        target-view: target);

  let root = the-window.root-view;

  // add all the views in view-list to the root view
  for (view in view-list)
    add-sub-view(root, view, update-caches: #f);
  end for;

  open(the-window);

  // return the window as the result
  the-window;
end method;
```

Window Objects Reference

The following sections discuss window-related classes and identify the functions that you use with them.

The <window-context> Class

The `<window-context>` class allows you to define subclasses of event handlers to manage one or more windows. For example, if your windows are not associated with a document, you could define a subclass of `<window-context>` in the same way that the Framework provides a `<document>` subclass of the `<window-context>` class. You do not create objects directly from a `<window-context>` class, rather you define a subclass of `<window-context>` and create objects of your subclass.

Note
Defining and using subclasses of `<window-context>` is completely experimental. ◆

Listing 17-5 shows the class definition for the `<window-context>` class.

Listing 17-5 The `<window-context>` class definition

```
define primary open class <window-context> (<command-context>)
  slot main-window, init-value: #f;
  slot close-with-last-window? :: <boolean>,
     init-value: #t,
     init-keyword: init-with-last-window:;
end class;
```

Slot descriptions

`main-window` The main window in this context. Use the `main-window:` keyword to specify the main window; otherwise, the value is false (`#f`). You can use the getter and setter to access this slot.

`close-with-last-window?`

Specifies whether the children of a context, which typically are windows, should be closed when the last window associated with the context is closed `(#t)` or whether they should remain open after the last window associated with the context is closed `(#f)`. Use the `close-with-last-window:` keyword and specify false `(#f)` to allow children of a context to remain open; otherwise, the value is true `(#t)`, meaning that all children of the context will be closed when the last window is closed. You can use the getter and setter to access this slot.

Initialization

No `initialize` method is defined for `<window-context>` objects.

The <window> Class

The `<window>` class represents a Macintosh Window Manager window. You do not need to define subclasses of the `<window>` class. You only need to create `<window>` objects. If the window is associated with a document, you typically create windows in a `make-windows` method, which is called when the document is opened. Listing 17-6 shows the class definition for the `<window>` class.

Listing 17-6 The `<window>` class definition

```
define class <window> (<command-context>, <frame>)
  slot target-view,
     init-value: #f;

  slot window-ptr :: <WindowRecord>,
     init-value: as(<WindowRecord>, 0);

  slot window-kind :: <integer>, init-value: #f,
     init-keyword: window-kind:;

  slot resizable? :: <boolean>,
     init-value: #f,
     init-keyword: resizable:;
```

```
slot closable? :: <boolean>,
   init-value: #f,
   init-keyword: closable:;

slot zoomable? :: <boolean>,
   init-value: #f,
   init-keyword: zoomable:;

slot active? :: <boolean>,
   init-value: #f;

slot %modal? :: <boolean>,
   init-value: #f,
   init-keyword: modal:;

slot %floats? :: <boolean>,
   init-value: #f,
   init-keyword: floats:;

slot open-initially? :: <boolean>,
   init-value: #t,
   init-keyword: open-initially:;

slot dispose-on-close? :: <boolean>,
   init-value: #t,
   init-keyword: dispose-on-close:;

slot main-window? :: <boolean>,
   init-value: #f,
   init-keyword: main-window:;

slot first-click? :: <boolean>,
   init-value: #f,
   init-keyword: first-click:;

slot qd-graphics-port :: <qd-graphics-port>;
slot gx-graphics-port, init-value: #f;

slot minimum-extent :: <point>,
```

```
      init-value: $window-minimum-extent,
      init-keyword: minimum-extent:;

  slot maximum-extent :: <point>,
      init-value: $window-maximum-extent,
      init-keyword: maximum-extent:;

slot scripting-object,
      init-value: #f,
      init-keyword: scripting-object:;

  slot window-content-color,
      init-value: #f,
      init-keyword: window-content-color:;

  // private
  // use the generic functions location, window-size, visible?
  // root-view, title, and their setters to access these slots.
  slot %root-view, init-value: #f;
  slot %visible? :: <boolean>,
      init-value: #f;
  slot was-visible? :: <boolean>,
      init-value: #f;
  slot %title :: <string>;
  end class;
```

Slot descriptions

`target-view` The view that initially receives events when this window is active. Use the `target:` or `target-id:` keywords to specify the target view; otherwise, the target is assumed to be defined in a stream. Call `target-view` to determine the value of this slot. Call the `set-target` method to set the value of this slot.

`window-ptr` The Macintosh Window Manager window. The window is defined using the toolbox `WindowRecord` data type. For information about the `WindowRecord` data type, see the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

`window-kind` The type of window, which are described in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox*

Essentials. Use the `window-kind:` keyword to specify the kind of window; otherwise the values of the `floats?`, `modal?`, and `zoomable?` slots determine the value of the `window-kind` slot. If the `window-kind:` keyword is not specified and the values for the `floats?`, `modal?`, and `zoomable?` slots are each false (`#f`), the window type is `documentProc`, meaning that the window has a title bar and supports a close box and size box but no zoom box.

`resizable?` Specifies whether the size of the window can be changed `(#t)` or not `(#f)`. Use the `resizable:` keyword to specify true `(#t)` to allow the window to be resized; otherwise, the value is false`(#f)`, meaning that a "grow box" is not present and the window cannot be resized. You can use the getter and setter to access this slot.

`closable?` Specifies whether the window supports a close box `(#t)` or not `(#f)`. Use the `closable:` keyword to specify true `(#t)` to provide a close box; otherwise, the value is false`(#f)`, meaning that a close box is not present. You can use the getter and setter to access this slot.

`zoomable?` Specifies whether the window can be zoomed in or out `(#t)` or not `(#f)`. Use the `zoomable:` keyword to specify true `(#t)` to provide a zoom box; otherwise, the value is false`(#f)`, meaning that a zoom box is not present. You can use the getter and setter to access this slot.

`active?` Specifies whether the window is active `(#t)` or not `(#f)`. The initial value is false`(#f)`, meaning that the window is not active. You can use the getter to determine whether or not a window is active. The Framework sets this slot during event handling for window events.

`%modal?` Specifies whether the window is modal `(#t)` or not `(#f)`. Use the `modal:` keyword to specify true `(#t)` to provide a modal window; otherwise, the value is false`(#f)`, meaning that the window must be closed before the application can continue. You can use the `modal?` and `modal?-setter` methods to access this slot.

`%floats?` Specifies whether the window floats `(#t)` or not `(#f)`. Use the `floats:` keyword to specify true `(#t)` to provide a floating window; otherwise, the value is false`(#f)`, meaning that the window is not a floating window. You

	can use the `floats?` and `floats?-setter` methods to access this slot.
`open-initially?`	Specifies whether the window should be opened when its context, such as a document, is opened `(#t)` or whether the window should be opened explicitly `(#f)`. Use the `open-initially:` keyword and specify false `(#f)` if you do not want the window to be opened with its context; otherwise, the value is true `(#t)`, meaning that the window is opened with its context. You can use the getter and setter to access this slot.
`dispose-on-close?`	Specifies whether the window and its data can be garbage collected when it is closed `(#t)` or whether the window is simply hidden from view `(#f)`. Use the `dispose-on-close:` keyword and specify false `(#f)` if you want the window to hidden rather than be garbage collected; otherwise, the value is true `(#t)`, meaning that a new window and its views must be recreated if the window is needed again. Setting the value to false (`#f`) is useful when the window is a dialog box that is likely to be invoked several times. You can use the getter and setter to access this slot.
`main-window?`	Specifies whether the window is the main one in its context, such as the main window for a document `(#t)` or not `(#f)`. Use the `main-window:` keyword and specify false `(#f)` if this is not the main window in its context; otherwise, the value is true `(#t)`, meaning that this is the main window in its context. You can use the getter and setter to access this slot.
`first-click?`	Specifies whether the window responds to the first click `(#t)` or not `(#f)`. Use the `first-click:` keyword to specify true `(#t)` if the window should respond to the first click; otherwise, the value is false `(#f)` for non-floating windows and true (`#t`) for floating windows, meaning that a floating window responds immediately to the first click in its content area while the first click in a non-floating window activates the window. You can use the getter and setter to access this slot.
`qd-graphics-port`	The QuickDraw GrafPort associated with the window. Internal.

`minimum-extent` A point that specifies the minimum size for the window. Use the `minimum-extent:` keyword to specify the extent; otherwise the value is `$minimum-extent`, which is defined as `point(80, 80)`. You can use the getter and setter to access this slot.

`maximum-extent` A point that specifies the minimum size for the window. Use the `maximum-extent:` keyword to specify the extent; otherwise the value is `$maximum-extent`, which is defined as `point(1000000, 1000000)`. You can use the getter and setter to access this slot.

`scripting-object` Specifies the scripting object that responds to Apple events sent to the window. For more information about scripting objects, see "About Scripting Support" on page 601.

`window-content-color`
Specifies the color to use for the window's contents.

`%root-view` The root view associated with the window. Use the `root-view:` keyword to specify whether the view is visible; otherwise, it is false (`#f`). Use the `root-view` and `root-view-setter` functions to access this slot. The `%root-view` slot is not exported.

`%visible?` Specifies whether any part of the view is visible onscreen. Use the `visible:` keyword to specify whether the view is visible; otherwise, it is true (`#t`). Use the `visible` and `visible-setter` functions to access this slot. The `%visible?` slot is not exported.

`was-visible?` Specifies whether a floating window was visible onscreen. Internal.

`%title` The window's title. Use the `title:` keyword to specify the title; otherwise, the slot is empty. Use the `title` and `title-setter` functions to access this slot. The `%title` slot is not exported.

Note
Setting the value of most slots has no effect if the window is already open. For example, if you change a window from modal to not modal, the change does not become effective until the window is closed and reopened. ◆

Initialization

The `initialize` method for `<window>` objects. Use the `item-list:` keyword to load the items dynamically. Use the `menu-rsrc-id:` keyword if you want to load the menu items from a resource. The `initialize` method's parameters are defined as follows:

```
define method initialize (window :: <window>,
                          #key title: title = "",
                               target-view: target,
                               target-id: target-id,
                                root-view: root
                         ) => ()
```

`window`	The window.
`title`	The window's title. Use the `title:` keyword to specify the title; otherwise, the title is an empty string.
`target`	The object that is the target view. Use the `target:` keyword to specify this view.
`target-id`	The ID of the target view. Use the `target-id:` keyword if you want to specify one of the subviews of the root view for the window.
`root`	The root view from the stream containing the window. Internal.

Specify either the `target:` or the `target-id:` keywords, but not both of them. If the specified target view that cannot be found, the root view becomes the target.

The Grow Icon Classes

The Framework provides the `<grow-icon>` and `<grow-icon-determiner>` classes to implement the "grow box" for a window. The `<grow-icon>` class implements the view for a "grow box" control. The `<grow-icon-determiner>` class implements the placement of the icon when the size or location of the window changes. You do not need to define subclasses of the `<grow-icon>` and `<grow-icon-determiner>` classes, nor do you need to create objects from these classes. The Framework completely handles manipulation of a window's "grow box" for you.

The <grow-icon> Class

Listing 17-7 shows the class definition for the `<grow-icon>` class.

Listing 17-7 The `<grow-icon>` class definition

```
define primary sealed class <grow-icon> (<view>)
  inherited slot identifier,
     init-value: #"grow-icon";
end class;
```

Slot descriptions

`identifier` The identifier for a `<grow-icon>` object.

Initialization

The `initialize` method for `<grow-icon>` objects installs a determiner so that the icon is placed in the lower-right corner of the window. The `initialize` method's parameters are defined as follows:

```
define method initialize (icon :: <grow-icon>, #key)
```

`icon` The grow icon.

The <grow-icon-determiner> Class

Listing 17-8 shows the class definition for the `<grow-icon-determiner>` class.

Listing 17-8 The `<grow-icon-determiner>` class definition

```
define primary sealed class <grow-icon-determiner> (<view-determiner>)
end class;
```

Initialization

No `initialize` method is defined for `<grow-icon-determiner>` objects.

Functions for Windows

The following functions can be used with window contexts.

Table 17-1 Functions for window contexts

Function	Purpose	Call	Sp.
`open`	Specify the actions to take when the context is opened.		√
`add-window`	Add a window to a context, such as the document's or main handler's context.	√	
`remove-window`	Remove a window from a context, such as a document or main handler.	√	

The following functions can be used with `<window>` objects.

Table 17-2 Functions for windows

Function	Purpose	Call	Sp.
`open`	Open the window.	√	
`close`	Close the window.	√	
`draggable?`	Determine whether the window can be moved.	√	
`get-command-context`	Get the context in which undo/redo is a applicable.	√	
`get-window-color`	Determine the window's color from its Window record.	√	
`set-window-color`	Set the window's color into its Window record.	√	
`print-views`	Print the views for this window.	√	
`pose-modally`	Pose a window as a modal dialog box.	√	

Windows

CHAPTER 18

Dialogs and Controls

Contents

About Dialogs and Controls 419
- Dialog Behaviors 419
- Control View Classes 421
- Control Events 423
- Highlighting in Controls 424

Using Dialogs and Controls 424
- Creating a Modal Dialog Box 425
  - Creating a Window for a Modal Dialog Box 429
  - Setting up Dialog Box Dismissers 429
  - Creating Static Text 430
  - Creating Number Text 430
  - Implementing a Popup Menu 431
  - Adding a Tabbing Behavior 432
  - Opening the Modal Dialog Box 432

Dialog and Control Objects Reference 433
- The <dialog-behavior> Class 433
- The <tabber> Class 435
- The <edit-text-dialog-filter> Class 436
- The <cluster> Class 437
- The <simple-icon> Class 438
- The <static-text> Class 439
- The <control> Class 440
- Subclasses of the <control> Class 442
  - The <button> Class 443
  - The <check-box> Class 443
  - The <radio> Class 443

The <scroll-bar> Class 444
The <popup> Class 445
Control Manager Control Classes 446
The <ctl-mgr-control> Class 446
The <ctl-mgr-button> Class 448
The <ctl-mgr-check-box> Class 449
The <ctl-mgr-radio> Class 450
The <ctl-mgr-scroll-bar> Class 451
The <ctl-mgr-popup> Class 452
Control Event Classes 454
The <default-button-adorner> Class 455
Functions for Dialogs and Controls 456

This chapter discusses the implementation of dialogs and controls. Dialogs are implemented as behaviors that are attached to windows. These windows are the dialog boxes that contain views, such as the following:

- edit text and number text, which are views.
- Control Manager controls, which are classes derived from several Framework classes including views.

The views and controls respond to events in the same way that other event handlers do.

For information about edit text and number text view classes, see "About Text View Classes" on page 373. This chapter discusses the behaviors that implement dialogs and the controls that are commonly found in dialog boxes and the events that are specific to controls.

About Dialogs and Controls

There are four kinds of classes that you will use to implement dialog boxes:

- behaviors that support dialogs
- views that implement controls
- views that implement editable fields
- events to which controls respond

The following sections discuss dialog behaviors, control views, and control events.

Dialog Behaviors

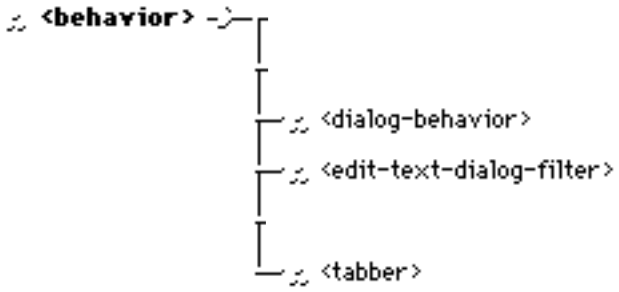
The Framework implements dialog boxes as windows that are posed modally. You create a window object. When you call the `pose-modally` function, the Framework attaches a dialog behavior to the window that controls which keystrokes or mouse clicks allow the dialog box can be dismissed. When the dialog box is dismissed, the behavior is removed.

The dialog may contain fields, which are text views, such as edit text or number text objects. The Framework provides a behavior that allows the user

to tab (or back-tab) between these views. The Framework also provides a behavior that prevents the keys associated with a dialog box from being interpreted as data in an edit text field.

Figure 18-1 shows the dialog-related subclasses of the <behavior> class.

Figure 18-1 The dialog behavior classes



Objects created from the classes in Figure 18-1 represent the following kinds of views:

<dialog-behavior> Represents a behavior that implements modal dialogs. For more information about the <dialog-behavior> class, see "The <dialog-behavior> Class" on page 433.

<edit-text-dialog-filter> Represents a behavior that handles conflicting keys, such as Tab and Return, while entering text in a dialog. For more information about the <edit-text-dialog-behavior> class, see "The <edit-text-dialog-filter> Class" on page 436.

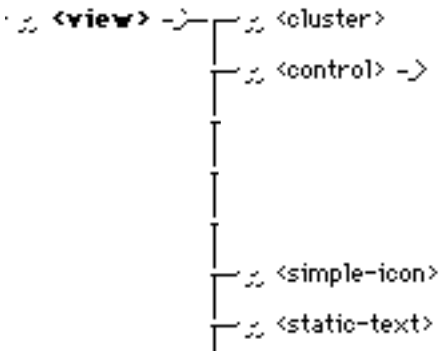
<tabber> Represents a behavior that provides tabbing between fields in a dialog box. For more information about the <tabber> class, see "The <tabber> Class" on page 435.

Control View Classes

Control view classes give a control its appearance onscreen. Controls are therefore derived directly from the <view> class, or derived indirectly, as a subclass of the <control> class. Controls that correspond to Control Manager controls and respond to events, are derived from both the <control> class and another subclass of <control>, the <ctl-mgr-control> class. The <ctl-mgr-control> class is mixed in with a subclass of <control>. Thus the <control> subclass implements the view and the <ctl-mgr-control> class implements the Control Manager capability.

Figure 18-2 shows the subclasses of the <view> class that are related to dialogs.

Figure 18-2 Dialog-related view classes

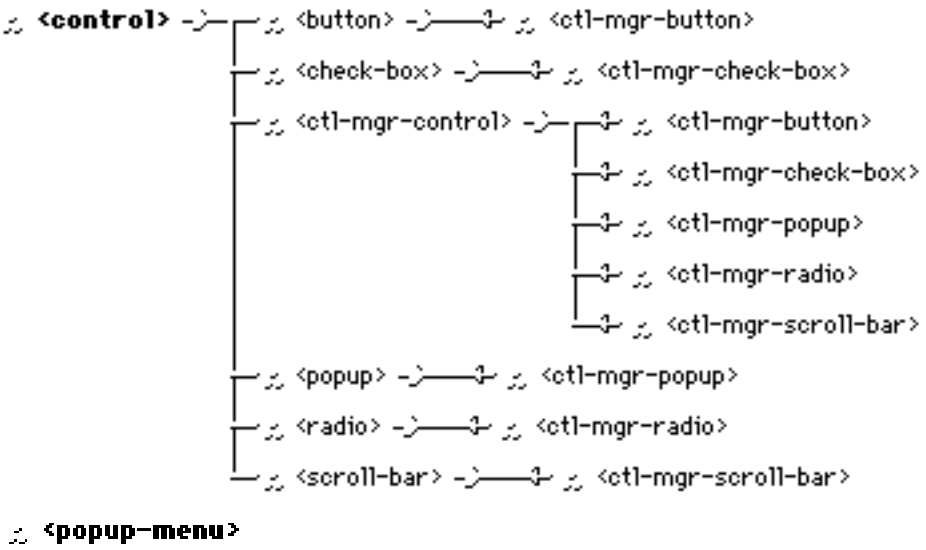


Objects created from the classes in Figure 18-2 represent the following kinds of views:

<cluster>	Represents a group of controls that operate together, such as a group of radio buttons.
<control>	Represents a control. See the next section for information on the <control> class and its subclasses.
<simple-icon>	Represents an icon.
<static-text>	Represents a view for unchangeable text.

Figure 18-3 shows the <control> class (from Figure 18-3) and its subclasses.

Figure 18-3 The <control> class and its subclasses



Objects created from the subclasses of <control> in Figure 18-3 represent the following kinds of views:

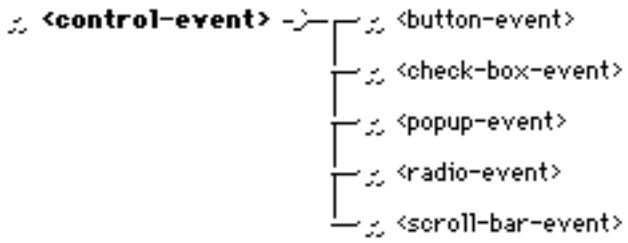
<button>	Represents an event associated with a push button control.
<check-box>	Represents an event associated with a check box control.
<popup>	Represents an event associated with a popup menu control.
<radio>	Represents an event associated with a radio control.
<scroll-bar>	Represents an event associated with a scroll bar control.
<ctl-mgr-control>	Represents a Control Manager control.
<ctl-mgr-button>	Represents a Control Manager push button control.

<ctl-mgr-check-box>	Represents a Control Manager check box control.
<ctl-mgr-popup>	Represents a Control Manager popup menu control.
<ctl-mgr-radio>	Represents a Control Manager radio control.
<ctl-mgr-scroll-bar>	Represents a Control Manager scroll bar control.
<popup-menu>	Represents a menu inside a Control Manager popup menu control. Internal.

Control Events

Control events are a subset of window events to which controls respond. These controls are the views that are subviews of a window's root view, such as push buttons. Figure 18-4 shows the subclasses of the <control-event> class and its subclasses.

Figure 18-4 The <control-event> class and its subclasses



Objects created from the classes in Figure 18-4 represent the following kinds of views:

<control-event>	Represents an event associated with a control.
<button-event>	Represents an event associated with a push button control.
<check-box-event>	Represents an event associated with a check box control.
<popup-event>	Represents an event associated with a popup menu control.

`<radio-event>`	Represents an event associated with a radio control.
`<scroll-bar-event>`	Represents an event associated with a scroll bar control.

Highlighting in Controls

Controls can be defined to support one of three mode styles, where these modes determine the hiliting behavior for the control. These modes are specified using symbols shown in Table 18-1:

Table 18-1 Control modes

Contorl Mode	Highlighting
`#"no-mode"`	Control does not handle highlighting.
`#"button-mode"`	Handles normal highlighting for buttons.
`#"radio-mode"`	Highlighting stays on whenthe button is clicked.
`#"switch-mode"`	Highlighting is toggled from on to orr or off to on when the control is clicked.

These modes are used by the `do-event` method to determine how to handle the hiliting for the control. This can primarily be used to implement customized controls that do not use the Control Manager.

Note
The control class does not do any hiliting based on these modes it is up to the subclass to implement the appropriate hiliting, either in the draw method or in an adorner. ◆

Using Dialogs and Controls

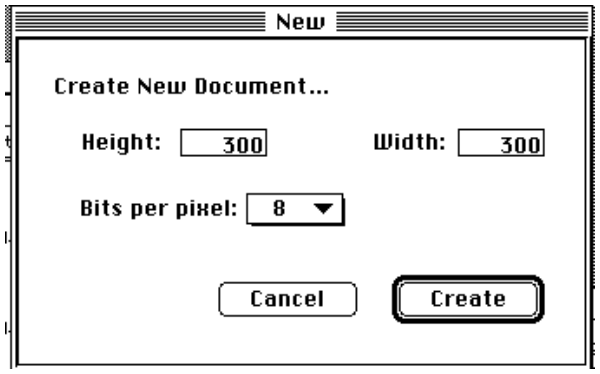
The following sections show examples of:

- creating a modal dialog box that consists of a window, static text, number text, and controls
- responding to control events

Creating a Modal Dialog Box

The Framework creates a modal dialog box from the window you define. Figure 18-5 shows a dialog box that contains static text, number text, a popup menu, and two push buttons. The push buttons are the Cancel button and the default button, labeled Create.

Figure 18-5 Moveable modal dialog box



Listing 18-1 shows the code that creates this dialog box and displays it.

Listing 18-1 Creating a dialog box

```
define variable *last-depth* = 8;

define variable *last-height* = 300;

define variable *last-width* = 300;
```

```
define constant new-image-dialog =
    method ()
        => (height :: <integer>, width :: <integer>, depth :: <integer>)
        let the-window = make(<window>,
                              location: point(50, 50),
                              extent: point(320, 180),
                              closable: #f,
                              modal: #t,
                              title: "New");

        let ok-button = make(<ctl-mgr-button>,
                    location: point(the-window.extent.h - 16 - 80 - 8,
                    the-window.extent.v - 16 - 20 - 8),
                    extent: point(80, 20),
                    title: "Create", dismisses-dialog: #t,
                    default: #t);
        add-sub-view(the-window.root-view, ok-button);

        let cancel-button = make(<ctl-mgr-button>,
            location: point(the-window.extent.h - 2 * 16 - 80 - 8 - 88,
            the-window.extent.v - 16 - 20 - 8),
            extent: point(80, 20),
            title: "Cancel", dismisses-dialog: #t);
       add-sub-view(the-window.root-view, cancel-button);


       let dialog-prompt
            = make(<static-text>, text: "Create New Document…",
                    location: point(16, 16),
                    extent: point(300, 16));
       add-sub-view(the-window.root-view, dialog-prompt);


       let height-prompt = make(<static-text>, text: "Height:",
                                location: point(32, 48),
                                extent: point(100, 16));
       add-sub-view(the-window.root-view, height-prompt);

       let height-text = make(<number-text>,
                              location: point(90, 48),
```

```
                        extent: point(50, 16),
                        current-value: *last-height*);
add-adorner(height-text, $frame-adorner);
add-sub-view(the-window.root-view, height-text);


let width-prompt = make(<static-text>, text: "Width:",
                         location: point(200, 48),
                         extent: point(100, 16));
add-sub-view(the-window.root-view, width-prompt);

let width-text = make(<number-text>,
                      location: point(250, 48),
                      extent: point(50, 16),
                      current-value: *last-width*);
add-adorner(width-text, $frame-adorner);
add-sub-view(the-window.root-view, width-text);


let depth-popup = make(<ctl-mgr-popup>,
                        identifier: #"popup",
                        location: point(32 - 4, 85),
                        extent: point(180, 20),
                        title: "Bits per pixel:",
                        item-list: list(pair("1", #"1"),
                                        pair("2", #"2"),
                                        pair("4", #"4"),
                                        pair("8", #"8"),
                                        pair("16", #"16"),
                                        pair("32", #"32")));
depth-popup.control-value := select(*last-depth*)
                               1 => #"1";
                               2 => #"2";
                               4 => #"4";
                               8 => #"8";
                               16 => #"16";
                               32 => #"32";
                             end select;

add-sub-view(the-window.root-view, depth-popup);
```

```
    add-behavior(the-window.root-view, make(<tabber>));

    the-window.target-view := height-text;
    open(the-window);

    block()
      let dismisser = pose-modally
             (the-window, default: ok-button, cancel: cancel-button);

      if (dismisser = cancel-button)
        abort();
      end if;

      *last-height* := current-value(height-text);
      *last-width* := current-value(width-text);
      *last-depth* := select(current-choice-id(depth-popup))
                        #"1" => 1;
                        #"2" => 2;
                        #"4" => 4;
                        #"8" => 8;
                        #"16" => 16;
                        #"32" => 32;
                        otherwise => #f;
                      end select;

      values(*last-height*, *last-width*, *last-depth*);
    cleanup
      close(the-window);
    end block;
  end method;
```

The code in Listing 18-1 shows several how to do several things:

- create a window for a modal dialog box
- create buttons that dismiss the dialog
- create static text
- create number text
- implement a popup menu

- implement tabbing behavior
- pose a modal dialog

The following sections go into more detail.

Creating a Window for a Modal Dialog Box

A modal dialog box is simply a window that does not have close, grow, or resize icons. The window is still moveable. For more information about windows, see "About Windows" on page 401. Listing 18-2 shows how the dialog box is created.

Listing 18-2 Creating a moveable modal dialog box

```
let the-window = make(<window>,
                              location: point(50, 50),
                              extent: point(320, 180),
                              closable: #f,
                              modal: #t,
                              title: "New");
```

Setting up Dialog Box Dismissers

Dialog box dismissers are controls that dismiss the dialog. The `dismisser: #t` keyword specifies this capability. Listing 18-3 shows two `<ctl-mgr-button>` objects, either of which dismisses the dialog.

The OK button is also the default button (`default: #t`), meaning that it has a wide border that makes it stand out. Note that both controls are added as views to the window's root view.

Listing 18-3 Setting up dialog box dismissers

```
let ok-button = make(<ctl-mgr-button>,
                     location: point(the-window.extent.h - 16 - 80 - 8,
                     the-window.extent.v - 16 - 20 - 8),
                     extent: point(80, 20),
```

```
                title: "Create", dismisses-dialog: #t,
                default: #t);
        add-sub-view(the-window.root-view, ok-button);

        let cancel-button = make(<ctl-mgr-button>,
            location: point(the-window.extent.h - 2 * 16 - 80 - 8 - 88,
            the-window.extent.v - 16 - 20 - 8),
            extent: point(80, 20),
            title: "Cancel", dismisses-dialog: #t);
       add-sub-view(the-window.root-view, cancel-button);
```

Creating Static Text

Static text is text that is displayed but which is not editable. Listing 18-4 shows creating static text and adding it to the dialog box.

Listing 18-4 Creating static text

```
let dialog-prompt = make(<static-text>, text: "Create New Document…",
                                location: point(16, 16),
                                extent: point(300, 16));
       add-sub-view(the-window.root-view, dialog-prompt);
```

Creating Number Text

Number text is text that can be edited, however, its contents are restricted to digits. Listing 18-5 shows creating number text and adding it to the dialog box. The current value of the field is set, although the range is not specified. Note that a frame adorner is added to the view; otherwise, the field's size would not be obvious.

Listing 18-5 Creating number text

```
let height-text = make(<number-text>,
                               location: point(90, 48),
                               extent: point(50, 16),
```

```
                            current-value: *last-height*);
    add-adorner(height-text, $frame-adorner);
    add-sub-view(the-window.root-view, height-text);
```

Implementing a Popup Menu

Listing 18-6 shows how to create a popup menu and add it to a dialog box. The `item-list:` keyword specifies the items' titles and assigns an identifier to each one. For example, the title of the first item is "1" and its identifier is `#"1"`. The current value is also being set using the item identifiers, such that if the value of `*last-depth*` is 1, the item corresponding to identifier `#"1"` is selected to be the current value for the popup menu.

Listing 18-6 Implementing a popup menu

```
let depth-popup = make(<ctl-mgr-popup>,
                            identifier: #"popup",
                            location: point(32 - 4, 85),
                            extent: point(180, 20),
                            title: "Bits per pixel:",
                            item-list: list(pair("1", #"1"),
                                            pair("2", #"2"),
                                            pair("4", #"4"),
                                            pair("8", #"8"),
                                            pair("16", #"16"),
                                            pair("32", #"32")));
    depth-popup.control-value := select(*last-depth*)
                                   1 => #"1";
                                   2 => #"2";
                                   4 => #"4";
                                   8 => #"8";
                                   16 => #"16";
                                   32 => #"32";
                                 end select;

    add-sub-view(the-window.root-view, depth-popup);
```

Listing 18-7 shows how the current choice is determined.

Listing 18-7 Determining the current choice ID

```
*last-depth* := select(current-choice-id(depth-popup))
                          #"1" => 1;
                          #"2" => 2;
                          #"4" => 4;
                          #"8" => 8;
                          #"16" => 16;
                          #"32" => 32;
                          otherwise => #f;
                        end select;
```

Adding a Tabbing Behavior

Because the dialog box contains several fields, it is useful to allow the tab and shift-tab characters to change the target view. The tabbing behavior does this, as shown in Listing 18-8.

Listing 18-8 Adding tabbing behavior

```
add-behavior(the-window.root-view, make(<tabber>));
```

Opening the Modal Dialog Box

As with any window, you must set the target view if you do not want it to be the root view. You must open the window to make it visible onscreen. You then call `pose-modally` to cause the window to behave like a dialog box. In the call to `pose-modally` you specify the default and cancel items. The default item responds to the return and enter keys. The cancel item responds to the escape key.

The `pose-modally` function returns the item that dismissed the dialog. If the key is the cancel key, you must ignore any settings to the control and revert to the previous values. The easiest way to do this is to call the `abort` method, which immediately executes the cleanup section of a block. If the key is the OK key, you may set the values and terminate the block normally. The cleanup section is still executed.

Listing 18-9 shows the code that sets the target view and then opens, poses, and closes the dialog box.

Listing 18-9 Posing a Dialog Box

```
the-window.target-view := height-text;
       open(the-window);

       block()
         let dismisser = pose-modally
                (the-window, default: ok-button, cancel: cancel-button);

         if (dismisser = cancel-button)
           abort();
         end if;

         *last-height* := current-value(height-text);
         *last-width* := current-value(width-text);
         *last-depth* := …

         values(*last-height*, *last-width*, *last-depth*);
       cleanup
         close(the-window);
       end block;
```

Dialog and Control Objects Reference

The following sections describe the classes and indentify the functions that can be used with dialogs and controls.

The <dialog-behavior> Class

The Framework provides the `<dialog-behavior>` class to implement a modal dialog for a window. You do not need to define subclasses from the `<dialog-behavior>` class or create `<dialog-behavior>` objects. The Framework

creates the behavior object for you and attaches it to your window when you call the `pose-modally` function; it removes the behavior from the window when it is no longer needed. The window typically contains buttons for the default and cancel items, such as buttons representing "OK" and "Cancel."

Listing 18-10 shows the class definition for the `<dialog-behavior>` class.

Listing 18-10 The `<dialog-behavior>` class definition

```
define primary open class <dialog-behavior> (<behavior>)
  slot default-item,
     init-value: #f,
     init-keyword: default-item:;
  slot cancel-item,
     init-value: #f,
     init-keyword: cancel-item:;
  slot dismisser,
     init-value: #f;
end class;
```

Slot descriptions

`default-item`	The `<ctl-mgr-button>` object that represents the default button, which is the one that typically responds to the `return` or `enter` keys. Use the `default-item:` keyword to specify the button; otherwise, the value is false `(#f)`, meaning the dialog does not have a default button.
`cancel-item`	The `<ctl-mgr-button>` object that represents the cancel button, which is the one that typically responds to the `escape` key or Command-period. Use the `cancel-item:` keyword to specify the Escape button; otherwise, the value is false `(#f)`, meaning the dialog does not have a cancel button.
`dismisser`	The object that caused the window to which the behavior is attached to be dismissed.

Initialization

No `initialize` method is defined for `<dialog-behavior>` objects.

The <tabber> Class

The `<tabber>` class implements tabbing between views, which typically are the views in a window for a dialog box. You do not need to define subclasses of the `<tabber>` class; you create a `<tabber>` object and add it as a behavior to the view whose subviews support tabbing. For example, if you want the user to press the `tab` or `shift-tab` keys to move between all fields in a dialog box, you add the `<tabber>` object to the window's root view.

The behavior handles the `tab` and `shift-tab` characters by highlighting and setting the target to the next or previous view, respectively, in the view's list of subviews to which the behavior is attached. Only a view that can be a target, wants to be the target, and is visible is allowed to become the target. Processing is circular; for example, tabbing out of the last field moves the highlight and target to the first view in the list. Using the shift-tab key to move from the first subview moves the highlight and target to the last subview in the list.

You also may specify recursive tabbing behavior, which means that all subviews of a view are processed, as described in the previous paragraph, before moving to the next view in the list.

Listing 18-11 shows the class definition for the `<tabber>` class.

Listing 18-11 The `<tabber>` class definition

```
define primary open class <tabber> (<behavior>)
  slot first-target,
     init-value: #f;
  slot next-target,
     init-value: #f;
  slot found-current-target,
     init-value: #f;
  slot search-recursively,
     init-value: #f,
     init-keyword: recursive:;
end class;
```

Slot descriptions

`first-target`	The first view eligible to be the target. Internal.
`next-target`	The next view eligible to be the target. Internal.

`found-current-target`
: Specifies whether the view actually found matches the next view. Internal.

`search-recursively`
: Specifies whether to search recursively (`#t`), or not (`#f`). Use the `recursive:` keyword to specify true (`#t`) if you want to support tabbing recursively through each view's list of views; otherwise, the value is false (`#f`), meaning that the tabbing behavior only operates on the subviews of the view to which the behavior is attached.

Initialization

No `initialize` method is defined for `<tabber>` objects.

The <edit-text-dialog-filter> Class

The Framework provides an `<edit-text-dialog-filter>` behavior class that prevents the `Enter`, `Return`, `Escape`, and `Tab` keys from being part of the text and allows these keys to control the dialog. By default, the Framework creates an `<edit-text-dialog-filter>` object and adds it to each `<edit-text>` object. Thus, editable fields or number fields within a dialog work as expected. If you are using these fields in a dialog box, you need do nothing. If you do not want this behavior for your `<edit-text>` object, you specify `filter-control-chars: #f` when you create your `<edit-text>` object. For more information about the `<edit-text>` object, see "The <edit-text> Class" on page 378. You do not need to define a subclass of `<edit-text-dialog-filter>` or create objects from the `<edit-text-dialog-filter>` class.

Listing 18-12 shows the class definition for the `<edit-text-dialog-filter>` class.

Listing 18-12 The `<edit-text-dialog-filter>` class definition

```
define primary open class <edit-text-dialog-filter> (<behavior>)
  constant slot control-chars :: <list>,
     init-value:
          list(as(<character>, 3),  // enter
               '\n',                // return
               as(<character>, 27), // escape
```

```
                '\t'                    // tab
        );
end class;
```

Slot descriptions

`control-char` The keys (`Enter`, `Return`, `Escape`, and `Tab`) that are not considered part of the editable text, rather they are used to terminate the dialog or move to the next field.

Initialization

No `initialize` method is defined for `<edit-text-dialog-filter>` objects.

The <cluster> Class

The `<cluster>` class allows controls, such as radio buttons or checkboxes, to be grouped together. The controls in the cluster must be subviews of the cluster and have the cluster as their superview. You do not need to define subclasses of the `<cluster>` class; you only need to create a `<cluster>` object for each group of controls that you want to treat as a single unit.

Listing 18-13 shows the class definition for the `<cluster>` class.

Listing 18-13 The `<cluster>` class definition

```
define primary open class <cluster> (<view>)
  slot title :: <string>,
     init-value: "",
     init-keyword: title:;

  inherited slot transparent?,
     init-value: #t;
end class;
```

Slot descriptions

`title` The view's title. Use the `title:` keyword to specify the title; otherwise, the slot is empty. Use the getter and setter functions to access this slot.

`transparent?` Specifies whether or not content drawn in the view allows overlapping content to be seen, (`#t`), or whether the content being drawn obscures overlapping content, (`#f`). Use the `transparent:` keyword and specify false (`#f`) if you want to prevent transparent drawing; otherwise, the value of this slot is true (`#t`), meaning that drawing in a view does not obscure what has been drawn already. You can use the getter and setter to access this slot.

Initialization

No `initialize` method is defined for `<cluster>` objects.

The <simple-icon> Class

The `<simple-icon>` class represents an icon in a dialog box. The icon must be a resource. You do not need to define subclasses of the `<simple-icon>` class; you only need to create a `<simple-icon>` object for each icon resource. Listing 18-14 shows the class definition for the `<simple-icon>` class.

Listing 18-14 The `<simple-icon>` class definition

```
define primary open class <simple-icon> (<view>)
  slot resource-id :: <integer>,
     required-init-keyword: resource-id:;
end class;
```

Slot descriptions

`resource-id` The resource ID of the resource that contains the icon. You must use the `resource-id:` and specify the resource id for your icon.

Initialization

No `initialize` method is defined for `<simple-icon>` objects.

The <static-text> Class

The <static-text> class represents text that you want to display in a dialog box yet cannot be changed. You do not need to define subclasses of the <static-text> class; you only need to create a <static-text> object for each text string. Listing 18-15 shows the class definition for the <static-text> class.

Listing 18-15 The <static-text> class definition

```
define primary open class <static-text> ( <view> )
  slot text :: <string>,
    init-value: "",
    init-keyword: text:;
  slot %justification :: <symbol>,
    init-value: #"left",
     init-keyword: justification:;
  slot truncate? :: <boolean>,
     init-value: #f,
     init-keyword: truncate:;
  inherited slot invalidate-all-on-resize?,
     init-value: #t;
end class <static-text>;
```

Slot descriptions

text	The text to be displayed in the view. Use the text: keyword to specify the text; otherwise, an empty string is displayed. Use the getter and setter functions to access this slot.
%justification	Specifies how the text is to be justified in the view. Use the justification: keyword to specify the justification as either #"left", #"right", or #"center"; otherwise, the value is left (#"left"). You can use the justification and justification-setter methods to determine or set the justification. The %justification is not exported.
truncate?	Specifies whether or not to truncate text that is too wide to fit within the view (#t), or not, (#f). Use the truncate: keyword to specify true (#t) if you want to truncate the text; otherwise, the value of this slot is false (#f), meaning

that text that is too long is broken into separate lines. You can use the getter and setter to access this slot.

`invalidate-all-on-resize?`
: Specifies whether or not the entire view is invalidated when the view's size changes (`#t`), or not, (`#f`). Use the `invalidate-all-on-resize:` keyword and specify false (`#f`) if you do not want to redraw the entire view; otherwise, the value of this slot is true (`#t`), meaning that all the text is redrawn whenever the view's size changes. You can use the getter and setter to access this slot.

Initialization

The `initialize` method for `<button>` objects sets up the button to be a default button. The `initialize` method's parameters are defined as follows:

```
define method initialize (static :: <static-text>,
                          #key text-style: the-style) => ()
```

`static` The static text view.

`the-style` The style for text in the view. Use the `text-style:` keyword to specify the style; otherwise, the style is 12-point Chicago normal.

The <control> Class

The `<control>` class is the superclass of controls, such as buttons, checkboxes, and so on. You do not create `<control>` objects or specialize parameters on them. You only need to define a subclass of the `<control>` class if you are creating your own kind of control; the Framework provides Control Manager controls for you.

Listing 18-16 shows the class definition for the `<control>` class.

Listing 18-16 The <control> class definition

```
define primary open class <control> (<view>)
    slot title :: <string>,
        init-value: "",
        init-keyword: title:;
    slot dismisses-dialog? :: <boolean>,
        init-value: #f,
        init-keyword: dismisses-dialog:;
    slot control-mode :: <symbol>,
        init-keyword: control-mode:,
        init-value: #"no-mode";
    slot %dimmed? :: <boolean>,
        init-value: #f,
        init-keyword: dimmed:;
    slot %hilited? :: <boolean>,
        init-value: #f,
        init-keyword: hilited:;
    slot %on? :: <boolean>,
        init-value: #f,
        init-keyword: on:;
  end class;
```

Slot descriptions

`title`	The control's title. Use the `title:` keyword to specify the title; otherwise, the slot is empty. Use the getter and setter functions to access this slot.
`dismisses-dialog?`	Specifies whether the control can dismiss the window it is in (`#t`), or not, (`#f`). Use the `dismisses-dialog:` keyword to specify true (`#t`) if you want the control to dismiss the dialog box; otherwise, the value of this slot is false (`#f`), meaning that the dialog box remains onscreen after the control is activated. You can use the getter and setter to access this slot.
`control-mode`	Specifies the way in which highlighting is to be performed for non-Control Manager controls. Use the `control-mode:` keyword to specify to specify the control mode; otherwise, the value of this slot is `#"no-mode"`, meaning that the control appears as normal for Control Manager controls.

	You can use the getter and setter to access this slot. For acceptable values for this slot, see Table 18-1 on page 424.
`%dimmed?`	Specifies whether the control appears to be dim (`#t`), or not, (`#f`). Use the `dimmed:` keyword to specify true (`#t`) if you want the control to appear dimmed; otherwise, the value of this slot is false(`#f`), meaning that the control appears as normal. You can use the `dimmed?` and `dimmed?-setter` to access this slot.
`%hilited?`	Specifies whether the control appears highlighted (`#t`), or not, (`#f`). Use the `hilited:` keyword to specify true (`#t`) if you want the control to highlighted; otherwise, the value of this slot is false(`#f`), meaning that the control appears as normal. You can use the `hilited?` and `hilited?-setter` to access this slot. This slot is applicable to buttons.
`%on?`	Specifies whether the control is on (`#t`), or off, (`#f`). It applies to check boxes, radio buttons, or other controls with two states, such as "checked" or "unchecked," in the case of a checkbox. Use the `on:` keyword to specify true (`#t`) if you want the control to be on; otherwise, the value of this slot is false(`#f`), meaning that the control is in its "off" state. You can use the `on?` and `on?-setter` to access this slot. This slot is applicable to buttons.

Initialization

No `initialize` method is defined for `<control>` objects.

Subclasses of the <control> Class

The Framework provides the following subclasses of the `<control>` class:

- the `<button>` class
- the `<check-box>` class
- the `<radio>` class
- the `<scroll-bar>` class
- the `<pop-up>` class

The following sections describe these subclasses.

The <button> Class

The `<button>` class provides Control Manager-style buttons. You do not create `<button>` objects or specialize parameters on this class. You typically create objects from the `<ctl-mgr-button>` class and specialize parameters on those objects instead. If you wanted to define a new kind of button, you could define a subclass of the `<button>` class and mix the subclass in with the `<ctl-mgr-control>` class.

Listing 18-17 shows the class definition for the `<button>` class.

Listing 18-17 The `<button>` class definition

```
define primary open class <button> (<control>)
end class;
```

Initialization

No `initialize` method is defined for `<button>` objects.

The <check-box> Class

The `<check-box>` class provides Control Manager-style checkboxes. You do not create `<check-box>` objects or specialize parameters on this class. You typically create objects from the `<ctl-mgr-check-box>` class and specialize parameters on those objects instead. If you wanted to define a new kind of check box, you could define a subclass of the `<check-box>` class and mix the subclass in with the `<ctl-mgr-control>` class.

Listing 18-18 shows the class definition for the `<check-box>` class.

Listing 18-18 The `<check-box>` class definition

```
define class <check-box> (<control>)
end class;
```

The <radio> Class

The `<radio>` class provides Control Manager-style radio buttons. You do not create `<radio>` objects or specialize parameters on this class. You typically

create objects from the <ctl-mgr-radio> class and specialize parameters on those objects instead. If you wanted to define a new kind of radio button, you could define a subclass of the <radio> class and mix the subclass in with the <ctl-mgr-control> class.

Listing 18-19 shows the class definition for the <radio> class.

Listing 18-19 The <radio> class definition

```
define primary open class <radio> (<control>)
end class;
```

The <scroll-bar> Class

The <scroll-bar> class provides Control Manager-style scroll bars. You do not create <scroll-bar> objects or specialize parameters on this class. You create objects from the <ctl-mgr-scroll-bar> class and specialize parameters on those objects instead. If you wanted to define a new kind of scroll bar, you could define a subclass of the <scroll-bar> class and mix the subclass in with the <ctl-mgr-control> class.

Listing 18-20 shows the class definition for the <scroll-bar> class.

Listing 18-20 The <scroll-bar> class definition

```
define primary open class <scroll-bar> (<control>)
  slot scroll-increment :: <integer>,
     init-value: 1,
     init-keyword: scroll-increment:;
  slot page-increment :: <integer>,
     init-value: 1,
     init-keyword: page-increment:;
end class;
```

Slot descriptions

`scroll-increment` The number of pixels to scroll if the user clicks in the up or down arrow boxes in the scroll bar. Use the `scroll-increment:` keyword to specify the number of

pixels to move in either direction; otherwise, clicking in an arrow box only scrolls a view by one pixel. Use the getter and setter functions to access this slot.

`page-increment` The number of pixels to scroll if the user clicks in the page-up or page-down areas of a scroll bar. Use the `page-increment:` keyword to specify the number of pixels to move in either direction; otherwise, clicking in these areas only scrolls a view by one pixel. Use the getter and setter functions to access this slot.

Initialization

No `initialize` method is defined for `<scroll-bar>` objects.

The <popup> Class

The `<popup>` class provides Control Manager-style popup menus. You do not define subclasses of the `<popup>` class, create `<popup>` objects, or specialize parameters on this class. You create objects from the `<ctl-mgr-popup>` class and specialize parameters on those objects instead.

Listing 18-21 shows the class definition for the `<popup>` class.

Listing 18-21 The `<popup>` class definition

```
define primary open class <popup> (<control>)
   slot dynamic-menu? :: <boolean>,  // true if the contents of the menu
is determined at runtime
    init-value: #f,
    init-keyword: dynamic-menu:;
 end class;
```

Slot descriptions

`dynamic-menu?` Specifies whether the menu items for a popup menu are created at runtime (`#t`), or whether they are loaded from a resource, (`#f`). Use the `dynamic-menu:` keyword to specify true (`#t`) if you specify the menu items at runtime; otherwise, the value of this slot is false (`#f`), meaning that the items are loaded from a resource. You can use the getter and setter to access this slot. You do not need to

change this slot if your pop-up menu is a <ctl-mgr-popup> object.

Initialization

No initialize method is defined for <popup> objects.

Control Manager Control Classes

Control Manager control classes are derived from a subclass of the <control> class and the <ctl-mgr-control> class. The following sections describe these classes.

The <ctl-mgr-control> Class

The <ctl-mgr-control> class implements the Control Manager interface for control views. You do not need to define subclasses of or create objects from the <ctl-mgr-control> class. You may specialize a parameter on this class if you want a method to apply to all kinds of Control Manager controls.

Listing 18-22 shows the class definition for the <ctl-mgr-control> class.

Listing 18-22 The <ctl-mgr-control> class definition

```
define free open class <ctl-mgr-control> (<control>)
  slot control-handle :: <ControlHandle>,
    init-value: as(<ControlHandle>, 0);
  slot ref-con :: <integer>, init-value: 0;
  slot proc-id :: <integer>, init-value: -1,
     init-keyword: proc-id:;
  slot action-proc, init-value: #f;
  slot uses-color?, init-value: #f,
     init-keyword: uses-color?: ;

  inherited slot invalidate-all-on-resize?,
     init-value: #t;

  // private
  slot %min :: <integer>,
```

```
     init-value: 0,
     init-keyword: min:;
  slot %max :: <integer>,
     init-value: 1,
     init-keyword: max:;
  slot %value :: <integer>,
     init-value: 0,
     init-keyword: value:;
end class;
```

Slot descriptions

`control-handle` The Control Manager `ControlHandle` record associated with the control.

`ref-con` A reference constant stored in the `ControlHandle` record. The Framework uses this slot to maintain information about popup menus and scroll bars.

`proc-id` The control definition ID, which determines which control definition function to use with the control. The Framework defines this value for its subclasses of `<ctl-mgr-control>`. Use the `proc-id:` keyword to specify the control definition ID if you define a subclass of `<ctl-mgr-control>` so that you can create a custom control; otherwise, the value of this slot is -1 for your subclass. You can use the getter and setter to access this slot. For more information about control definition IDs and functions, see the Control Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

`action-proc` The action procedure that follows and responds to cursor movements in a control. Use the `action-proc:` keyword to specify the routine descriptor for the action procedure; otherwise, the value of this slot is false (`#f`), meaning that the Control Manager's standard actions are taken.

`uses-color?` Specifies whether the control uses colors other than the default one used by the system software. Use the `uses-color?:` keyword to specify that non-default system colors may be used (`#t`); otherwise, the value of this slot is false (`#f`), meaning that a control color table is not used even if it is available.

`invalidate-all-on-resize?`
: Specifies whether or not the entire control is invalidated when the view's size changes (`#t`), or not, (`#f`). Use the `invalidate-all-on-resize:` keyword and specify false (`#f`) if you do not want to redraw the entire view; otherwise, the value of this slot is true(`#t`), meaning that the entire control is redrawn whenever the view's size changes. You can use the getter and setter to access this slot.

`%min`
: The minimum value for the control. Use the `min:` keyword and specify a value that represents the minimum value; otherwise, the value of this slot is 0. You can use the `scroll-min` and `scroll-min-setter` methods to access this slot for scroll bars. You can use the `control-min` and `control-min-setter` methods to access this slot for other kinds of controls. The `%min` slot is not exported.

`%max`
: The maximum value for the control. Use the `max:` keyword and specify a value that represents the maximum value; otherwise, the value of this slot is 1. You can use the `scroll-max` and `scroll-max-setter` methods to access this slot for scroll bars. You can use the `control-max` and `control-max-setter` methods to access this slot for other kinds of controls. The `%max` slot is not exported.

`%value`
: The current state of the control; for example a 0 if a check box or radio button is not marked or 1 if it is marked. Use the `value:` keyword and specify a value that represents the state; otherwise, the value of this slot is 0. You can use the `control-value` and `control-value-setter` methods to access this slot. The `%value` slot is not exported.

Initialization

No `initialize` method is defined for `<ctl-mgr-control>` objects.

The <ctl-mgr-button> Class

The `<ctl-mgr-button>` class implements Control Manager push buttons. You do not need to define subclasses from the `<ctl-mgr-button>` class. You create a `<ctl-mgr-button>` object for each button and specialize parameters on this class so that your `<ctl-mgr-button>` objects respond to events.

Listing 18-23 shows the class definition for the `<ctl-mgr-button>` class.

Listing 18-23 The <ctl-mgr-button> class definition

```
define primary open class <ctl-mgr-button> (<ctl-mgr-control>, <button>)
  inherited slot proc-id,
     init-value: $pushButProc;
end class;
```

Slot descriptions

`proc-id` The control definition ID, which determines which control definition function to use with the control. It is `$pushButProc`, which is the Control Manager's control definition ID for buttons.

Initialization

The `initialize` method for `<button>` objects sets up the button to be a default button. The `initialize` method is defined as follows:

```
define method initialize (button :: <ctl-mgr-button>,
                          #key default: default) => ()
```

`button` The Control Manager button.

`default` Specifies whether the button is a "default" button `(#t)`, which is the one that typically responds to the `return` key, or a non-default button `(#f)`. Use the `default:` keyword to specify true `(#t)` if this is a default button; otherwise, the value is false `(#f)`, meaning that the wide border around the button is not drawn.

The <ctl-mgr-check-box> Class

The `<ctl-mgr-check-box>` class implements Control Manager checkboxes. You do not need to define subclasses from the `<ctl-mgr-check-box>` class. You create a `<ctl-mgr-check-box>` object for each checkbox and specialize parameters on this class so that your `<ctl-mgr-check-box>` objects respond to events.

Listing 18-24 shows the class definition for the `<ctl-mgr-check-box>` class.

Listing 18-24 The <ctl-mgr-check-box> class definition

```
define primary open class <ctl-mgr-check-box>
                                        (<ctl-mgr-control>, <check-box>)
  inherited slot proc-id,
     init-value: $checkBoxProc;
end class;
```

Slot descriptions

proc-id — The control definition ID, which determines which control definition function to use with the control. It is $checkBoxProc, which is the Control Manager's control definition ID for checkboxes.

Initialization

No initialize method is defined for <ctl-mgr-check-box> objects.

The <ctl-mgr-radio> Class

The <ctl-mgr-radio> class implements Control Manager radio buttons. You do not need to define subclasses from the <ctl-mgr-radio> class. You create a <ctl-mgr-radio> object for each button and specialize parameters on this class so that your <ctl-mgr-radio> objects respond to events.

Listing 18-25 shows the class definition for the <ctl-mgr-radio> class.

Listing 18-25 The <ctl-mgr-radio> class definition

```
define primary open class <ctl-mgr-radio> (<ctl-mgr-control>, <radio>)
  inherited slot proc-id,
     init-value: $radioButProc;
end class;
```

Slot descriptions

proc-id — The control definition ID, which determines which control definition function to use with the control. It is $radioBoxProc, which is the Control Manager's control definition ID for radio buttons.

Initialization

No `initialize` method is defined for `<ctl-mgr-radio>` objects.

The <ctl-mgr-scroll-bar> Class

The `<ctl-mgr-scroll-bar>` class implements Control Manager scroll bars. You do not need to define subclasses from the `<ctl-mgr-scroll-bar>` class. You create a `<ctl-mgr-scroll-bar>` object for each scroll bar and specialize parameters on this class so that your `<ctl-mgr-scroll-bar>` objects respond to events.

Listing 18-26 shows the class definition for the `<ctl-mgr-scroll-bar>` class.

Listing 18-26 The `<ctl-mgr-scroll-bar>` class definition

```
define primary open class <ctl-mgr-scroll-bar>
                                        (<ctl-mgr-control>, <scroll-bar>)
  inherited slot proc-id,
     init-value: $scrollBarProc;
  slot hide-on-deactivate? :: <boolean>,
     init-value: #t,
     init-keyword: hide-on-deactivate:;
 end class;
```

Slot descriptions

`proc-id` The control definition ID, which determines which control definition function to use with the control. It is `$scrollBarProc`, which is the Control Manager's control definition ID for scroll bars.

`hide-on-deactivate?` Specifies whether the scroll bar is hidden when the scroll bar is not active `(#t)`, or not hidden when the scroll bar is not active `(#f)`. Use the `hide-on-deactivate:` keyword and specify false `(#f)` if you want the scroll bar to be drawn when it is not active; otherwise, the value is true `(#t)`, meaning that a framed rectangle is drawn in place of the scroll bar. You can use the getter and setter to access this slot.

Initialization

The `initialize` method for `<ctl-mgr-scroll-bar>` objects sets up a control action procedure for the scroll bar. The `initialize` method's parameters are defined as follows:

```
define method initialize (bar :: <ctl-mgr-scroll-bar>, #key) => ()
```

`bar` The scroll bar.

The <ctl-mgr-popup> Class

The `<ctl-mgr-popup>` class implements Control Manager popup menus. You do not need to define subclasses from the `<ctl-mgr-popup>` class. You create a `<ctl-mgr-popup>` object for each menu and specialize parameters on this class so that your `<ctl-mgr-popup>` objects respond to events.

Listing 18-27 shows the class definition for the `<ctl-mgr-popup>` class.

Listing 18-27 The `<ctl-mgr-popup>` class definition

```
define primary open class <ctl-mgr-popup> (<ctl-mgr-control>, <popup>)
  inherited slot proc-id,
     init-value: $popupMenuProc;

  slot menu :: <menu>;

  slot popup-style :: <integer>,
    init-value: $popupTitleLeftJust,
    init-keyword: popup-style:;

  slot res-menu-type,
    init-value: #f,
    init-keyword: res-menu-type:;

end class;
```

Slot descriptions

`proc-id` The control definition ID, which determines which control definition function to use with the control. It is

	`$popupMenuProc`, which is the Control Manager's control definition ID for popup menus.
`menu`	The menu object associated with the popup menu, which maintains the popup's menu items. Internal.
`popup-style`	The style of the popup menu's title. Use the `popup-style:` keyword to specify the title; otherwise, the value is `$popupTitleLeftJust`, meaning that the title is placed to the left of the popup menu. You can use the getter and setter to access this slot. For more information about control title styles, see the Control Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.
`res-menu-type`	The kind of menu item resources to add to the popup menu.

Initialization

The `initialize` method for `<ctl-mgr-popup>` objects loads the menu items in a popup menu either dynamically or from a resource. Use the `item-list:` keyword to load the items dynamically. Use the `resource:` and `library:` keywords if you want to load the menu items from a resource. The `initialize` method's parameters are defined as follows:

```
define method initialize (popup :: <ctl-mgr-popup>,
                          #key    item-list: item-list,
                                  resource: resource,
                                  library: lib) => ()
```

`popup`	The Control Manager popup.
`item-list`	The list of items in a dynamically loaded popup menu. Use the `item-list:` keyword to specify the list of items.
`resource`	The resource ID of the popup menu. Use the `resource:` keyword to specify the resource ID.
`lib`	The library that contains the popup menu's resources. Use the `library:` keyword to specify the library.

You must use both the `resource:` and `library:` keywords together if you want to load the menu items from a resource. If you are loading from a resource, do not specify the `item-list:` keyword.

Control Event Classes

Control event classes represent events that operate on controls. You do not need to define subclasses of a control event class. You may, if you wish, respond to a control event and make another one from it. Typically, however, you respond to control events by specializing parameters of the `do-event` and `behavior-event` methods on the control event and the Control Manager control view object that responds to the event.

Listing 18-28 shows the class definition for the `<control-event>` class and its subclasses.

Listing 18-28 The `<control-event>` class and subclass definitions

```
define primary open class <control-event> (<event>)
  slot control :: <control>, init-value: #f, init-keyword: control:;
  slot sender :: <object>, init-value: #f, init-keyword: sender:;
end class;

define primary open class <button-event> (<control-event>)
end class;

define primary open class <check-box-event> (<control-event>)
end class;

define primary open class <radio-event> (<control-event>)
end class;

define primary open class <scroll-bar-event> (<control-event>)
end class;

define primary open class <popup-event> (<control-event>)
end class;
```

Slot descriptions

`control` The control that initiated the event, such as a `<ctl-mgr-control>` object that responded to a `<mouse-down>` event.

`sender`	The object that made the event, such as a `<ctl-mgr-control>` object or a behavior.

Initialization

No `initialize` method is defined for control event objects.

The <default-button-adorner> Class

The `<default-button-adorner>` class provides a 3-pixel wide border drawn as a rounded rectangle inside the view to which its is attached. You do not need to define subclasses of the `<default-button-adorner>` class, nor do you need to create objects from this class. The Framework sets up the adorner for you if you specify `default: #t` when you create your `<button>` object. For more information about the `<button>` class, see "The <button> Class" on page 443.

Listing 18-29 shows the class definition for the `<default-button-adorner>` class.

Listing 18-29 The `<default-button-adorner>` class definition

```
define primary sealed class <default-button-adorner> (<adorner>)
  inherited slot identifier,
     init-value: #"default-button";
end class;
```

Slot descriptions

`identifier`	The identifier for a `<default-button-adorner>` object.

Initialization

No `initialize` method is defined for `<default-button-adorner>` objects.

Functions for Dialogs and Controls

This section identifies the functions you can use to implement dialogs and controls. Table 18-2 shows the functions you can use with controls in general.

Table 18-2 Functions for Control Manager controls

Function	Purpose	Call	Sp.
`draw`	Draw a control. (Only used if you define a subclass of a `<control>` class.		√
`control-min`	Determine or set the control's minimum value.	√	
`control-max`	Determine or set the control's maximum value.	√	
`control-value`	Determine or set the control's current value.	√	
`set-ctl-mgr-color`	Set a control's color table so that the control can be drawn in a nondefault system color.	√	

Table 18-3 shows functions you can use with scroll bars.

Table 18-3 Functions for Control Manager scroll bar controls

Function	Purpose	Call	Sp.
`activate`	Activate the scroll bar.		√
`scroll-min`	Determine or set the scroll bar's minimum value.	√	
`scroll-max`	Determine or set the scroll bar's maximum value.	√	

Table 18-3 Functions for Control Manager scroll bar controls

Function	Purpose	Call	Sp.
`scroll-value`	Determine or set the scroll bar's current value.	√	
`do-scroll`	Specifies how to scroll, typically by defining a new current value.		√
`do-scroll-action`	Specifies a Control Manager scrolling action procedure.		√

Table 18-4 shows functions you can use with popup menus.

Table 18-4 Functions for Control Manager popup menu controls

Function	Purpose	Call	Sp.
`current-choice-item`	Determine the menu item currently selected in the popup menu.	√	
`current-choice-id`	Determine the ID of the event handler associated with the currently selected in the popup menu.	√	
`current-choice-string`	Determine the title of the menu item currently selected in the popup menu.	√	

CHAPTER 19

Scrolling and Tracking

Contents

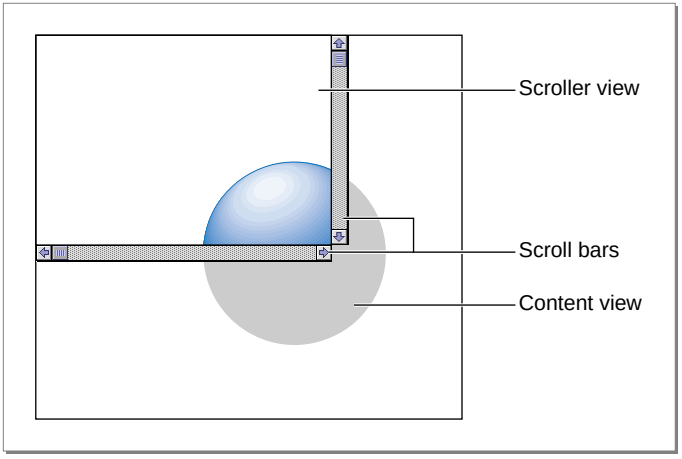
About Scrolling 461
About Tracking 463
Using Scrolling and Tracking 465
  Setting up a Scrolling View 465
    Setting up a Scroller View 466
    Setting up Scroll Bars 467
  Implementing Mouse Tracking 467
Scrolling and Tracking Objects Reference 471
  The <scroller> Class 471
  The <tracker> Class 473
  The <tracker-adorner> Class 474
  Functions for Scrolling and Tracking 475

This chapter discusses how to set up scrolling views and how to track mouse movements across a view. For information about scroll bars, see “The <ctl-mgr-scroll-bar> Class” on page 451.

About Scrolling

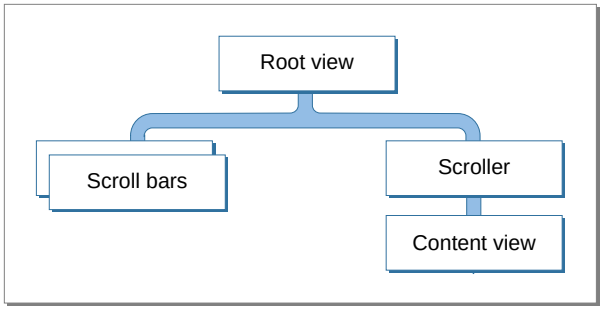
Scrolling provides the ability to manage a view, only part of which is visible through a window. Figure 19-1 shows a view whose contents are too large to entirely display in the window.

Figure 19-1 The relationship between a scroller view, scroll bars, and a content view



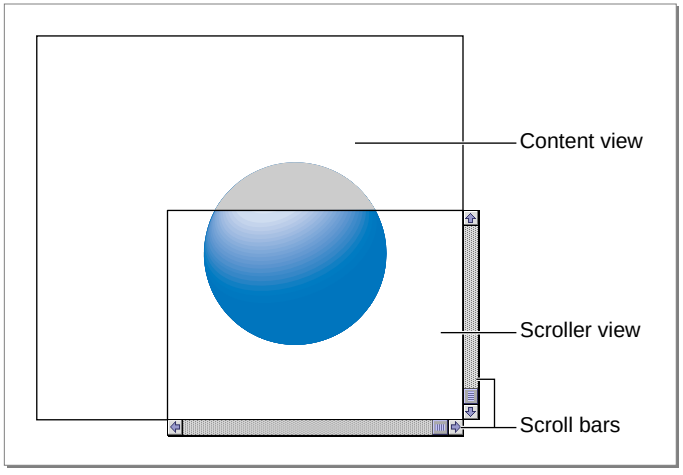
The content view is a subview of the scroller. The scroller and scroll bars are subviews of the root view, as shown in Figure 19-2.

Figure 19-2 Scrolling view hierarchy



When a mouse event occurs in the scroll bar, the scroller translates its position in relation to the its subviews. Thus, you may want to think of the scroller "moving" to expose a different area of its subviews in response to mouse-down events in a scroll bar. Figure 19-3 shows the visible part of the content view after mouse-down events have occurred.

Figure 19-3 The translation between a content view and a scroller



Note
The relationship between the scroll bars and scroller do not change as a result of mouse-down events in a scroll bar; they change only in relation to a change in the root view, such as the result of a change in the size or position of the root view's window. ◆

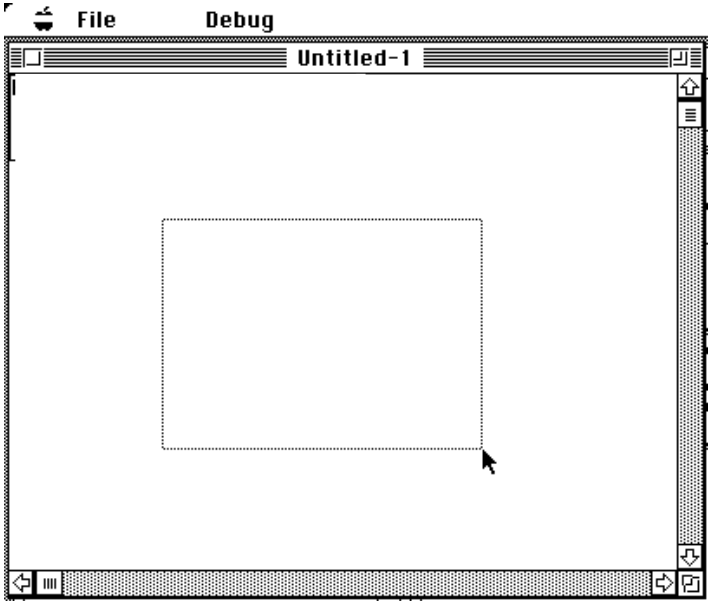
The Framework handles the translation between the content view and the scroller for you and also handles mouse down events in a scroll bar. You are responsible for setting up a view hierarchy, similar to the one shown in Figure 19-2, and for specifying the scroll increments, which controls how many pixels to "move" the scroller in response to an event in the arrow or the bar part of the scroll bar.

For an example that implements a scrolling view, see "Setting up a Scrolling View" on page 465. For information about the `<scroller>` class, see "The <scroller> Class" on page 471.

About Tracking

Tracking provides feedback about mouse movements while the mouse is down. Tracking is typically provided for mouse movements over content views. For example, a graphics program could provide tracking feedback to show the shape of an object that will be drawn when the mouse is released. Figure 19-4 shows an example of feedback for a drawing a rectangle.

Figure 19-4 Mouse tracking feedback



Mouse tracking is provided by a `<tracker>` class, from which you can provide subclasses. In addition to tracking the mouse and returning its the position when the button is released, the Framework allows you to constrain the movement of the mouse; for example, by requiring the feedback to be a square instead of a rectangle. In this example, the position when the mouse is released would be a corner of a square, regardless of whether the mouse was moved further to the left or right than up or down, and so on.

For an example of that implements mouse tracking, see “Implementing Mouse Tracking” on page 467. For information about the `<tracker>` class, see “The <tracker> Class” on page 473.

Using Scrolling and Tracking

The following sections show how to set up a view that scrolls and a tracker for mouse movements within a view.

Setting up a Scrolling View

In general, you can call `make-scrolling-window` to set up a window that supports scrolling. If you want to support scrolling in some other way, such as by splitting a window and supporting scrolling within each part, you can use the `make-scrolling-window` method as a model. Listing 19-1 shows how the method is implemented.

Listing 19-1 Setting up a scrolling view

```
define constant make-scrolling-window =
    method (view-list :: <list>, #rest window-args)
            => result :: <window>;
        let the-window = apply(make, <window>, window-args);

        let root = the-window.root-view;

        let scroller = make(<scroller>, identifier: #"scroller",
                        extent: point(root.extent.h - $scroll-bar-size,
                                      root.extent.v - $scroll-bar-size));
        add-sub-view(root, scroller, update-caches: #f);
        add-determiner(scroller, make(<super-view-relative-determiner>));

        let h-scroll = make(<ctl-mgr-scroll-bar>, identifier: #"h-bar",
                location: point(-1, root.extent.v - $scroll-bar-size),
                extent: point(root.extent.h - $scroll-bar-size + 2, 16),
                scroll-increment: 32);
        add-sub-view(root, h-scroll, update-caches: #f);
        add-determiner(h-scroll, make(<horiz-scroll-bar-determiner>));
```

```
    let v-scroll = make(<ctl-mgr-scroll-bar>, identifier: #"v-bar",
            location: point(root.extent.h - $scroll-bar-size, -1),
            extent: point(16, root.extent.v - $scroll-bar-size + 2),
            scroll-increment: 32);
    add-sub-view(root, v-scroll, update-caches: #f);
    add-determiner(v-scroll, make(<vert-scroll-bar-determiner>));

    // add all the views in view-list to the scroller
    for (view in view-list)
        add-sub-view(scroller, view, update-caches: #f);
    end for;

    // do this after we add the views.
    attach-to-scroller(h-scroll, scroller);
    attach-to-scroller(v-scroll, scroller);

    // return the window as the result
    the-window;
end method;
```

The following sections describe the major steps, which are setting up the scroller view and setting up the scroll bars.

Setting up a Scroller View

To set up a scroller, take the following actions:

1. Create a `<scroller>` object.
2. Add the scroller as a subview of the root view.
3. Create and add a relative size determiner to the scroller so that the scroller is resized in proportion to the root view whenever the window is resized.
4. Add your content views as subviews of the scroller.

Listing 19-2 shows an example.

Listing 19-2 Setting up a scroller view

```
let scroller = make(<scroller>, identifier: #"scroller",
                extent: point(root.extent.h - $scroll-bar-size,
                        root.extent.v - $scroll-bar-size));
add-sub-view(root, scroller, update-caches: #f);
add-determiner(scroller, make(<super-view-relative-determiner>));
…
for (view in view-list)
    add-sub-view(scroller, view, update-caches: #f);
end for;
```

In this example, the scroller is the same size as the root view, and thus the same size as the window, after reserving space for the scroll bars.

Setting up Scroll Bars

To setup a scroll bar, take the following actions:

1. Create a `<scroll-bar>` object.
2. Add it as a subview of the window's root view.
3. Add the appropriate scroll bar determiner.
4. Attach the scroll bar to the scroller.

Note
You must attach the scroll bar to the scroller after you have added your content subviews. ◆

Implementing Mouse Tracking

Mouse tracking provides visual feedback to the user when the mouse is being moved and the mouse key is down. You call `track-mouse` to initiate mouse tracking, typically from the `do-event` or `behavior-event` method whose parameters are specialized on both the view on which you wish to track mouse movements and a mouse-down event. When the mouse key is released, the `track-mouse` method returns the point at which the mouse-up event occurred. Listing 19-3 shows `track-mouse` being called from a `do-event` method.

Listing 19-3 Initiating mouse tracking

```
define method do-event (view :: <my-view>,
                        event :: <mouse-down-event>,
                        id :: <object>) => ()
    ignore(event, id);
    let final-point = track-mouse(make(<my-tracker>),
                                  view, event.local-mouse);
    let draw-rect = make(<rect>);
    set-rect-from-points(draw-rect, event.local-mouse, final-point);
    view.the-document.data :=
        insert(view.the-document.data, draw-rect, last: #t);
    invalidate(view, draw-rect);
end method;
```

The `track-mouse` method has three parameters:

- the tracker object that manages feedback. You must create this object.
- the view over which mouse movements are tracked. Typically, this is a content view.
- the starting point for tracking the mouse, which typically is the mouse position when the mouse-down event occurred.

It is necessary to define a `<tracker>` subclass if you need additional slots. For example, you may need a slot to maintain information about how to constrain the feedback within the specified bounds even if the mouse moves outside of them. Listing 19-4 shows a subclass defined for this reason.

Listing 19-4 Defining a `<tracker>` subclass

```
define class <my-tracker> (<tracker>)
    slot constraint :: <rect>,
        init-function:
            method()
                rect(0,0,0,0);
            end method;
end class;
```

Mouse tracking is implemented with four methods:

- the `track-begin` method sets up mouse tracking.
- the `track-end` method cleans up mouse tracking.
- the `track-constrain` method keeps the feedback within some bounds.
- the `track-feedback` method provides visual feedback during mouse tracking.

The Framework calls these methods at the appropriate time after you call `track-mouse`. You must define a `track-feedback` method to provide visual feedback and a `track-end` method to terminate the feedback. You may to define `track-begin` method to set up the tracker and a `track-constrain` method to implement a constraint.

Listing 19-5 shows a `track-begin` method that sets up the constraint rectangle to match the extent of the view.

Listing 19-5 Beginning mouse tracking

```
define method track-begin (tracker :: <my-tracker>) => ()
    ignore(tracker);
    tracker.constraint.top
        := tracker.tracker-view.location.v;
    tracker.constraint.left
        := tracker.tracker-view.location.h;
    tracker.constraint.bottom
        := tracker.tracker-view.location.v
         + tracker.tracker-view.extent.v;
    tracker.constraint.right
        := tracker.tracker-view.location.h
         + tracker.tracker-view.extent.h;
end method;
```

Listing 19-6 shows a `track-end` method that turns off feedback.

Listing 19-6 Ending mouse tracking

```
define method track-end (tracker :: <my-tracker>,
                                 final-point :: <point>) => ()
    ignore(final-point);

    track-feedback(tracker, final-point, final-point, #"off");
end method;
```

Listing 19-7 shows a `track-constrain` method that limits the current mouse position to be within the rectangle,

Listing 19-7 Constraining tracking

```
define method track-constrain (tracker :: <my-tracker>,
                                    last-point :: <point>,
                                    current-point :: <point>)
                                    => constrained-point :: <point>;
    ignore(last-point);
    let new-v = current-point.v;
    let new-h = current-point.h;

    if (current-point.v < tracker.constraint.top)
        new-v := tracker.constraint.top;
    end if;
    if (current-point.v > tracker.constraint.bottom)
        new-v := tracker.constraint.bottom;
    end if;
    if (current-point.h < tracker.constraint.left)
        new-h := tracker.constraint.left;
    end if;
    if (current-point.h > tracker.constraint.right)
        new-h := tracker.constraint.right;
    end if;

    point(new-h, new-v);
end method;
```

Listing 19-8 shows a `track-feedback` method that draws an outline of the rectangle as the mouse moves. You could also call `draw-feedback-rect` after creating the rectangle object to provide the same kind of feedback as shown in Listing 19-8.

Listing 19-8 Implementing feedback

```
define method track-feedback (tracker :: <my-tracker>,
                                      last-point :: <point>,
                                      current-point :: <point>,
                                      mode :: <symbol>) => ()
    ignore(last-point, mode);

    let draw-rect = make(<rect>);
    set-rect-from-points(draw-rect,
    tracker.start-point, current-point);

    PenMode($srcXor);
    PenPat($gray-pattern);
    frame-region(draw-rect);
    PenNormal();
end method;
```

Scrolling and Tracking Objects Reference

The following sections describe the classes used to implement scrolling and tracking.

The <scroller> Class

The `<scroller>` class represents the visible part of a view in a scrolling window. You do not need to define subclasses of the `<scroller>` class. For each scrolling window, need to create a `<scroller>` object and add it as a subview of the window's root view. Listing 19-9 shows the class definition for the `<scroller>` class.

Listing 19-9 The <scroller> class definition

```
define primary sealed class <scroller> (<view>)
    slot %translation :: <point>,
     init-value: $zero-point,
     init-keyword: translation:;

  slot sub-view-extent :: <point>,
     init-value: $zero-point;

  slot vertical-size-sub-views :: <boolean>,
     init-value: #t,
     init-keyword: vertical-size-sub-views:;

  slot horizontal-size-sub-views :: <boolean>,
     init-value: #t,
     init-keyword: horizontal-size-sub-views:;
end class;
```

Slot descriptions

`%translation` The distance between a scroller view's location and the location of its subviews. You can use the `translation:` keyword or `translation-setter` to change this value. Use `translation` to get the value.

`sub-view-extent` The extent of the largest subview. The Framework calculates the extent for you when a subview is added to the scroller or a subview's extent changes.

`vertical-size-sub-views` Specifies whether to resize the vertical side of a subview (`#t`) or not (`#f`) when a view is added to or removed from the scroller or when a view's extent changes.

`horizontal-size-sub-views` Specifies whether to resize the horizontal side of a subview (`#t`) or not (`#f`) when a view is added to or removed from the scroller or when a view's extent changes.

Initialization

No `initialize` method is defined for `<scroller>` objects.

The <tracker> Class

The `<tracker>` class represents feedback for mouse movements while the mouse key is down. You need to define a subclass of the `<tracker>` class if you want to add slots; for example, to maintain constraint information. You can also create `<tracker>` objects and use them if your tracker does not require additional state information. Listing 19-10 shows the class definition for the `<tracker>` class.

Listing 19-10 The `<tracker>` class definition

```
define primary open class <tracker> (<object>)
  slot tracker-view,
     init-value: #f,
     init-keyword: view:;
  slot tracker-scroller,
     init-value: #f;
  slot auto-scroll? :: <boolean>,
     init-value: #t,
     init-keyword: auto-scroll:;
  slot start-point :: <point>,
     init-keyword: start-point:;
end class;
```

Slot descriptions

`tracker-view` The view on which the mouse is being tracked. Use the `view:` keyword to specify the value; otherwise, the value is false (`#f`). You can use the getter and setter to access this slot.

`tracker-scroller` The scroller whose subview is the view on which the mouse is being tracked. The Framework sets the value of this slot for you. You can use the getter and setter to access this slot.

`auto-scroll?` Specifies whether you want the view to scroll during tracking (`#t`) or not (`#f`). Use the `auto-scroll:` keyword and specify false (`#f`) to prevent scrolling; otherwise, the value is true (`#t`), meaning that scrolling occurs when the

	mouse if moved outside the scroller. You can use the getter and setter to access this slot.
`start-point`	The starting point for tracker feedback. Use the `start-point:` keyword to specify the point; otherwise, the value is not defined. You can use the getter and setter to access this slot.

Initialization

No `initialize` method is defined for `<tracker>` objects.

The <tracker-adorner> Class

The `<tracker-adorner>` class implements the visual track feedback. The Framework sets up the adorner for you. You do not need to define a subclass or create objects from this class. Listing 19-11 shows the class definition for the `<tracker-adorner>` class.

Listing 19-11 The `<tracker-adorner>` class definition

```
define primary sealed class <tracker-adorner> (<adorner>)
  slot the-tracker :: <tracker>,
     setter: #f,
     required-init-keyword: tracker:;

  slot the-last-point :: <point>,
     required-init-keyword: last-point:;

  slot the-current-point :: <point>,
     required-init-keyword: current-point:;
end class;
```

Slot descriptions

`the-tracker`	The tracker to which the adorner is attached.
`the-last-point`	The previous mouse position.
`the-current-point`	The current mouse position.

Initialization

No `initialize` method is defined for `<tracker-adorner>` objects.

Functions for Scrolling and Tracking

The following functions can be used with `<scroller>` objects.

Table 19-1 Functions for scrolling

Function	Purpose	Call	Sp.
`add-sub-view`	Add a view to the specified `<scroller>` object.	√	
`remove-sub-view`	Remove a view from the specified `<scroller>` object.	√	
`get-scroller`	Retrieve a view's `<scroller>` superview.	√	

The following functions can be used with `<tracker>` objects.

Table 19-2 Functions for tracking

Function	Purpose	Call	Sp.
`track-mouse`	Start tracking the mouse.	√	
`track-begin`	Specify the actions to take when mouse tracking begins.		√
`track-end`	Specify the actions to take when mouse tracking ends.		√
`track-constrain`	Specify actions to take to constrain mouse movements.		√
`track-feedback`	Specify actions to take to provide visual feedback to the user about mouse movements.	√	√
`draw-feedback-rect`	Draw a simple feedback rectangle.	√	

CHAPTER 20

Documents

Contents

About Documents 479
The Open Protocol 479
Commands and Change Counts 480
Document Menu Items 480
Using Documents 481
Defining a Document Subclass 481
Registering a Document 482
Creating and Opening a Document 482
Creating a Document's Windows 483
Specifying a Document's File Type 484
Document Objects Reference 484
The <document> Class 485
Functions for Documents 487

This chapter discusses the use of documents associated with an application. Often, but not always, a document has a file to hold the document's data. This chapter discusses the relationship between a document and a file; however, this chapter does not describe files completely. For information about files, see "About Files" on page 491.

About Documents

A document represents data that is displayed in a window. Often this data is stored in a file. A document does not by itself contain data, although it may refer to collections, TextEdit handles, or other structures that do contain data. A document instead manages the representation of data onscreen and also manages its movement between the application and disk.

The Open Protocol

It is useful to know the actions that the Framework takes with you open a document. When you call `open` for a document object, The Framework takes the following actions:

1. The command context is established for undo and redo. Typically, the context is the document itself, which provides undo and redo on a per-document basis. For information about the command context, see "The <command-context> Class" on page 539.
2. If the document has a disk file associated with it, it is opened and the contents are read. You can define `open-files` and `read-from-file` methods specialized on your document class to control which files are opened and how the contents are retrieved.
3. If a document does not have a disk file associated with it, the Framework calls `do-initial-state` to set up the document. You define a `do-initial-state` method specialized on your document class. The method specifies how to set up the document.
4. The document's title is set, either using the file name if the document has a disk file associated with it, or by using "Untitled-n," where the "n" is replaced by the next number, starting with 1.

5. The Framework calls make-windows, which allows you to set up the document's window. For an example, see "Creating a Document's Windows" on page 483.

Commands and Change Counts

Changes to a document are often implemented with commands. Commands allow a change to be undone and redone as necessary. You are not required to use commands to make changes to a document; however, the state of certain menu items, such as whether or not the Save item is enabled, rely on change counts that are maintained by commands. If you do not use commands, you need to maintain the change count information yourself so that the change count increments when the document changes and it is reset whenever the document is saved. For information about commands, see "About Commands" on page 531.

Document Menu Items

Document-related menu items are New, Open, Save, Save As, Save Copy, and Revert. These menu items are enabled in several ways, as follows:

- You are responsible for enabling the `#"new"` menu item. Typically, you handle this in a behavior that you add to the main handler. For an example, see "Creating and Opening a Document" on page 482.
- The `#"open"` menu item is enabled by the Framework after a document has been registered with the main handler. See "Registering a Document" on page 482 for an example.
- The `#"save"` and `#"revert"` menu items are enabled by the Framework if a document object is in the target chain and the document's change count is greater than zero, indicating that a change has occurred. See the previous section, "Commands and Change Counts," for more information.
- The `#"save-as"` and `#"cave-copy"` menu items are always enabled by the Framework if a document object is in the target chain.

Using Documents

The following sections provide examples of using documents:

- defining document subclasses
- creating and opening documents
- creating windows for documents
- registering a document
- specifying the kind of data in a document's files

Defining a Document Subclass

You almost always define a subclass of the `<document>` class that includes storage for your data, a reference to the document's views, or other per-document information. Listing 20-1 shows a `<document>` subclass definition for a text document. It includes slots to hold the text, style information, and a reference to a view of the data.

Listing 20-1 Defining a document subclass

```
define class <text-document> (<document>)
  slot text-handle :: <machine-pointer>,
     init-value: $null-machine-pointer;

  slot style-handle :: <StScrpHandle>,
     init-value: as(<StScrpHandle>, 0);

  slot text-view :: <text-view>;
end class;
```

Registering a Document

You must register each document class with the main handler. This allows the application to open a document by looking at the data type of the file associated with the document. If the file type is registered, the application can create the appropriate document object to manage the file.

You specify the file type and the document class that supports the type to your application by calling the `add-document-type` method from the method that initializes your application. Listing 20-2 shows the call to `add-document-type` that sets up this correspondence between text files and the `<text-document>` class.

Listing 20-2 Specifying a document's data type

```
add-document-type(*main-handler*, ostype("TEXT"), <text-document>);
```

Creating and Opening a Document

Listing 20-3 shows how to create and open a document in response to the New menu item being selected.

Listing 20-3 Creating a document

```
define method behavior-event (behavior :: <text-app-behavior>,
                              next-behaviors :: <list>,
                              main-handler :: <main-handler>,
                              event :: <menu-event>,
                              id == #"new") => ()
  ignore(behavior, next-behaviors, main-handler, event, id);

  open(make(<text-document>));
 end method;
```

When a document is opened, the Framework opens a document's files and reads their contents if the document has a disk file; otherwise the Framework calls `do-initial-state` to perform other initialization. You can specialize the

`do-initial-state` method on your document subclass to specify the actions to take before a file has been created for a document; for example, in response to the New menu item being selected, and specialize the `read-from-file` method for your document to read the contents of an existing document.

Note
A file is not associated with a document until the document has been saved. For more information about the relationship between documents and files, see "Documents and Files" on page 492. ◆

Creating a Document's Windows

When a document is opened, the Framework opens a document's windows after it opens files and sets up the initial state for a document. It calls the `make-windows` method which you must provide for your document subclass. In addition to creating windows, the `make-windows` method is the appropriate place to create the window's views and determiners, as well as behaviors for either the window or its views. Listing 20-4 shows an example of creating a window, creating the window's view and contents, and setting up a behavior for the view. The document's reference to its view is also set up here.

Listing 20-4 Creating a document's windows

```
define method make-windows (document :: <text-document>) => ()
  let view = make(<text-view>,
                  extent: point(700, 300),
                  use-styles: #t,
                  text-style: $geneva12);


  add-behavior(view, make(<styled-text-behavior>));

  if (~ nil?(document.text-handle))
    set-text(view, document.text-handle, style-handle:
document.style-handle);
    DisposeHandle(document.text-handle);
    document.text-handle := $null-machine-pointer;
```

```
  end if;

open(make-scrolling-window(list(view), title: document.title,
                           target-view: view,
                           next-handler: document,
                           main-window: #t,
                           location: point(100, 100),
                           extent: point(400, 300),
                           closable: #t, zoomable: #t, resizable: #t));
  document.text-view := view;
end method;
```

Specifying a Document's File Type

You must specify the file type of the kind of file you want to create by defining a `main-file-type` method specialized on the document parameter. This method is called whenever a document's file is saved. Listing 20-5 shows an example.

Listing 20-5 Specifying a document's file type

```
define method main-file-type (doc :: <text-document>)
                                                => result :: <integer>;
  ignore(doc);
  ostype("TEXT");
end method;
```

Document Objects Reference

The following sections describe the `<document>` class and functions that you can use with documents.

The <document> Class

The `<document>` class represents a container for the information that is manipulated by your application. Typically, a document appears onscreen in the content views of a window and is saved to and restored from a file. A document is not required to have a file associated with it, however.

Because a document's contents are one of the major ways that your application is differentiated from other applications, you almost always define a subclass of the `<document>` class and create an object from your subclass for each document that is being manipulated by the application. Your subclass typically defines or refers to the data structures of the information content in your application.

Listing 20-6 shows the class definition for the `<document>` class.

Listing 20-6 The `<document>` class definition

```
define primary open class <document> (<window-context>)
    slot change-count :: <integer>,
        init-value: 0;
    slot commit-on-save? :: <boolean>,
        init-value: #f;
    slot open? :: <boolean>,
        init-value: #f;
    slot main-file,
        init-value: #f;
    slot main-file-type :: <integer>,
        init-value: ostype("????"),
        init-keyword: main-file-type:;
    slot main-file-creator :: <integer>,
        init-value: ostype("????"),
        init-keyword: main-file-creator:;
  // private
    slot %title :: <string>,
        init-value: "", init-keyword: title:;
end class;
```

Slot descriptions

`change-count`	The number of changes to the document's contents since it was last saved. The Framework manipulates this value for you when you use commands to change the document. You can call the getter to determine the number of changes.
`commit-on-save?`	Specifies whether the current command is considered complete and cannot be undone at the time the document's contents are saved. By default, the command is considered undoable after a save operation.
`open?`	Specifies whether the document is open. Call `open` to open the document, which sets the value of this slot to true (`#t`), or `close` to close the document, which sets the value of this slot to false (`#f`). You can call `open?` to determine whether or not a document is open.
`main-file`	The file that stores the document's data. For more information, see the section "Initialization," described below.
`main-file-type`	The OS type that specifies the kind of file.
`main-file-creator`	The OS type of the creating application.
`%title`	The document's title. Use the `title:` keyword to specify the title; otherwise, the title is an empty string. Use the `title` and `title-setter` functions to access this slot. The `%title` slot is not exported.

Initialization

The `initialize` method for `<document>` objects sets up the association between a document and its file and the next event handler in the target chain for the document. The `initialize` method's parameters are defined as follows:

```
define method initialize (doc :: <document>,
                          #key main-file: the-file) => ()
```

`doc`	The document.
`the-file`	The file to associate with this document. Use the `main-file:` keyword to specify the file; otherwise, the document does not have a file.

Functions for Documents

The following functions can be used with documents.

Table 20-1 Functions for documents

Function	Purpose	Call	Sp.
`has-disk-file?`	Determine whether a document has a disk file associated with it.	√	
`do-initial-state`	Specify the actions to take when opening a document for which a file does not exist.		√
`make-windows`	Specify the actions to take when creating windows for a document.		√
`open`	Open a document.	√	
`close`	Close a document.	√	
`revert`	Change the contents of a document back to its last saved state.		√
`do-setup-menus`	Specify actions to take to set up a document's menu items.		√
`do-event`	Specify the actions to take in response to an event.		√
`get-document`	Determine the document associated with an event handler.	√	

The following functions can be used for document file manipulation. For information about files, see “About Files” on page 491.

Table 20-2 Functions for files associated with documents

Function	Purpose	Call	Sp.
`main-file-type`	Specify a document’s file type.	√	
`setup-file -permissions`	Specify permissions for a document’s file.	√	
`open-files`	Open the files associated with a document.		√
`close-files`	Close the files associated with a document.		√
`read-from-file`	Read the contents of a document’s file.		√
`write-to-file`	Write the contents of a document’s file.		√

CHAPTER 21

Files

Contents

About Files 491
Manipulating Files 491
Manipulating Resources 492
Documents and Files 492
Using Files 493
Reading From a File 493
Writing To a File 494
Reading From a Resource Fork 495
Updating a Resource Fork 495
File Objects Reference 496
The <file> Class 496
File Functions 500

This chapter discusses the use of files, which represent Macintosh File Manager files. This chapter also discusses resources that you may store in or retrieve from the resource fork of a file. The use of files is closely related to the use of documents and streams. This chapter discusses how to use files with documents. For information about streams, see "About Streams" on page 505.

About Files

A `<file>` object represents a file on disk. Specifically, the `<file>` class defines a Macintosh File Manager file specification record that the Framework uses to implement file operations. These operations include:

- creating files
- opening, closing, and deleting files
- reading and writing files

Reading and writing files may require you to read and write resources as well as data. The Framework provides functions that allow you to retrieve and save data into both the data fork and resource fork of a file.

Manipulating Files

Before you can manipulate a file on disk, you must create a file object, for which you must define a Macintosh File Manager file specification record. The Framework allows you to build the record from an alias or from a pathname if you do not have the file specification record itself.

After you create a `<file>` object, you can use it in the following ways:

- create a file that matches the specification in the `<file>` object, if the file does not exist
- open the file
- close the file
- delete the file; it must be closed first
- read the contents of the file

- write the contents of the file

Table 21-1 on page 500 identifies the functions you can use to perform these operations.

Manipulating Resources

The Framework provides functions that:

- change the current resource file
- retrieve resources
- add resources
- delete resources

IMPORTANT
Each Framework-provided function may change the current resource fork. If you are calling both Framework functions and Macintosh Toolbox routines to manipulate the contents of a resource fork, you may wish to explicitly set the current resource fork before calling a Toolbox routine. ▲

Table 21-2 on page 501 identifies the functions you can use to perform operations on resources.

Documents and Files

Often a file is related to a document, because the file serves to hold the document's contents. If you only work with documents, you seldom need to manipulate files directly. For example, you do not need to open, close, or delete files yourself. The Framework handles these tasks for you at the appropriate times. You must, however, specify how to read and write the contents of the data fork and, if you use it, the resource fork.

When an existing document that uses a file is opened, the Framework creates a file object and reads the contents of the file for you. (For more information about how a document is opened, see "The Open Protocol" on page 479.) For new documents, the Framework creates a file object and writes the document's

data to a file when the user requests Save, Save As, or Save a Copy. The Framework does not create a file object before the contents need to be written.

You must specify how to store the document's contents in a file and retrieve those contents. You provide methods that

- read the contents of the file associated with a document when a document is opened. See "Reading From a File" on page 493 for an example.
- write the contents of the file when its document is saved. See "Writing To a File" on page 494 for an example.
- restore the contents of a document from a file when the user chooses Revert.

Table 21-3 on page 502 identifies the file-related functions for documents.

Using Files

The following sections show examples of how to:

- read data from a file
- write data to a file
- read a resource from a resource fork
- update a resource fork

Reading From a File

When reading data from a file associated with a document, the Framework calls `read-from-file`. You must define a `read-from-file` method whose document parameter is specialized on your document subclass. If you are reading from a file that is not associated with a document, you must decide when to actually read the data. In either case, you can call `read-data` to read data from a file.

Listing 21-1 shows a `read-from-file` method that calls `read-data` to read data from a file. The `read-from-file` method calculates the size of the file, in bytes, by calling `get-end-of-file` to determine the last byte. Enough space is reserved to hold the data in memory and it is locked down while it is filled.

Listing 21-1 Reading from a file

```
define method read-from-file (document :: <text-document>,
                              file :: <file>) => ()
  let length = get-end-of-file(file);
  document.text-handle := NewHandle(length);
  fail-nil(document.text-handle);

  with-locked-handle(document.text-handle)
    let ptr = pointer-at(document.text-handle);

    read-data(file, ptr, length);

end;
end method;
```

Writing To a File

When writing data to a file associated with a document, the Framework calls `write-to-file`. You must define a `write-to-file` method whose document parameter is specialized on your document subclass. If you are writing to a file that is not associated with a document, you must decide when to actually write the data. In either case, you can call `write-data` to write data to a file.

Listing 21-2 shows a `write-to-file` method that calls `write-data` to write data to a file. The `write-to-file` method calls `set-end-of-file` to reset the file's length to 0 bytes. The data in memory is locked down while it is being written.

Listing 21-2 Writing to a file

```
define method write-to-file (document :: <text-document>,
                             file :: <file>,
                             making-copy? :: <boolean>) => ()
  ignore(making-copy?);

  set-end-of-file(file, 0);

  let handle = get-text-handle(document.text-view);
```

```
  with-locked-handle(handle)
    let ptr = pointer-at(handle);
    let length = GetHandleSize(handle);

    write-data(file, ptr, length);
  end;
end method;
```

Reading From a Resource Fork

Listing 21-3 shows the first `"styl"` resource being retrieved from the resource fork of a file. It is then detached so that the in-memory version can be manipulated without affecting the resource on disk. Note that the file's resource fork must be open in order to retrieve its contents.

Listing 21-3 Reading from a resource fork

```
if (file.resource-fork-open?)
      let style = get-ith-resource-from-file(file, ostype("styl"), 1);
      unless(nil?(style))

        DetachResource(style);
        document.style-handle := as(<StScrpHandle>, style);
      end unless;
    end if;
```

Updating a Resource Fork

Listing 21-4 shows the `"styl"` resource replacing all other resources of the same type in the resource fork of a file. Note that the file's resource fork must be open in order to retrieve its contents.

Listing 21-4 Updating a resource fork

```
if (file.resource-fork-open?)
    delete-all-resources-in-file(file, ostype("styl"));
    add-resource-to-file(file,
                         as(<machine-pointer>,
                         get-style-handle(document.text-view)),
                         ostype("styl"),
                         128);
end if;
```

File Objects Reference

The following sections describe the `<file>` class and the functions you can use on files.

The <file> Class

The `<file>` class represents a Macintosh file as defined by a Macintosh File Manager file specification record. For information about file specification records, see the description of the `FSSpec` data type in Inside Macintosh: Files.

You do not define subclasses of the `<file>` class; rather you create `<file>` objects. The Framework provides methods that create `<file>` objects for you when you prompt the user for a document name using the Open and Save As dialog boxes. (See `chooses-document` and `request-file-name`, respectively.) Listing 21-5 shows the class definition for the `<file>` class.

Listing 21-5 The `<file>` class definition

```
define primary open class <file> (<object>)
  slot file-name :: <string>,
     init-keyword: name:;
  slot dir-id :: <integer>,
     init-keyword: dir-id:;
```

```
  slot vol-ref-num :: <integer>,
     init-keyword: vol-ref-num:;

  slot data-ref-num :: <integer>,
     init-value: $no-ref-num;
  slot rsrc-ref-num :: <integer>,
     init-value: $no-ref-num;

  slot use-data-fork? :: <boolean>,
     init-value: #t,
     init-keyword: use-data-fork:;
  slot use-rsrc-fork? :: <boolean>,
     init-value: #f,
     init-keyword: use-rsrc-fork:;
  slot require-rsrc-fork? :: <boolean>,
     init-value: #f,
     init-keyword: require-rsrc-fork:;

  slot file-type :: <integer>,
     init-value: ostype("????"),
     init-keyword: file-type:;
  slot file-creator :: <integer>,
     init-value: ostype("????"),
     init-keyword: file-creator:;

  slot data-permission :: <integer>,
     init-value: $read-write-permission,
     init-keyword: data-permission:;
  slot rsrc-permission :: <integer>,
     init-value: $read-write-permission,
     init-keyword: rsrc-permission:;
end class;
```

Slot descriptions

`file-name` The name of the file. Use the `name:` keyword to specify the file name; otherwise, the name must be specified in a Macintosh File Manager file specification record. For more information about using a file specification record, see the following section, “Initialize.”

`dir-id` The directory ID of the directory containing the file. Use the `dir-id:` keyword to specify the directory ID; otherwise, the ID must be specified in a Macintosh File Manager file specification record. For more information about using a file specification record, see the following section, "Initialize."

`vol-ref-num` The volume reference number of the volume containing the file. Use the `vol-ref-num:` keyword to specify the file volume; otherwise, the reference number must be specified in a Macintosh File Manager file specification record. For more information about using a file specification record, see the following section, "Initialize."

`data-ref-num` The data reference number of the fork containing the file's data. Use the `data-ref-num:` keyword to specify the data fork; otherwise, the reference number is determined in when the fork is opened. Initially, the value of this slot is `$no-ref-num`, which specifies that the data fork is not open. You can call `data-fork-open?` to determine the status of the data fork.

`rsrc-ref-num` The resource reference number of the fork containing the file's resources. Use the `data-ref-num:` keyword to specify the resource fork; otherwise, the reference number is determined in when the fork is opened. Initially, the value of this slot is `$no-ref-num`, which specifies that the resource fork is not open. You can call `rsrc-fork-open?` to determine the status of the resource fork.

`use-data-fork?` Specifies whether the file uses a data fork (`#t`) or not (`#f`). Use the `use-data-fork:` keyword to specify true (`#t`) if you want to use the data fork; otherwise the value is false (`#f`), which specifies that the data fork is not being used. If you specify true (`#t`), the Framework creates the data fork when the file is created.

`use-rsrc-fork?` Specifies whether the file uses a resource fork (`#t`) or not (`#f`). Use the `use-rsrc-fork:` keyword to specify true (`#t`) if you want to use the resource fork; otherwise the value is false (`#f`), which specifies that the resource fork is not being used. If you specify true (`#t`), the Framework creates the resource fork when the file is created.

require-rsrc-fork?	Specifies whether the file requires a resource fork (#t) or not (#f). Use the require-rsrc-fork: keyword to specify true (#t) if the file must have a resource fork; otherwise the value is false (#f), which specifies that file may have a resource fork, however, it is not required to have one. If you specify true (#t), the Framework notifies the user of an error if the file does not have a resource fork.
file-type	The file type, such as "TEXT" or "PICT". Use the file-type: keyword to specify the file type; otherwise, the file type is retrieved from the Finder if the file exists or "????" is used if the file does not exist.
file-creator	The file's creator. Use the file-creator: keyword to specify the creator; otherwise, the file's creator is retrieved from the Finder if the file exists or "????" is used if the file does not exist.
data-permission	The permissions for the data fork of a file. Use the data-permission: keyword to specify the permissions; otherwise, the default permission is for reading ($read-write-permission).
rsrc-permission	The permissions for the resource fork of a file. Use the rsrc-permission: keyword to specify the permissions; otherwise, the default permission is for reading ($read-write-permission).

Initialization

The initialize method for <file> objects sets up a File Manager file specification record to associate with the object. If you do not specify keywords when you create a <file> object, you must specify either a file specification record or an alias handle. The initialize method's parameters are defined as follows:

```
define method initialize (file :: <file>,
                          #key fs-spec: fs-spec, alias: alias) => ()
```

file	The file.
fs-spec	The Macintosh File Manager file specification record associated with the <file> object. Use the fs-spec: keyword to specify the record.

`alias` An alias handle that resolves to the file specification record associated with the `<file>` object. Use the `alias:` keyword to specify the record.

If you specify both a file specification record and an alias handle, the file specification record is used.

File Functions

The following functions can be used to manipulate files.

Table 21-1 Functions for files associated with documents

Function	Purpose	Call	Sp.
`choose-document`	Create a file object to associate with a file selected with the Open dialog box.	√	
`request-file-name`	Create a file object associated with the name specified in the Save As dialog box.	√	
`data-fork-open?`	Determines whether or not the file's data fork is open.	√	
`create`	Create a new file.	√	
`delete`	Delete the specified file.	√	
`open`	Open a file. (Opens the data and resource forks, as required.)	√	
`open-data-fork`	Open a file's data fork.	√	
`open-resource-fork`	Open a file's resource fork.	√	
`close-data-fork`	Close a file's data fork.	√	
`close-resource-fork`	Close a file's resource fork.	√	
`close`	Close a file.	√	
`read-data`	Read bytes from a file.	√	
`write-data`	Write bytes to a file.	√	

Table 21-1 Functions for files associated with documents

Function	Purpose	Call	Sp.
`specify-from-fs-spec`	Set the slots of a `<file>` object from a Macintosh File manager file specification record.	√	
`specify-from-alias`	Set the slots of a `<file>` object from an alias handle.	√	
`specify-from-path-name`	Set the slots of a `<file>` object from a path name.	√	
`set-fs-spec`	Set up a Macintosh File manager file specification record from the slots in a `<file>` object.	√	
`specify-from-path-name`	Set the slots of a `<file>` object from a path name.	√	
`get-file-position`	Determine the current file position mark.	√	
`set-file-position`	Set the current file position mark.	√	
`get-end-of-file`	Determine the current logical end of a file.	√	
`set-end-of-file`	Set the current logical end of a file.	√	
`make-alias`	Create an alias to a file.	√	

The following functions can be used for resource fork manipulation.

Table 21-2 Functions for resources

Function	Purpose	Call	Sp.
`set-current-resource-file`	Specify the current resource file to use.	√	
`get-ith-resource-from-file`	Retrieve the resource specified by index from an open resource fork.	√	
`get-resource-from-file`	Retrieve the specified resource from an open resource fork.	√	

Table 21-2 Functions for resources

Function	Purpose	Call	Sp.
`get-named-resource -from-file`	Retrieve the specified named resource from an open resource fork.	√	
`count-resources-in -file`	Determine the number of resources matching the specified type in the current resource file.	√	
`add-resource-to-file`	Add a resource to the current resource file.	√	
`delete-resource-from -file`	Remove the specified resource from the resource file.	√	
`delete-all-resources -in-file`	Remove all resources of the specified type from the resource file.	√	

The following functions can be used for document file manipulation. For information about documents, see "About Documents" on page 479.

Table 21-3 Functions for files associated with documents

Function	Purpose	Call	Sp.
`main-file-type`	Specify a document's file type.	√	
`setup-file -permissions`	Specify permissions for a document's file.		√
`open-files`	Open the files associated with a document.		√
`close-files`	Close the files associated with a document.		√
`read-from-file`	Read the contents of a document's file.		√
`write-to-file`	Write the contents of a document's file.		√

CHAPTER 22

Streams

Contents

About Streams 505
About Class and Slot Specifications 507
Using Class and Slot Specifications 507
  Setting up the Class and Slot Specifications 508
  Using Slot Specification Getters and Setters 509
  Cloning Objects 511
  Using Shared and Default Objects 512
Using Streams 513
  Using Handle Streams 513
  Using Object Streams 514
    Reading and Writing Objects 515
    Reading and Writing Primitive Data Types 516
Stream Objects Reference 517
  The <stream> Class 517
  The <file-stream> Class 518
  The <handle-stream> Class 519
  Object Stream Classes 521
  The <class-spec> Class 522
  The <slot-spec> Class 523
  Functions for Streams 525

This chapter discusses streams, which allow you to interpret data, such as the contents of a handle or file, as a sequence of bytes or other data types. The Framework provides a mechanism for interpreting its objects, which are defined by class and slot specifications. You must provide a mechanism of interpreting other kinds of data in streams. This chapter also discusses other uses of class and slot specifications, namely their use in cloning and sharing of objects.

About Streams

Streams provide the ability to work with a sequence of data at a higher level of abstraction than might otherwise be possible if you need to also manage the source of the data, in the case of input, or its sink, in the case of output. For example, if you are reading a file and examining its contents, you must be concerned about when you to read the next chunk of data from the file into memory, how to detect the end of the file, and so on. These requirements add to the complexity of obtaining the next byte, integer, string, or object from the source.

A stream allows you to divide the responsibility for obtaining a piece of data from the responsibility of managing the source or sink for the data. In the case of a file stream, the stream manages the file to which it is attached and, thus, transfers data between the file and the stream at the appropriate times. You are only concerned about obtaining data from the stream or adding to the stream. You may, of course, need to specify requirements, such as the interval (determined by buffer size) that transfers take place between the file and stream, but this is a one-time task and separate from manipulating the stream itself.

The Framework provides streams whose source or sink can be either handles or files. It allows you to read or write to the stream in several ways, which may be combined:

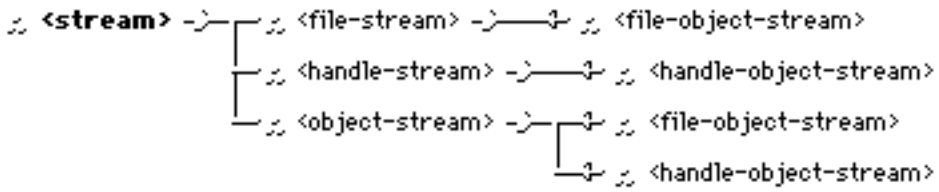
- by byte or by a series of bytes
- by numbers, which are two-byte or four-byte integers
- by strings, which can be either null-terminated or Pascal-style strings
- by objects, which are defined by their respective classes and can be identified by name or symbol

The Framework provides other housekeeping functions as well.

Note
You can use the functions that read data from streams and write data to streams without specialization, or you can define implementations of the methods that efficiently meet your special needs. For a list of functions that read from and write to streams, see Table 22-1, "Functions for streams" on page 525. ◆

Figure 22-1 shows the `<stream>` class and its subclasses.

Figure 22-1 Stream classes



The classes shown in Figure 22-1 represent the following kinds of objects:

`<stream>`	Represents an abstract stream from which subclasses can be defined.
`<file-stream>`	Represents a stream associated with a file.
`<handle-stream>`	Represents a stream associated with a handle.
`<object-stream>`	Represents a stream that contains objects.
`<file-object-stream>`	Represents a stream associated with a file that contains objects.
`<handle-object-stream>`	Represents a stream associated with a handle that contains objects.

About Class and Slot Specifications

The Framework provides functions that allow you to specify the contents of classes and slots. You make these specifications in top-level forms that are called when Dylan starts to execute code. The functions are:

`make-class-spec`	Specifies how to create a class.
`make-slot-spec`	Specifies how to create a slot.

These top-level forms allow the Framework to build tables of class specifications. These tables are used in three ways:

- to specify how to stream an object of a class
- to specify how to clone an object of a class
- to provide a default object that can be shared

The tables specify the following kinds of information:

- functions to call to obtain the value of a slot when the object is being written to a stream or cloned.
- keywords used to specify values for slots as values are read from a stream.
- whether or not an object can be shared, which affects how the object is streamed or cloned

Because some objects can be shared, the `make-class-spec` top-level form allows a default object to be created when the class is specified. A reference to the default object can be obtained in place of creating a new object whenever a new shareable object is needed.

Using Class and Slot Specifications

The following sections discuss how to set up class and slot specifications and how to use them for cloning and sharing objects. For examples of how to use class and slot specifications with object streams, see "Using Object Streams" on page 514.

Setting up the Class and Slot Specifications

When Dylan starts to execute, it first executes all top-level forms. To set up object streaming, you provide a `make-class-spec` form that specifies the class and its slots for each class that you may wish to stream. The specification consists of:

- the class name
- its identifying symbol
- a list of slot specifications, one for each slot that should be streamed.

You set up a slot specification by calling `make-slot-spec`, which must provide at least the following information:

- the getter function, which is used to obtain a value for the slot before it is written to a stream and to obtain its value when it is displayed in the Apple Dylan User Interface Builder property editor
- the setter function, which is used to set the slot's value from the Apple Dylan User Interface Builder property editor
- the initial keyword associated with the slot, which is used to set the slot when the value is read from a stream

Listing 22-1 shows a class specification for the `<window>` class, whose identifying symbol is `#"window"`. The list of slot specifications follows. Each slot specification shows the function used to obtain the value to write to the stream, a setter function used by the Apple Dylan User Interface Builder, and the slot's initial keyword, which is used to set the slot when its value is retrieved from the stream. The setter function for the window's resizeable slot is `$no-setter`, which means that the slot cannot be changed by the Apple Dylan User Interface Builder property editor.

Listing 22-1 The `make-class-spec` and `make-slot-spec` forms

```
make-class-spec(<window>, #"window",
    list(
        make-slot-spec(title, title-setter, title:),
        make-slot-spec(resizable?, $no-setter, resizable:),
        …
        make-slot-spec(maximum-extent,
```

```
                        maximum-extent-setter,
                        maximum-extent:)
    )
);
```

If you do not specify otherwise, the getter function in your call to `make-slot-spec` is also used to create clones of objects as well as to write a slot's value to a stream. For more information about cloning objects, see "Cloning Objects" on page 511.

Using Slot Specification Getters and Setters

In Listing 22-1, the getter and setter functions for each slot are the same as those defined by Dylan when you define a class. You are not required to use the default getter and setter functions, you may create your own.

Listing 22-2 shows the slot specification for a view's adorner list. The `streamable-adorners` function is used to obtain the adorners to write to the stream. The `clone-adorner-list` function specifies how these adorners are written to the stream.

Listing 22-2 Using non-default getters and setters

```
make-class-spec(<view>, #"view",
    list(
         // getter should only get streamable adorners
        make-slot-spec(streamable-adorners,
            adorner-list-setter,
            adorners:,
            cloning-getter: clone-adorner-list),
        …
    )
);
```

For example, the Framework allows you to specify whether to allow adorners to be streamed. (For information about adorners, see "Adorners" on page 339.) When writing to the stream, therefore, it is necessary to specify a getter function that only obtains streamable adorners.

Listing 22-3 shows the definition of the `streamable-adorners` function used in Listing 22-2.

Listing 22-3 Specifying a slot getter function for streaming

```
define method streamable-adorners (view :: <view>) => adorners :: <list>;
  choose(streamable?, view.adorner-list);
end method;
```

For example, the Framework does not clone shareable adorners; it only creates a clone it if it should not be shared. Because the getter function does not make this distinction, a separate clone getter function is specified.

Listing 22-4 shows the definition of the `clone-adorner-list` function used in Listing 22-2.

Listing 22-4 Specifying a slot getter function for cloning

```
define method cloneable-adorners (view :: <view>) => adorners :: <list>;
  choose(cloneable?, view.adorner-list);
end method;

define method clone-adorner-list (view :: <view>) => adorners :: <list>;
  local method get-adorner (adorner :: <adorner>)
          if (adorner.shareable?)
            adorner;
          else
            clone(adorner);
          end;
        end method;

  map(get-adorner, view.cloneable-adorners);
end method;
```

Cloning Objects

By default, the contents of a slot are cloned. You do not typically want to clone numbers, booleans, constants, or references to shared objects because the value itself is sufficient. In these cases, you specify `clone: #f` when you call `make-slot-spec`.

Listing 22-5 shows the `make-slot-spec` calls for the slots that define a rectangle. Cloning is not performed because it would cause `<number>` objects to be created when integer values are sufficient.

Listing 22-5 Preventing cloning

```
make-class-spec(<rect>, #"rect",
    list(
        make-slot-spec(top, top-setter, #"top", clone: #f),
        make-slot-spec(left, left-setter, #"left", clone: #f),
        make-slot-spec(bottom, bottom-setter, #"bottom", clone: #f),
        make-slot-spec(right, right-setter, #"right", clone: #f)
    )
);
```

The Framework provides a `clone` method to clone an object of any class for which `make-class-spec` has been called. If you want to create a clone of an object for which `make-class-spec` has not been called or if you wish to specify the way that cloning is implemented for an object, you may provide a `clone` method whose object parameter is specialized on your class. Listing 22-6 shows a `clone` method that handles cloning of rectangles.

Listing 22-6 A `clone` function

```
define method clone (r :: <rect>) => result :: <rect>;
  make(<rect>, top: r.top, left: r.left,
       bottom: r.bottom, right: r.right);
end method;
```

Using Shared and Default Objects

If an object can be used by several other objects, you can specify that it is to be shared. If an object can be shared, it is created automatically before the application starts to execute.

IMPORTANT
If a large object could be shared but it is rarely used, you may wish not to share it because the Framework guarantees that a sharable object always exists. ▲

You can specify `shareable: #t` in your call to `make-class-spec` to prevent an object from being written to a stream more than once. Listing 22-7 shows an example of the class specification for frame adorners. Because views can share adorners, an adorner needs to appear in a stream only once.

Listing 22-7 Specifying a class for shareable objects without slots

```
make-class-spec(<frame-adorner>, #"frame-adorner", #(), shareable: #t);
```

Note
Listing 22-7 also shows how to specify a class that does not have slots. Its slot list parameter is an empty list (`#()`). ◆

To make use of a shared object, you call the `default-instance` function wherever you would have created the object. Listing 22-8 shows a call to `default-instance`. Whenever a frame adorner is needed, you use to `$frame-adorner` instead of creating a `<frame-adorner>` object.

Listing 22-8 Using a default object

```
define constant $frame-adorner = default-instance(<frame-adorner>);
```

You could still create a `<frame-adorner>` object or call `clone` if, for some reason, you needed an unshared version.

Using Streams

The Framework provides stream classes that you can use without defining subclasses:

- file streams provide a stream whose source or sink is a file
- handle streams build a stream whose source or sink is a handle
- file object streams are file streams that support reading and writing objects
- handle object streams are handle streams that support reading and writing objects

The following sections show examples of using these kinds of streams.

Using Handle Streams

A handle stream allows you to manipulate the contents of a handle by reading or writing those contents. Reading from the stream allows you retrieve data from the handle. Writing to the stream allows you to add data to the handle.

To use a handle stream, you must first obtain a handle; for example, you can retrieve an existing handle or create a new one. After you obtain a handle, you can use it to create a stream object.

Listing 22-9 shows an example from the Framework that retrieves error messages from the application's resource fork. A handle containing the data is returned by the `get-ith-resource-from-file` method. The `read-error-list-resource` method is called to read the contents of the handle.

Listing 22-9 Retrieving the contents of a handle

```
define method read-error-lists (lib :: <framework-library>) => ()
    let count = count-resources-in-file(lib, $error-list-type);
    for (i from 1 to count)
      let error-list-handle
        = get-ith-resource-from-file(lib, $error-list-type, i);
```

```
      fail-nil(error-list-handle);
      HNoPurge(error-list-handle);
     // keep it non-purgeable so it is there when we have problems
      read-error-list-resource(error-list-handle);
    end for;
end method;
```

Listing 22-10 shows the `read-error-list-resource` method. After the handle is created, its contents are read into variables using `read-short-integer` to read two-byte integers and `read-handle-string` to read strings. The `align-word` method is called to ensure that the stream is positioned to the start of the next piece of data, since the resource contents are word-aligned.

Listing 22-10 Reading a handle as a stream

```
define constant read-error-list-resource =
     method (handle :: <machine-pointer>) => ()
       let stream = make(<handle-stream>, handle: handle);

       let count = read-short-integer(stream);
       for (i from 1 to count)
         let error-number = read-short-integer(stream);
         let reason-string = read-handle-string(stream);
         align-word(stream);
         let recovery-string = read-handle-string(stream);
         align-word(stream);

         element(*os-error-table*, error-number)
            := pair(reason-string, recovery-string);
       end for;
     end method;
```

Using Object Streams

Object streams use the `make-class-spec` and `make-slot-spec` forms to define the structure of objects that exist in the stream or written to a stream. For information about these setting up these forms, see “Setting up the Class and Slot Specifications” on page 508 and “Using Slot Specification Getters and

Setters" on page 509. The following sections show examples of how to implement streaming for objects.

Reading and Writing Objects

After you set up your `make-class-spec` forms, the objects specified by them may be written to streams and retrieved from streams. You must create a stream and write objects to it. After the stream contains an object, it may be read.

Listing 22-11 shows how to write an object to an object stream that uses a handle. The `<handle-object-stream>` object is created. It is then written to the stream and the handle is updated to reflect the change.

Listing 22-11 Writing an object to a stream

```
define method stream-out-window (window :: <window>,
                                 file :: <file>,
                                 resource-type :: <integer>,
                                 resource-id :: <integer>) => ()
  let stream = make(<handle-object-stream>);
  write-object(stream, window);

  AddResource(stream.stream-handle,
    resource-type, resource-id, "streamed window");
  WriteResource(stream.stream-handle);
  UpdateResFile(file.rsrc-ref-num);
end method;
```

Listing 22-12 shows reading a window from an object stream that uses a handle. The `<handle-object-stream>` object is created. The next object, which in the example is also the first object, is read from the stream.

Listing 22-12 Reading an object from a steam

```
define method stream-in-window (file :: <file>,
                                resource-type :: <integer>,
                                resource-id :: <integer>)
```

```
                                => window :: <window>;
  let handle = get-resource-from-file(file, resource-type, resource-id);
  fail-nil(handle);
  let stream = make(<handle-object-stream>, handle: handle);
  read-object(stream);
end method;
```

Reading and Writing Primitive Data Types

Use the `read-object` and `write-object` functions to access streams. These functions call `read-from-stream` and `write-to-stream`, respectively, to retrieve data from a stream or move data to a stream.

IMPORTANT

You typically do not need to define `read-from-stream` and `write-to-stream` functions; the Framework handles most cases for you. You only need to define an implementation to read and write primitive data types or collections that are not already handled by the Framework. ▲

The Framework provides `read-from-stream` and `write-to-stream` functions for objects as well as primitive data types for which slots are not defined. For example, the Framework provides `read-object` and `write-object` methods that are specialized for booleans because they can exist in the stream without information that identifies its class or slot.

The `read-from-stream` and `write-to-stream` functions implement reading and writing to the stream in a way that matches the `make-class-spec` specification. Thus, the functions must know the order of the data and its structure.

Listing 22-13 shows the `read-from-stream` method for the `<boolean>` class. It returns a true value (`#t`) if the next byte in the stream is non-null.

Listing 22-13 A `read-from-stream` function

```
make-class-spec(<boolean>, #"boolean", #());

define method read-from-stream (class-spec :: <class-spec>,
                                stream :: <stream>,
                                the-class == <boolean>,
```

```
                                class-version == 0) => result :: <point>;
    ignore(class-spec, the-class, class-version);

    read-byte(stream) ~= 0;
end method;
```

Listing 22-14 shows the `write-to-stream` function for the `<boolean>` class. It writes a 1 if the value is true (`#t`) or a 0 if it is false (`#f`).

Listing 22-14 The `write-to-stream` function

```
define method write-to-stream (class-spec :: <class-spec>,
                               stream :: <stream>,
                               b :: <small-point>) => ()
    ignore(class-spec);

    write-byte(stream, if (b) 1 else 0 end);
end method;
```

Stream Objects Reference

The following sections describe the classes and identify the functions that you can use with streams.

The <stream> Class

The `<stream>` class is an abstract class from which you can define subclasses. Listing 22-15 shows the class definition for the `<stream>` class.

Listing 22-15 The `<stream>` class definition

```
define primary open class <stream> (<object>)
end class;
```

Initialization

No `initialize` method is defined for `<stream>` objects.

The <file-stream> Class

The `<file-stream>` class represents a stream that is used to move data to and from a file. You create `<file-stream>` objects. You do not need to define subclasses of the `<file-stream>` class. Listing 22-15 shows the class definition for the `<file-stream>` class.

Listing 22-16 The `<file-stream>` class definition

```
define primary open class <file-stream> (<stream>)
  slot file :: <file>,
     required-init-keyword: file:;
  slot buffer :: <machine-pointer>,
     init-value: $null-machine-pointer;
  slot buffer-size :: <integer>,
     init-value: 1024,
     init-keyword: buffer-size:;
  slot bytes-in-buffer :: <integer>,
     init-value: 0;
  slot buffer-position :: <integer>,
     init-value: 0;
  slot stream-position :: <integer>,
     init-value: 0;
end class;
```

Slot descriptions

`file`	The file object associated with this stream. You must use the `file:` keyword to specify a file object.
`buffer`	The buffer that holds the stream's contents in memory. The Framework sets up the buffer for you.
`buffer-size`	The size of the buffer. Use the buffer: keyword to specify the size; otherwise the buffer size is 1,024 bytes.
`bytes-in-buffer`	The current number of bytes in the buffer. The Framework maintains this slot for you. You can call `bytes-in-buffer` if

	you want to know the number of bytes left to read or write in the buffer.
`buffer-position`	The current location to read from or write to in the buffer. The Framework maintains this slot for you. You can call `buffer-position` if you want to know the position.
`stream-position`	The current location to read from or write to in the stream. The Framework maintains this slot for you. You can call `stream-position` if you want to know the position.

Initialization

The `initialize` method for `<file-stream>` objects allocates a buffer in memory. The `initialize` method's parameters are defined as follows:

```
define method initialize (stream :: <file-stream>, #key) => ()
```

`stream`	The file stream.

The <handle-stream> Class

The `<handle-stream>` class represents a stream that is used to move data to and from a handle. You create `<handle-stream>` objects. You do not need to define subclasses of the `<handle-stream>` class. Listing 22-15 shows the class definition for the `<handle-stream>` class.

Listing 22-17 The `<handle-stream>` class definition

```
define primary sealed class <handle-stream> (<stream>)
  slot stream-handle :: <machine-pointer>,
     init-value: $null-machine-pointer;
  slot handle-size :: <integer>,
     init-value: 0;
  slot stream-position :: <integer>,
     init-value: 0;
  slot owns-handle? :: <boolean>,
     init-value: #f,
     init-keyword: owns-handle:;
  slot allocation-size :: <integer>,
```

```
    init-value: 16,
    init-keyword: allocation-size:;
end class;
```

Slot descriptions

`stream-handle` The handle associated with this stream. Use the `handle:` keyword to specify the handle. For more information, see the "Initialization" section, below.

`handle-size` The size of the handle. If the handle exists, the size is its actual size; if it is a new handle, the size is its allocation size. You can call `handle-size` to determine the current size of a handle associated with a stream; you can call `reallocate` to change the size.

`stream-position` The current location to read from or write to in the stream. The Framework maintains this slot for you. You can call `stream-position` if you want to know the position.

`own-handle?` Specifies whether or not the stream owns the handle and, thus, can dispose of the handle on termination (`#t`) or not (`#f`). Use the `owns-handle:` keyword to specify that the stream owns its handle (`#t`); otherwise, the slot's value is (`#f`), meaning that the stream is being created for a handle that it cannot dispose of itself.

`allocation-size` The size by which to increment the handle when more space is needed. Use the `allocation-size:` keyword to specify the size in bytes; otherwise, the size of the handle will be increased 16 bytes each time more space is needed. You can also call the getter and setter to access this slot.

Initialization

The `initialize` method for `<handle-stream>` objects creates a new handle, if needed, and sets up the handle size. The `initialize` method's parameters are defined as follows:

```
define method initialize (stream :: <handle-stream>,
                          #key handle: the-handle) => ()
```

`stream` The handle stream.

`the-handle` The handle associated with the stream. Use the `handle:` keyword to specify the handle; otherwise a new handle is created.

Object Stream Classes

The object stream classes are `<object-stream>`, `<handle-object-stream>`, and `<file-object-stream>`. The `<object-stream>` class is an abstract class from which you can define subclasses or mix in with a concrete class to stream objects. You can create objects from the `<handle-object-stream>`, and `<file-object-stream>` classes. You do not need to define subclasses from them. To use an object stream class, you must provide a `make-class-spec` top-level form for each class you want to stream. For more information about using object streams, see "Using Object Streams" on page 514.

Listing 22-18 shows the class definition for object stream classes.

Listing 22-18 Object stream class definition

```
define free open class <object-stream> (<stream>)
  slot symbol-cache :: <stretchy-object-vector>,
    init-function: curry(make, <stretchy-object-vector>);
end class;

define class <handle-object-stream> (<object-stream>, <handle-stream>)
end class;

define class <file-object-stream> (<object-stream>, <file-stream>)
end class;
```

Slot descriptions

`symbol-cache` Internal.

Initialization

No `initialize` method is defined for `<object-stream>` objects.

The <class-spec> Class

The <class-spec> class represents a template for an object in a stream. It also controls how an object is cloned. You create a <class-spec> object from the make-class-spec top-level form. For an example, see "Setting up the Class and Slot Specifications" on page 508. You do not create <class-spec> objects except by calling make-class-spec; you do not need to define subclasses from the <class-spec> class.

Listing 22-15 shows the class definition for the <class-spec> class.

Listing 22-19 The <class-spec> class definition

```
define primary sealed class <class-spec> (<object>)
  slot the-class :: <class>,
    required-init-keyword: class:;

  slot class-symbol :: <symbol>,
     required-init-keyword: class-symbol:;

  slot ws-template,
     init-value: #f;

  slot local-slots :: <list>,
    required-init-keyword: slots:;

  // cached slot list, includes superclasses
  slot all-slots :: <list>,
    init-value: #();

  slot shareable-instance,
    init-value: #f,
    init-keyword: shareable-instance:;
end class;
```

Slot descriptions

the-class	The name of the class. You must use the class: keyword to provide the name.

`class-symbol`	The identifying symbol of the class. You must use the `class-symbol:` keyword to provide the symbol.
`ws-template`	Internal.
`local-slots`	The slots defined in the class. You must use the `slots:` keyword and provide a list that contains a `<slot-spec>` object for each slot you wish to stream or clone.
`all-slots`	All slots for this class. The Framework maintains this list for you.
`shareable-instance`	An object from this class that can be shared. Use the `shareable-instance:` keyword to specify the object to use; otherwise, the Framework will use one created from this `<class-spec>` object.

Initialization

No `initialize` method is defined for `<class-spec>` objects.

The <slot-spec> Class

The `<slot-spec>` class represents a template for a slot in a stream. It also controls how an object is cloned.You create a `<slot-spec>` object by calling the `make-slot-spec` top-level form, which is used with the `make-class-spec` top-level form to define a class specification. For an example, see "Setting up the Class and Slot Specifications" on page 508. You do not create `<slot-spec>` objects except by calling `make-slot-spec`; you do not need to define subclasses from the `<slot-spec>` class.

Listing 22-15 shows the class definition for the `<slot-spec>` class.

Listing 22-20 The `<slot-spec>` class definition

```
define primary sealed class <slot-spec> (<object>)
  slot slot-symbol :: <symbol>,
    required-init-keyword: slot-symbol:;

  slot getter-function :: <function>,
    required-init-keyword: slot-getter:;
```

```
  slot setter-function :: <function>,
     required-init-keyword: slot-setter:;

  slot ws-template,
     init-value: #f;

  // getter used for cloning.
  slot cloning-getter :: <function>,
    required-init-keyword: cloning-getter:;

  // set clone-slot to #f to have clone copy the reference
  // to an object instead of cloning the slots value.
  slot clone-slot?, init-value: #t,
    init-keyword: clone:;

  // set stream-slot to #f to prevent the slot from being streamed
  slot stream-slot?, init-value: #t,
    init-keyword: stream:;
end class;
```

Slot descriptions

`slot-symbol`	The initial keyword used to set the value of the slot; for example, `foo:` or `#"foo"`, which are equivalent. You must use the `slot-symbol:` keyword to specify the initial keyword.
`getter-function`	The function that obtains the value to write to a stream. You must use the `slot-getter:` keyword to specify the function.
`setter-function`	A function that the Framework uses internally. You must use the `slot-getter:` keyword to specify the function. You should specify `$no-setter` if you do not want to allow setting the value from the Apple Dylan User Interface Builder property editor.
`ws-template`	Internal.
`cloning-getter`	The function that obtains the value of the slot when making a clone of an object. Use the `cloning-getter:` keyword to specify the function; otherwise, the Framework uses the function specified in the

	`getter-function` slot when cloning an object, meaning that the slot is cloned the same way it is streamed.
`clone-slot?`	Specifies whether the object referenced in a slot is actually cloned (`#t`) or whether only the reference is copied to the new object (`#f`) when an object is cloned. Use the `clone:` keyword and specify false (`#f`) if you do not wish to clone the object referenced by a slot; otherwise, the reference is cloned, which creates a new object instead of a new reference to the existing object.
`stream-slot?`	Specifies whether the slot is written to a stream (`#t`) or not (`#f`). Use the `stream:` keyword and specify false (`#f`) if you do not want to stream this slot; otherwise, the slot is streamed.

Initialization

No `initialize` method is defined for `<slot-spec>` objects.

Functions for Streams

The following functions can be used with streams.

Table 22-1 Functions for streams

Function	Purpose	Call	Sp.
`read-byte`	Read a byte from a stream.	√	√
`write-byte`	Write a byte to a stream.	√	√
`read-bytes`	Read the specified number of bytes from a stream.	√	√
`write-bytes`	Write the specified number of bytes to a stream.	√	√
`peek-byte`	Examine the contents of a byte in a stream.	√	√
`end-of-stream?`	Determine whether the end of a stream has been reached.	√	√

Table 22-1 Functions for streams

Function	Purpose	Call	Sp.
`stream-position`	Determine the current position in a stream.	√	√
`flush-stream`	Finishing writing the stream.	√	√
`read-short-integer`	Read two bytes into an integer.	√	√
`write-short-integer`	Write two bytes from an integer.	√	√
`read-long-integer`	Read four bytes into an integer.	√	√
`write-long-integer`	Write four bytes from an integer.	√	√
`read-string`	Read a string.	√	√
`write-string`	Write a string.	√	√
`read-pstring`	Read a Pascal-style string.	√	√
`write-pstring`	Write a Pascal-style string.	√	√
`read-symbol`	Read a symbol.	√	√
`write-symbol`	Write a symbol.	√	√
`terminate`	Terminate a stream.	√	√
`reset-stream`	Discard the contents of a stream and set the position to the beginning of the stream.	√	√

The following functions can be used with file streams.

Table 22-2 Functions for file streams

Function	Purpose	Call	Sp.
`read-buffer`	Internal. Read the specified number of bytes from a stream.		
`write-buffer`	Internal. Write the specified number of bytes to a stream.		

The following functions can be used with handle streams.

Table 22-3 Functions for handle streams

Function	Purpose	Call	Sp.
`read-handle-string`	Read a string from a handle.	√	
`align-word`	Set the stream position to a word boundary.	√	
`reallocate`	Internal. Increase (or decrease) the size of a handle.		

The following functions can be used with object streams.

Table 22-4 Functions for object streams

Function	Purpose	Call	Sp.
`get-class-spec`	Internal. Determine the class of an object.		
`read-object`	Read an object from a stream.	√	
`write-object`	Write an object to a stream.	√	
`read-from-stream`	Specify how to read an object from a stream.		√
`write-to-stream`	Specify how to write an object to a stream.		√
`write-slot`	Internal.Write a slot to a stream.		
`is-slot-editable?`	Internal. Determine if the contents of a slot can be set.		
`slot-tile`	Internal. Determine a slot's title.		
`clone`	Create a copy of an object.	√	√

The following functions can be used with class and slot specifications.

Table 22-5 Functions for class specifications

Function	Purpose	Call	Sp.
`slot-list`	Internal. Retrieve all slots in a class specification.		
`flush-slot-cache`	Internal.		
`make-class-spec`	Create a class specification object.	√	
`make-slot-spec`	Create a slot specification object.	√	
`shareable?`	Determine if an object can be shared.	√	
`class-streaming -version`	Specify the version of a class written to the stream.		√
`is-slot-editable?`	Internal. Determine if the contents of a slot can be set.		
`class-to-symbol`	Determine the identifying symbol of a class.	√	
`symbol-to-class`	Determine the class from its identifying symbol.	√	

CHAPTER 23

Commands

Contents

About Commands 531
Using Commands 534
  Defining a Command Class 534
  Doing a Command 535
  Undoing and Redoing a Command 535
  Committing a Command 536
  Completing a Command 537
  Performing a Command 537
Command Objects Reference 539
  The <command-context> Class 539
  The <command> Class 540
  Text Command Classes 541
    The <typing-command> Class 542
    The <text-style-command> Class 544
    The Font Command Classes 545
  Functions for Commands and Command Contexts 547

This chapter discusses commands, which allow you to package actions into a single operation. Typically commands are used to implement undo and redo for an operation, such as changing data in your application. Undo and redo are implemented within a context, called a command context, that limits the scope of the undo or redo operation. For example, a document might be the context in which a change can be made, undone, and redone.

About Commands

Commands allow you to specify how to perform a unit of work, called an operation, within an application. A command may be undoable, meaning that it can be undone or redone. By default commands are undoable; however, they are not required to be undoable.

Note
The Framework provides commands that handle changes to the clipboard and text for you automatically. For information about clipboard commands, see "Clipboard Command Classes" on page 596. Information about text commands, including information about commands for changing fonts and styles, is discussed in this chapter. ◆

For example, the user may draw a rectangle in a drawing application. Rather than drawing the rectangle directly, you can implement a command class to handle the operation. In this example, the operation may be a single line of code; however, it could also perform related tasks, such as drawing handles on the rectangle, as well.

An undoable command allows you to specify how to undo the operation, and how to redo it after it is undone. In the drawing example, undoing the command may simply involve invalidating the area in which the rectangle and its handles were drawn. It might also involve removing the data that specifies where to draw it from the document. Redoing the command may be as simple as preforming the original operation over again.

You specialize methods of your command class to specify the actions to take:

- to do or perform the operation
- to undo the operation

- to redo the operation

In addition, doing or performing the operation can be divided into two parts:

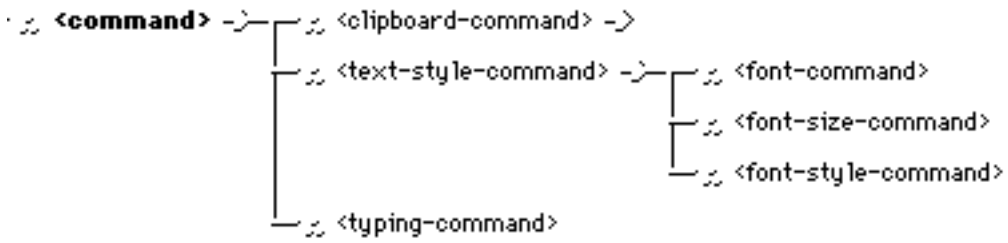
- actions to take to do or perform the operation
- actions to take immediately after performing the operation

There may be efficiency reasons that you do not want to completely reverse an operation when it is undone. This allows you to reduce the effort to redo the operation if that becomes necessary. You can specialize methods that specify when to:

- commit the operation, which prevents it from being undone. For example, you might allow changes to a database record to be made in memory and then commit the changes when the record is placed on disk.
- complete the operation, which allows you to perform actions after the operation is finished, whether or not the operation was undoable.

The Framework provides several command classes that automatically implement undo and redo for common operations. Figure 23-1 shows the `<command>` class and its subclasses.

Figure 23-1 Command classes



The classes shown in Figure 23-1 represent the following kinds of objects:

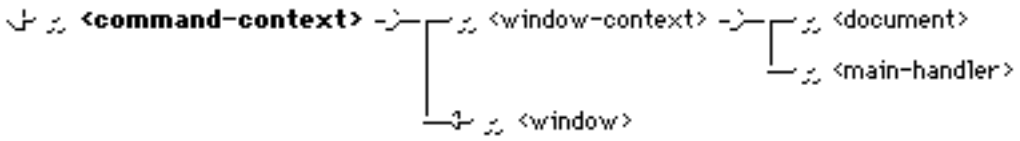
`<command>`	Represents a command that implements undo and redo operations.
`<clipboard-command>`	Represents a command that implements undo and redo operations for the clipboard.

`<text-style-command>`	Represents a command that implements undo and redo for text styles.
`<font-command>`	Represents a command that implements undo and redo for the kind of font in a text style.
`<font-size-command>`	Represents a command that implements undo and redo for the font size in a text style.
`<font-style-command>`	Represents a command that implements undo and redo for the font style (bold, italic, and so on) in a text style.
`<typing-command>`	Represents a command that implements undo and redo for text as it is being typed.

Commands operate within a context, called the command context. The Framework maintains information about the current command, called the last command, so that it can be used to undo an operation or redo it. Thus, when your application "does" a command, the Framework retains the command until it is committed. Thus, your command classes should maintain enough information about the state of the context so that a command may be undone or redone.

The Framework provides several command contexts. It is seldom necessary to provide other contexts, although you may wish to provide subclasses of command contexts for other reasons; for example, the document subclass is a sufficient context by itself, however it is also a class from which subclasses often are created. For more information about documents, see "About Documents" on page 479. Figure 23-2 shows the `<command-context>` class and its subclasses.

Figure 23-2 Command context classes



The classes shown in Figure 23-2 represent the following kinds of objects:

`<command-context>`	Represents objects whose state can be undone or redone.
`<window-context>`	Represents an event handler that manages one or more windows.
`<window>`	Represents a Macintosh window that allows changes to its contents to be undone or redone.
`<document>`	Represents the data in a view or file to which changes may be undone or redone.
`<main-handler>`	Represents an event handler of last resort, which can support undo and redo.

Using Commands

You can create commands that are not undoable simply by specifying false (`#f`) for the `undoable?` slot in your command subclass and specializing the `doit` method on the subclass. You can also specialize a `completed` method, if you wish.

The examples in the following sections show how to implement undoable commands. The following sections show how to:

- define a command class
- specify the actions to execute the command
- specify the actions to undo and redo the command
- specify how to commit the command
- perform a command

Defining a Command Class

You define a command class for each kind of action that you want to allow to be undone and redone. Your subclass needs to include slots that allow access to the data to be undone or redone. For example, Listing 23-1 shows the definition of a command subclass that allows a shape to be drawn. Because rendering will

not use the draw method, it is necessary to maintain information about the view in addition to information about the shape to be drawn. (To see how this state information is used, see Listing 23-2.)

Listing 23-1 Defining a `<command>` subclass

```
define class <draw-shape-command> (<command>)
  slot the-view :: <my-view>,
     required-init-keyword: view:;
  slot the-shape :: <my-shape>,
     required-init-keyword: shape:;
end class;
```

Doing a Command

The `doit` method specifies the actions to take when the command is executed. If drawing occurs while the command is being performed, you must focus the view being drawn. Listing 23-2 shows an example of a `doit` method.

Listing 23-2 Specializing the `doit` method

```
define method doit (command :: <draw-shape-command>) => ()
  with-focused-view(command.the-view)
    draw-shape(command.the-shape);
    if (command.the-shape.selected?)
      draw-handles(command.the-shape.shape-size);
    end if;
  end;
end method;
```

Undoing and Redoing a Command

The `undoit` method specifies the actions to take when the command is undone. Listing 23-3 shows an example of an `unoit` method.

Listing 23-3 Specializing the `undoit` method

```
define method undoit (command :: <draw-shape-command>) => ()
  invalidate(command.the-view,
             offset-rect(command.the-shape.shape-size, 3, 3));
end method;
```

The `redoit` method specifies the actions to take when the command is redone. In this example, the undo state is left so that the `doit` method can be executed without alteration. Listing 23-3 shows an example of an `unoit` method.

Listing 23-4 Specializing the `redoit` method

```
define method redoit (command :: <draw-shape-command>) => ()
  doit(command);
end method;
```

Committing a Command

When a new command is executed, the Framework commits the last command. The command's `commit` method is executed to make the effects of the command "permanent" in the sense that the command can no longer be undone or redone via the Edit menu's Undo and Redo items.

Listing 23-5 shows an example of the `commit` method that places the data in the data structure after it can no longer be undone.

Listing 23-5 Specializing the `commit` method

```
define method commit (command :: <draw-shape-command>) => ()
  command.the-view.the-document.data
     := insert(command.the-view.the-document.data,
               command.the-shape, last: #t);
end method;
```

Note
Because this data structure is in memory, it could have easily been manipulated by the `doit`, `undoit`, and `redoit` methods. If the data structure was expensive to manipulate, such as if it requires data base accesses, the `commit` method is the preferred way to handle the situation.

Completing a Command

You can use the `completed` method to specify actions to take after an undoable command has been committed or after a non-undoable command has been executed. Listing 23-6 shows and example of a `completed` method.

Listing 23-6 Specializing the `completed` method

```
define method completed (command :: <draw-shape-command>) => ()
  unless (command.context.shift-down?)
    command.the-shape.selected? := #f;
    invalidate(command.the-view,
               offset-rect(command.the-shape.shape-size, 3, 3));
  end unless;
end method;
```

Performing a Command

To perform a command, you must create the command object and then execute the `post-command` method. After executing the `post-command` method, the Framework automatically handles the Undo and Redo menu items until the next command is posted. Listing 23-7 shows the creation and posting of a command object, which occur after mouse tracking completes for a shape.

Listing 23-7 Creating and posting a command

```
define method behavior-event (tool :: <my-tool>,
                              next :: <list>,
                              view :: <my-view>,
                              event :: <mouse-down-event>,
                              id :: <symbol>) => ()
  ignore(next, event, id);
  let final-point = track-mouse(make(tool.tracker-class),
                                view, event.local-mouse);
  let draw-rect = make(<rect>);
  set-rect-from-points(draw-rect, event.local-mouse, final-point);
  let the-shape = make(tool.shape-type,
                       shape-size: draw-rect,
                       view: view);
  let context = get-command-context(view);
  context.shift-down? := event.shift-key?;
  let command = make(<draw-shape-command>,
                     context: context,
                     view: view,
                     shape: the-shape,
                     undo-name: concatenate-as (<string>,
                                                get-indexed-string($undo-strings-id,
                                                tool.tool-name),
                     redo-name: concatenate-as (<string>,
                                                get-indexed-string($redo-strings-id,
                                                tool.tool-name)
                    );

  post-command(*main-handler*, command);
end method;
```

Note that the `get-command-context` method is called to determine the view's command context. The undo and redo names are being determined from string resources.

Command Objects Reference

The following sections describe the classes related to commands.

The <command-context> Class

The `<command-context>` class represents a context in which undo and redo can supported. Examples of a command context are documents and windows. You typically do not define subclasses of `<command-context>` directly, rather you define subclasses of the `<document>` subclass instead. You might want to create a `<command-context>` object if you need the ability to undo or redo an action associated with an event handler that is not a window, document, or the main handler.

Listing 23-8 shows the definition of the `<command-context>` class.

Listing 23-8 The `<command-context>` class

```
define primary open class <command-context> (<event-handler>)
    slot children, init-value: #();
    slot last-command, init-value: #f;
end class;
```

Slot descriptions

`children`	Event handlers that are affected by undo and redo operations on the context object.
`last-command`	The last command that was done and can be undone.

Initialization

No `initialize` method is defined for `<command-context>` objects.

The <command> Class

A <command> class represents one or more actions that are bundled into an operation. An operation can be undone and redone unless you specify otherwise. You define subclasses for each set of actions you want to execute as an operation and create a command object from your subclass whenever you wish to perform the operation. The Framework executes the command, and undoes or redoes it as necessary, for you.

You specialize the parameters of methods on your command subclass to specify how to perform the operation (doit), how to undo it (undoit), and how to redo it (redoit), as well as to specify how to commit the operation so that it no longer can be undone (commit), and what to do after the operation is committed (completed).

The Framework provides several <command> subclasses that handle standard operations with text and the clipboard for you. For information about text classes, see the following section "Text Command Classes" on page 541. For information about clipboard command classes, see "Clipboard Command Classes" on page 596.

Figure 23-2 shows the class definition for the <command> class.

Listing 23-9 The <command> class

```
define primary open class <command> (<object>)
    slot context, :: <command-context>,
    setter: #f,
    required-init-keyword: context:;

  slot causes-change? :: <boolean>,
     init-value: #t;

  slot done? :: <boolean>,
     init-value: #f;
  slot undone? :: <boolean>,
     init-value: #f;

  slot undo-name :: <string>,
     init-keyword: undo-name:;
```

```
  slot redo-name :: <string>,
     init-keyword: redo-name:;

  slot recurring? :: <boolean>,
     init-value: #f;

  slot undoable? :: <boolean>,
     init-value: #t;
end class;
```

Slot descriptions

`context`	The context to which the command applies.
`causes-change?`	Whether the command causes a change (`#t`) or not (`#f`). For example, you can specify false (`#f`) if you do not want the Save dialog to appear after a command changes the appearance of a document, such as its display magnification, but not its content.
`done?`	Whether the command has been performed (`#t`) or not (`#f`).
`undone?`	Whether the command has been performed and then undone (`#t`) or not (`#f`).
`undo-name`	The string displayed in the Edit menu when the command can be undone.
`redo-name`	The string displayed in the Edit menu when the command can be redone.
`recurring?`	Whether the command should be kept on the queue of commands to execute (`#t`) or not (`#f`).
`undoable?`	Whether the command can be undone (`#t`) or not (`#f`).

Initialization

No `initialize` method is defined for `<command>` objects.

Text Command Classes

The Framework provides classes the handle common text manipulation operations for you:

- typing

- changing the text style
- changing the font name
- changing the font size
- changing the font style

Commands for each of these operations, by default, are undoable and, thus, can be undone or redone if necessary. The following sections discuss these text-related commands.

The <typing-command> Class

The `<typing-command>` class represents text as it is being typed. The Framework creates `<typing-command>` objects as needed. Specifically, the Framework creates a `<typing-command>` object when a keystroke is first pressed in a text view. The `<typing-command>` object allows the user to type, change, or delete characters until the user presses an arrow key, the user presses a command-key sequence, or until a `<mouse-down-event>` event occurs. After the user finishes typing, as indicated by one of these actions, the typing command may be undone or redone until another command object is created.

You do not need to define a subclass of the `<typing-command>` or create `<typing-command>` objects. Listing 23-10 shows the definition of the `<typing-command>` class.

Listing 23-10 The `<typing-command>` class

```
define primary open class <typing-command> (<command>)
  slot text-view :: <text-view>,
     setter: #f
     required-init-keyword: view:;

  slot old-style :: <StScrpHandle>;
  slot new-style :: <StScrpHandle>;

  inherited slot undo-name,
     init-value: $undo-typing-name;
  inherited slot redo-name,
     init-value: $redo-typing-name;
```

```
  slot old-selection-start :: <integer>;
  slot old-selection-stop :: <integer>;
  slot old-text :: <string>,
     init-value: "",
     init-keyword: old-text:;
  slot new-selection-start :: <integer>;
  slot new-selection-stop :: <integer>;
  slot new-text :: <string>,
     init-value: "";

  slot text-changed? :: <boolean>,
     init-value: #f;

  slot selected-entire-contents? :: <boolean>;
end class;
```

Slot descriptions

`text-view` The view that receives keystrokes while typing.

`old-style` The previous text style.

`new-style` The current text style.

`undo-name` The string displayed in the Edit menu when the command can be undone.

`redo-name` The string displayed in the Edit menu when the command can be redone.

`old-selection-start` The beginning of the previous set of characters, which are used to undo or redo the command.

`old-selection-stop` The end of the previous set of characters, which are used to undo or redo the command.

`new-selection-start` The beginning of the current set of characters that can be undone and redone.

`new-selection-stop` The end of the current set of characters that can be undone and redone.

`new-text` The text added while processing the command.

`text-changed?` Whether the text changed (`#t`) or not (`#f`) while typing.

`selected-entire-contents?`
Whether all the text in the view is selected (`#t`) or not (`#f`).

Initialization

The `initialize` method for `<typing-command>` objects determines the current state of the selection within the text view before typing occurs so that the command can be undone. The `initialize` method's parameters are defined as follows:

```
define method initialize (command :: <typing-command>, #key) => ()
```

`command` The typing command.

The <text-style-command> Class

The `<text-style-command>` class represents a command to change a text style. The Framework provides subclasses that change the font, font size, and font style for you. You can define additional subclasses of the `<text-style-command>` class to represent specific commands that you want to reuse frequently, or you can create `<text-style-command>` objects and use them without defining a subclass.

Listing 23-11 shows the definition of the `<text-style-command>` class.

Listing 23-11 The `<text-style-command>` class

```
define primary open class <text-style-command> (<command>)
  slot text-view :: <text-view>,
     setter: #f,
     required-init-keyword: view:;

  slot old-style :: <StScrpHandle>;
  slot new-style :: <StScrpHandle>;

  slot selection-start :: <integer>;
  slot selection-stop :: <integer>;
```

```
  slot style-action :: <function>,
     required-init-keyword: action:;
end class;
```

Slot descriptions

`text-view`	The view to which the command applies.
`old-style`	The previous text style.
`new-style`	The current text style.
`selection-start`	The beginning of the current set of characters whose style can be undone and redone.
`selection-stop`	The end of the current set of characters whose style can be undone and redone.
`style-action`	The function that specifies how to change the style.

Initialization

The `initialize` method for `<text-style-command>` objects determines the current state of the selection within the text view so that the command can be undone. The `initialize` method's parameters are defined as follows:

```
define method initialize (command :: <text-style-command>, #key) => ()
```

`command`	The text style command.

The Font Command Classes

The Framework provides command classes that represent a change to the font, font size, or font style for selected text. Respectively, these classes are `<font-command>`, `<font-size-command>`, and `<font-style-command>`. You do not need to define subclasses from these classes. You can create objects directly from them whenever you need to change the font characteristics of selected text.

Listing 23-12 shows the definition of the `<font-command>`, `<font-size-command>`, and `<font-style-command>` classes.

Listing 23-12 The <font-command>, <font-size-command>, and <font-style-command> classes

```
define primary sealed class <font-command> (<text-style-command>)
  inherited slot undo-name,
     init-value: $undo-font-name;
  inherited slot redo-name,
     init-value: $redo-font-name;
end class;

define primary sealed class <font-size-command> (<text-style-command>)
  inherited slot undo-name,
     init-value: $undo-font-size-name;
  inherited slot redo-name,
     init-value: $redo-font-size-name;
end class;

define primary sealed class <font-style-command> (<text-style-command>)
  inherited slot undo-name,
     init-value: $undo-font-style-name;
  inherited slot redo-name,
     init-value: $redo-font-style-name;
end class;
```

Slot descriptions

undo-name	The string displayed in the Edit menu when the command can be undone.
redo-name	The string displayed in the Edit menu when the command can be redone.

Initialization

No initialize method is defined for <font-command>, <font-size-command>, and <font-style-command> objects.

Functions for Commands and Command Contexts

Table 23-1 shows functions that can be used with command contexts.

Table 23-1 Functions for command contexts

Function	Purpose	Call	Sp.
`close`	Internal. Close the children of the context.		
`add-child`	Internal. Add an object to the context.		
`remove-child`	Internal. Remove an object from the context.		
`get-command-context`	Determine the context for a command.	√	
`did-command`	Internal. Marks the command as being done.		
`undid-command`	Internal. Marks the command as being undone.		
`redid-command`	Internal. Marks the command as being done.		
`perform-command`	Internal. Executes the command.		
`undo-redo`	Internal. Handles undo and redo for a command in its context.		

Table 23-1 shows functions that can be used with commands.

Table 23-2 Functions for commands

Function	Purpose	Call	Sp.
`doit`	Specify the actions to take to process the command.		√
`undoit`	Specify the actions to take to undo processing for the command.		√
`redoit`	Specify the actions to take to redo processing for the command.		√
`commit`	Specify the actions to take when after command can no longer be undone.		√
`completed`	Specify the actions to take for an undoable command (one that could never be undone).		√
`post-command`	Place a command on the execution queue.	√	

Table 23-1 shows functions that can be used with typing commands.

Table 23-3 Functions for typing commands

Function	Purpose	Call	Sp.
`add-character`	Internal. Add a character to the typing command associated with a text view.		
`done-typing`	Internal.		

CHAPTER 24

Data Interchange

Contents

About Data Interchange 551
Using Data Interchange Classes 554
  Using Typed Data 554
    Using Typed Handles 554
    Using Typed Pointers 555
    Using Data Items 555
  Using Designators 556
  Creating Designator Objects 557
  Defining Designator Subclasses 557
Data Interchange Objects Reference 558
  The <typed-data> Class 558
  The <typed-pointer> Class 559
  The <typed-handle> Class 560
  The <data-item> Class 560
  Designator Classes 561
    The <designator> class 561
    The <selection-designator> class 562
    The <text-designator> class 563
    The <character-designator> class 563
    The <range-designator> Class 564
    The <offset-designator> Class 564
  Functions for Data Interchange 566

This chapter describes the classes that represent in-memory data structures supported by the Framework. This chapter also describes how to reference subsets of data through the use of designators. The classes representing data structures and designators are used by the Framework to implement data interchange between the application and the clipboard, data interchange via drag and drop, as well as data interchange via Apple events.

About Data Interchange

The Framework supports data interchange, such as that which occurs using the clipboard, drag, and drop, or Apple events with two kinds of classes:

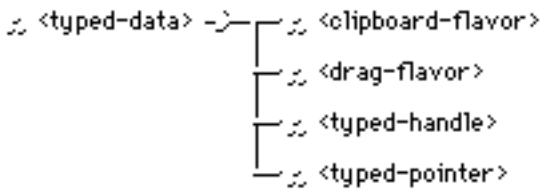
- typed data classes, in which the data type is specified. The type of data is a **flavor** that specifies the format of the data. The actual data is represented in a particular flavor and is referenced typically via a pointer or handle that is stored along with information about the kind of flavor.
- data item classes, which are used to represent a collection of flavors. In other words, a data item is a list one or more typed data objects.

In addition to data items and the typed data from which they are composed, the Framework provides a way of referring to part of data item. The part of a data item which is of interest can be designated; thus, objects that represent a designated part of a data item are called **designators**. Designators can be used to specify the selected items in a window, such as selected text, and for other purposes in which a subset of the data needs to be referenced.

Designators can be used for other kinds of data, in addition to data items and typed data. Often they are used in conjunction with Apple events. For example, the Framework provides property designators to refer to properties of Apple event descriptor records. (Property objects are discussed in “Property Designator Classes” on page 615.) Designators can be used for more than one kind of data, as well. For example, range designators that represent a sequence of bytes can be used to designate part of some typed data or a part of a property in an Apple event descriptor record.

Figure 24-1 shows the typed data classes.

Figure 24-1 Typed data classes

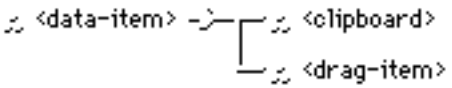


The classes shown in Figure 24-1 represent the following kinds of objects:

`<typed-data>`	Represents data in which the data type is specified. For more information about the `<typed-data>` class, see "The <typed-data> Class" on page 558.
`<clipboard-flavor>`	Represents data in the clipboard. For more information about the `<clipboard-flavor>` class, see "The <clipboard-flavor> Class" on page 595.
`<drag-flavor>`	Represents data being dragged. For more information about the `<drag-flavor>` class, see "The <drag-flavor> Class" on page 579.
`<typed-handle>`	Represents typed data that resides in memory and is accessible via a handle. For more information about the `<typed-data>` class, see "The <typed-handle> Class" on page 560.
`<typed-pointer>`	Represents typed data that resides in memory and is accessible via a pointer. For more information about the `<typed-data>` class, see "The <typed-pointer> Class" on page 559.

Figure 24-2 shows the data item classes.

Figure 24-2 Data item classes

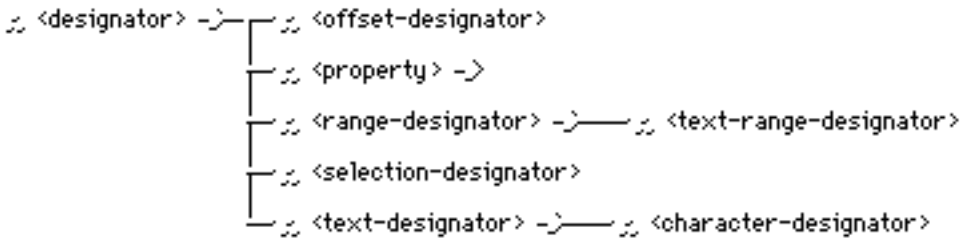


The classes shown in Figure 24-2 represent the following kinds of objects:

`<data-item>`	Represents data in one or more data type formats called flavors. A flavor is typically represented by a subclass of `<typed-data>`. For more information about the `<data-item>` class, see "The <data-item> Class" on page 560.
`<clipboard>`	Represents the flavors of data associated with the clipboard. For more information about the `<clipboard>` class, see "The <clipboard> Class" on page 594.
`<drag-item>`	Represents an item in a drag operation. For more information about the `<drag-item>` class, see "The <drag-item> Class" on page 578.

Figure 24-3 show the designator classes.

Figure 24-3 Designator classes



The classes shown in Figure 24-3 represent the following kinds of objects:

`<designator>`	Represents a designation of a subset of data.
`<offset-designator>`	Represents a subset of data by offset from the beginning of the data.
`<property>`	Represents a subset of data by its Apple event property. For more information about the `<property>` class, see "The <property> Class" on page 616.
`<range-designator>`	Represents a subset of data that is a range of bytes.
`<text-range-designator>`	Represents a subset of text that is a range of bytes.

`<selection-designator>`	Represents a designation of selected data.
`<text-designator>`	Represents a designation of text.
`<character-designator>`	Represents a designation of a character.

Using Data Interchange Classes

The following sections show examples of how to use typed data and designators. For additional examples that use typed data and designators to implement clipboard support, see "Using the Clipboard" on page 588.

Using Typed Data

The following sections show how to create `<typed-handle>` and `<typed-pointer>` objects.

Using Typed Handles

Typed handle objects are used to refer to data that is accessible by a handle. Listing 24-1 shows part of a Framework method that writes an object to a handle stream and then creates a `<typed-handle>` object so that the data can be referenced as a drag flavor.

Listing 24-1 Creating a typed handle

```
let stream = make(<handle-object-stream>);
write-object(stream, view);
let stream-data = make(<typed-handle>, data-type: ostype("dvew"),
                       data-handle: stream.stream-handle);

make(<drag-flavor>, item: item, flavor-data: stream-data);
```

Using Typed Pointers

Typed pointer objects are used to refer to data that is accessible by a pointer. Listing 24-2 shows a Framework method that creates a typed pointer object for a reference to an element of an array whose address is specified in `index`.

Listing 24-2 Creating a typed pointer

```
define method make-object-reference-flavor (item :: <drag-item>,
                                            object :: <object>,
                                            flavor-type :: <integer>)
                                            => result :: <integer>;
  let index = make-external-reference(object);

  with-struct(object-ref :: <machine-pointer> size 4)
    signed-long-at(object-ref) := index;
    let data = make(<typed-pointer>,
                    data-type: flavor-type,
                    data-pointer: object-ref,
                    data-length: 4);

    make(<drag-flavor>, item: item,
         flavor-data: data,
         flavor-flags: $flavorSenderOnly + $flavorNotSaved);
  end;

  index;
end method;
```

Using Data Items

A data item object is a list of typed data objects. For example, the clipboard can hold many kinds of data, each of which is represented by a typed data object. When data is pasted from the clipboard, the Framework builds a data item that includes only the data which is allowed to be pasted. Listing 24-3 shows how the Framework creates a `<data-item>` object from the flavors of data stored in the clipboard. The `get-flavors` method is called to locate the allowable data types.

Listing 24-3 Creating a data item of allowable flavors in the clipboard

```
define method doit (command :: <paste-command>) => ()
  local wants-type (a-type)
          can-receive-data?(command.target, a-type, *clipboard*)
        end method;

  // figure out which data types the target wants
  let types = choose(wants-type, *clipboard*.available-types);

  // get the flavors corresponding to the chosen types
  let paste-flavors = map(curry(get-flavor, *clipboard*), types);
  command.data-item := make(<data-item>, flavors: paste-flavors);

  command.redo-selection := get-selection(command.target);
  command.undo-selection := set-selection-data(command.target,
command.data-item);
end method;
```

Using Designators

You can use the Framework provided designator subclasses to designate data in the following ways:

- as a range of bytes
- as an offset from the start of data
- as a selection
- as a piece of text
- as characters

There are other ways to designate data, which are only used with Apple events. For information about Apple event-related designator subclasses, see “Property Designator Classes” on page 615. You can also define your own designator subclasses.

The following sections show examples of creating a designator object and defining a designator subclass.

Creating Designator Objects

You can create designators whenever you wish to identify a subset of data on a temporary basis. Listing 24-4 shows how to make a `<range-designator>` object that is used to designate the range of bytes in the currently selected text of an EditText record associated with a text view.

Listing 24-4 Creating a `<range-designator>` object

```
let designator = make(<range-designator>,
                      container: view,
                      start: view.te-handle.TERec$selStart,
                      stop: view.te-handle.TERec$selEnd);
```

The designator can then be used, for example to create a typed data object, as shown in Listing 24-5.

Listing 24-5 Using a `<range-designator>` object

```
let handle = get-text-range(view, designator.range-start,
                            designator.range-stop);
let text-flavor = make(<typed-handle>,
                       data-type: ostype("TEXT"),
                       data-handle: handle);
```

Defining Designator Subclasses

If you need to designate data that cannot be handled by a designator class provided by the Framework, you can define your own subclass. For example, the `<my-selection-designator>` designates objects in a data structure that is associated with a document.

Listing 24-6 Defining a <designator> subclass

```
define class <my-selection-designator> (<designator>)
    slot items :: <list>,
        init-keyword: items:,
        init-value: #();
end class;
```

When you define a subclass of `<designator>`, you may need to specialize the `is-empty?` method on your subclass. You may also need to specialize the `class-type` method. Listing 24-7 shows a specialization of the `is-empty?` method.

Listing 24-7 Specializing the `is-empty?` method

```
define method is-empty? (designator :: <my-selection-designator>)
                        => result :: <boolean>;
    empty?(designator.items);
end method;
```

Data Interchange Objects Reference

The following sections describe the classes related to data interchange.

The <typed-data> Class

The `<typed-data>` class represents data in an explicit format or data type. The data type is called a flavor. Subclasses of the `<typed-data>` class refer to actual data in the specified flavor and may provide additional information about the data, such as its size. The Framework provides several subclasses of `<typed-data>`. You can create objects from these subclasses or you can define your own subclasses of the `<typed-data>` class and create objects from your subclasses.

Listing 24-8 shows the definition of the `<typed-data>` class.

Listing 24-8 The <typed-data> class

```
define primary open class <typed-data> (<object>)
  slot data-type :: <integer>,
    init-keyword: data-type:;
end class;
```

Slot descriptions

data-type	The kind of data represented by the object, such as ostype("PICT").

Initialization

No initialize method is defined for <typed-data> objects.

The <typed-pointer> Class

The <typed-pointer> class represents a flavor of data which is referenced via a pointer. You can create objects from the <typed-pointer> class; you do not need to define subclasses from this class.

Listing 24-9 shows the definition of the <typed-pointer> class.

Listing 24-9 The <typed-pointer> class

```
define class <typed-pointer> (<typed-data>)
  slot data-pointer :: <machine-pointer>,
    setter: #f,
    required-init-keyword: data-pointer:;
  slot data-length :: <integer>,
    setter: #f,
    required-init-keyword: data-length:;
end class;
```

Slot descriptions

data-pointer	The beginning of the data associated with the object.
data-length	The length of the data.

Initialization

No `initialize` method is defined for `<typed-pointer>` objects.

The <typed-handle> Class

The `<typed-handle>` class represents a flavor of data which is referenced via a handle. You can create objects from the `<typed-handle>` class; you do not need to define subclasses from this class.

Listing 24-10 shows the definition of the `<typed-handle>` class.

Listing 24-10 The `<typed-handle>` class

```
define primary open class <typed-handle> (<typed-data>)
  slot data-handle :: <machine-pointer>,
    init-value: $null-machine-pointer,
    init-keyword: data-handle:;
end class;
```

Slot descriptions

`data-handle` The beginning of the data associated with the object.

Initialization

The parameters of the `initialize` method for `<typed-handle>` objects are defined as follows:

```
define method initialize (handle :: <typed-handle>, #key) => ()
```

`handle` The `<typed-handle>` object.

The <data-item> Class

The `<data-item>` class represents a list of flavors of data. The flavors are usually typed data objects. You do not need to define subclasses of the `<data-item>` class. You can create `<data-item>` objects and use them to refer to data that may have several representations or flavors.

Listing 24-11 shows the definition of the <data-item> class.

Listing 24-11 The <data-item> class

```
define primary open class <data-item> (<object>)
  slot flavors :: <list>,
    init-value: #(),
    init-keyword: flavors:;
end class;
```

Slot descriptions

flavors	A list of <typed-data> objects that identifies the various representations of the data item and, possibly, the data item's content in each representation.

Initialization

No initialize method is defined for <data-item> objects.

Designator Classes

Designator classes are used to mark or designate a subset of data. They are commonly used as a temporary reference to data associated with Apple events and can also be used more generally to designate any subset of data, including typed data.

The following sections describe the classes that designate data.

The <designator> class

The <designator> class is an abstract class from which subclasses that designate a subset of data are defined. The Framework provides subclasses of the <designator> class for use with data associated with Apple events. Many of these subclasses can also be used with other kinds of data, too.

You can define your own subclasses from the <designator> class and create objects from your subclasses, or you can use objects created from subclasses provided by the Framework. You should define a class-type method for each of your subclasses.

Listing 24-12 shows the definition of the <designator> class.

Listing 24-12 The <designator> class

```
define primary open class <designator> (<object>)
    slot container :: <object>,
        setter: #f,
        required-init-keyword: container:;
end class;
```

Slot descriptions

container	The object in which the designated data resides. You must use the container: and specify a container.

Initialization

No initialize method is defined for <designator> objects.

The <selection-designator> class

The <selection-designator> class represents a selected portion of data. You do not need to define subclasses from the <selection-designator> class, you can create a <selection-designator> object when you need to designate selected data. The class-type method for this class returns $cSelection.

Listing 24-13 shows the definition of the <selection-designator> class.

Listing 24-13 The <selection-designator> class

```
define primary open class <selection-designator> (<designator>)
end class;
```

Initialization

No initialize method is defined for <selection-designator> objects.

The <text-designator> class

The `<text-designator>` class represents textual data. You do not need to define subclasses from the `<text-designator>` class, you can create a `<text-designator>` object when you need to designate textual data. The `class-type` method for this class returns `$cText`.

Listing 24-14 shows the definition of the `<text-designator>` class.

Listing 24-14 The `<text-designator>` class

```
define primary open class <text-designator> (<designator>)
end class;
```

Initialization

No `initialize` method is defined for `<text-designator>` objects.

The <character-designator> class

The `<character-designator>` class represents one or more characters. You do not need to define subclasses from the `<character-designator>` class, you can create a `<character-designator>` object when you need to designate characters in data. The `class-type` method for this class returns `$cChar`.

Listing 24-15 shows the definition of the `<character-designator>` class.

Listing 24-15 The `<character-designator>` class

```
define primary sealed class <character-designator> (<text-designator>)
end class;
```

Initialization

No `initialize` method is defined for `<character-designator>` objects.

The <range-designator> Class

The `<range-designator>` class represents a range of bytes in the data. You do not need to define subclasses from the `<range-designator>` class, you can create a `<range-designator>` object when you need to designate a range of bytes.

Listing 24-16 shows the definition of the `<range-designator>` class.

Listing 24-16 The `<range-designator>` class

```
define primary sealed class <range-designator> (<designator>)
  slot range-start :: <integer>,
     init-keyword: start:;
  slot range-stop :: <integer>,
     init-keyword: stop:;
end class;
```

Slot descriptions

`range-start`	The start of a range.
`range-stop`	The end of a range.

Initialization

The `initialize` method for `<range-designator>` objects sets the start and stop bytes of a range if the `range-specifier:` keyword is specified; otherwise the method does nothing. The `initialize` method's parameters are defined as follows:

```
define method initialize (designator :: <range-designator>,
                          #key range-specifier: specifier) => ()
```

`designator`	The range designator.
`specifier`	The range specifier.

The <offset-designator> Class

The `<offset-designator>` class represents data offset from its starting location. You do not need to define subclasses from the `<offset-designator>` class, you

can create an `<offset-designator>` object when you need to designate bytes of data offset from a starting point.

Listing 24-17 shows the definition of the `<offset-designator>` class.

Listing 24-17 The `<offset-designator>` class

```
define primary sealed class <offset-designator> (<designator>)
  slot offset :: <integer>,
     init-keyword: offset:;
end class;
```

Slot descriptions

`offset` An offset within an object.

Initialization

The `initialize` method for `<offset-designator>` objects sets the offset from the object associated with the `offset-specifier:` keyword. The `initialize` method's parameters are defined as follows:

```
define method initialize (designator :: <offset-designator>,
                          #key offset-specifier: specifier) => ()
```

`designator` The offset designator.

`specifier` The offset specifier.

Functions for Data Interchange

Table 24-1 shows functions that can be used to read and write typed data.

Table 24-1 Functions for typed data

Function	Purpose	Call	Sp.
`read-from`	Read the data in the specified representation.	√	√
`write-to`	Write the data with the specified representation.	√	√
`data-length`	Determine the length of the data	√	
`get-data-handle`	Retrieve the data as a handle.	√	

Table 24-2 shows additional functions that can be used with typed handles.

Table 24-2 Functions for typed handles

Function	Purpose	Call	Sp.
`terminate`	Internal. Dispose of the Macintosh handle associated with a `<typed-handle>` object.		
`resize-handle`	Internal. Set the handle size of the Macintosh handle associated with a `<typed-handle>` object.		

Table 24-3 shows functions that can be used with data items.

Table 24-3 Functions for data items

Function	Purpose	Call	Sp.
`available-types`	Determine the available data types for a data item.	√	
`flavor-available?`	Determine whether a specific data type is available for the data item.	√	
`get-flavor`	Retrieve the `<typed-data>` object for a specified data type in the data item.	√	

Table 24-4 shows functions that can be used to access data for exchange.

Table 24-4 Functions for designators

Function	Purpose	Call	Sp.
`get-selection`	Retrieve the selection from an object.		√
`set-selection`	Specify, perhaps visually, the selection within an object.		√
`get-data`	Retrieve data associated with an object.		√
`get-selection-data`	Retrieve the data from the selection.	√	
`set-selection-data`	Specify the data within a selection.		√
`delete-data`	Delete data associated with an object.		√
`delete-selection`	Delete the selected data.	√	
`can-receive-data?`	Specify wether or not an object can handle the proposed kind of data.		√

Table 24-5 shows functions that can be used with designators.

Table 24-5 Functions for designators

Function	Purpose	Call	Sp.
`class-type`	Specifies the Apple event class associated with a designator.	√	√
`is-empty?`	Determine if the designator refers to some content.	√	√

CHAPTER 25

Drag and Drop

Contents

About Drag and Drop 571
Using Drag and Drop Classes 572
Setting up a Drag 572
Setting up a Promise Function 574
Accepting a Drag 575
Receiving a Drop 576
Drag and Drop Objects Reference 576
The <drag> Class 577
The <drag-item> Class 578
The <drag-flavor> Class 579
Drag Event Classes 580
Functions for Drag and Drop 582

The Framework provides drag and drop support, which typically allows you to move data within or between windows. This chapter discusses the classes that support drag and drop and shows how to use them.

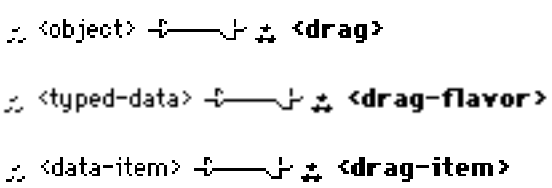
About Drag and Drop

The Framework provides drag and drop support in the following way:

- You can initiate a drag within your application. The drag operation is represented by a `<drag>` object. Associated with the drag operation are the items to be dragged, which are represented as `<drag-item>` objects.
- You can specify which views can accept a drag; for example, you can highlight the view as an item is dragged over it. The Framework provides a `<drag-flavor>` class that can be used to determine whether an item can be accepted by the view. The `<drag-flavor>` class is a subclass of the `<typed-data>` class. (For information about the `<typed-data>` class, see "The <typed-data> Class" on page 558.)
- You can specify how to receive a drop if the mouse button is released while over an object that can accept a drag.

Figure 25-1 shows the `<drag>`, `<drag-flavor>`, and `<drag-item>` classes.

Figure 25-1 Drag classes



The classes shown in Figure 25-1 represent the following kinds of objects:

`<drag>`	Represents a drag operation.
`<drag-flavor>`	Represents the kind of data being dragged.

`<drag-item>`	Represents an item, such as a view, associated with a drag operation.

When items are being dragged into or within the application's windows, the Framework creates drag events to track them. The Framework creates a `<track-drag-event>` object for a drag operation and a `<receive-drag-event>` object when a drag operation completes.

Using Drag and Drop Classes

The following sections show how to:

- set up a drag operation
- implement a promise function
- accept a drag
- receive a drop

Setting up a Drag

To set up a drag, execute, and terminate a drag operation, you need to take the following steps:

1. Create a `<drag>` object
2. Create a `<drag-item>` object for each item you want to drag
3. Create a `<drag-flavor>` object for each flavor of data associated with an item. The data associated with the flavor is either provided when the object is created or a function, called a **promise function**, is provided to retrieve the data when the drop is received. For information about promise functions, see "Setting up a Promise Function" on page 574.
4. Optionally, call `make-object-reference-flavor` for each flavor associated with each item. This is a convenience function used for dragging within the application's windows; for dragging out of the application, you must create `<drag-flavor>` objects.
5. Call `set-item-bounds` to set up the bounds for the drag feedback.

6. Call `start-drag` to initiate the drag operation.
7. Call `release-object-references` for each flavor of data specified in a call to `make-object-reference-flavor`.
8. Call `dispose` to release any other resources associated with the drag.

Listing 25-1 shows sample code that performs these steps.

Listing 25-1 Setting up a drag operation

```
define constant drag-these-views =
     method ( starting-view :: <view>,
              views-to-drag :: <list>,
              event :: <generic-mouse-down-event>,
              promise-function :: <function>) => ()
       // event should be the mouse down received by starting-view.

       let drag = make ( <drag>, view: starting-view );
       let references-to-free = #();

       for (view in views-to-drag)
         if (view.super-view) // don't stream the root view
           let item = new-item ( drag );

           let view-index = make-object-reference-flavor(item,
                                                         view,
                                                         ostype("view"));

           references-to-free := add(references-to-free, view-index);

           if (promise-function)
             make(<drag-flavor>, item: item,
                        promise: promise-function,
                        data-type: ostype("dvew"));
             else
                   let stream = make(<handle-object-stream>);
                   write-object(stream, view);
                   let stream-data = make(<typed-handle>, data-type:
                                   ostype("dvew"),
```

```
                                    data-handle: stream.stream-handle);

                make(<drag-flavor>, item: item,
                                     flavor-data: stream-data);
            end if;

            set-item-bounds(item, view, view.local-bounds);
          end if;
        end for;

        start-drag ( drag, event );
        do(release-object-reference, references-to-free);
        dispose ( drag );
      end method;
```

Setting up a Promise Function

If you want to delay moving the data to be dropped until the latest moment, you can create a promise function that implements a promise to drop the data. Listing 25-2 shows a promise function.

Listing 25-2 A promise function

```
define method promise-function ()
    let stream = make(<handle-object-stream>);
    write-object(stream, view);
    with-locked-handle(stream.stream-handle)
    block()
        let err = SetDragItemFlavorData(drag.drag-reference,
                                        item.item-reference,
                                        ostype("dvew"),
                                        pointer-at(stream.stream-handle),
        GetHandleSize(stream.stream-handle),0);
        if (err ~= $noErr)
            DebugStr("sdifd error");
        end;
        cleanup
        dispose(stream);
```

```
    end block;
    end;
end method promise-function;
```

Accepting a Drag

If your view can respond to a drag over it, you can specialize the `accept-drag?` method on your view, or you can implement a behavior and specialize the `behavior-accept-drag?` method. Listing 25-3 shows a `behavior-accept-drag?` method that allows the views to which the behavior is attached to accept local drags if the type of the flavor is `"mine"`.

Listing 25-3 A `behavior-accept-drag?` method

```
define method behavior-accept-drag? (behavior :: <my-drag-drop-behavior>,
                                     next-behaviors :: <list>,
                                     view :: <my-view>,
                                     drag :: <drag>)
  ignore ( behavior, next-behaviors );

  local check-flavor ( item )
          let addition-flavor = get-favor ( item, ostype ( "mine" ));
          if ( addition-flavor )
            view;
          end if;
        end method;

  let accept =  if ( drag.local? )
                  any? ( check-flavor, drag.item-list );
                end if;

  if ( accept )
    view;
  else
    next-method ();
  end if;
```

Receiving a Drop

If your view can accept a drag over it, it should also be able to receive a drop. To receive a drop, you can specialize the `receive-drop` method on your view, or you can implement a behavior and specialize the `behavior-receive-drop` method. Listing 25-3 shows a `behavior-receive-drop` method that allows the views to which the behavior is attached to receive the data if the type of the flavor is `"mine"`.

Listing 25-4 A `behavior-receive-drop` method

```
define method behavior-receive-drop (behavior :: <my-drag-drop-behavior>,
                                     next-behaviors :: <list>,
                                     view :: <my-view>,
                                     drag :: <drag>)
  ignore ( behavior, next-behaviors );

  // For the sake of legibility we will base the flavor on the first
  // item in the drag item list
  let firstItem = first ( drag.item-list );
  let additionFlavor = get-flavor ( firstItem, ostype ( "mine" ));

  // Handle the dragging of an addition to the view
  if ( additionFlavor )
    let draggedItem = get-referenced-object ( additionFlavor );
    view.my-data := draggedItem;
    end if;
  end if;
```

Drag and Drop Objects Reference

The following sections describe the classes and identify the functions that you can use to implement drag and drop support.

The <drag> Class

The <drag> class represents a drag operation. You do not need to define a subclass of the <drag> class; rather, you create a <drag> object for each drag operation. Listing 25-5 shows the class definition for the <drag> class.

Listing 25-5 The <drag> class

```
define primary open class <drag> (<object>)
  slot drag-reference :: <integer>,
     init-keyword: drag-reference:;
  slot item-list :: <list>,
    init-value: #();
  slot local? :: <boolean>,
     init-value: #t,
     init-keyword: local:;
  slot originating-view :: <view>,
    init-keyword: view:;
  slot current-view,
     init-value: #f;
  slot hilite-region, init-value: #f;
end class;
```

Slot descriptions

drag-reference	The drag reference used internally in calls to the Drag Manager. You do not need to set this slot.
item-list	The list of distinct items that can be dragged by this object.
local?	Whether the drag was initiated by this application (#t) or not (#f). If true (#t), the drag terminates when it enters a window that is not handled by the current drag handler. If false (#f), the drag is allowed to enter a window that is not handled by the current drag handler.
originating-view	The view from which the drag started.
current-view	The view over which the drag is currently tracked or false (#f) if the view cannot receive a drag.
hilite-region	The part of the current view to highlight when a drag occurs over the view.

Initialization

The `initialize` method for `<drag>` objects sets up the drag reference and drag items. The `initialize` method's parameters are defined as follows:

```
define method initialize (drag :: <drag>, #key) => ()
```

`drag` The `<drag>` object.

If the drag is local, `local: #t`, the `drag-reference:` keyword is ignored and a new drag reference is created. If the drag is not local, the drag reference is used to create `<drag-item>` objects for each item associated with the reference so that they may be received by a window.

The <drag-item> Class

The `<drag-item>` class represents an item in a drag operation. You do not need to define a subclass of the `<drag-item>` class. You create a `<drag-item>` object for each item that you want to associate with a drag if the drag is non-local. For local drags, the Framework creates `<drag-item>` objects for you. (See the `local?` slot description in the previous section for information about local and non-local drags.) Listing 25-5 shows the class definition for the `<drag-item>` class.

Listing 25-6 The `<drag-item>` class

```
define primary open class <drag-item> (<data-item>)
  slot item-reference :: <integer>,
    init-keyword: item-reference:;

  slot drag :: <drag>,
    required-init-keyword: drag:;

  slot promises :: <list>,
    init-value: #();
  end class;
```

Slot descriptions

`item-reference` The item reference number of the item being dragged.

drag	The <drag> object that is responsible for the item.
promises	A list of functions that set the drag item's flavor data.

Initialization

The initialize method for <drag-item> objects creates a <drag-flavor> object is for each of the item's flavors if the drag is a receiving drag. The initialize method's parameters are defined as follows:

```
define method initialize (item :: <drag-item>,
                   #key item-reference: item-ref,
                        receiving-drag: receiving) => ()
```

item	An item associated with a drag operation.
item-ref	The item reference number of the item being dragged, which is an item that is external to the application. Use the item-reference: keyword to specify the item reference number, otherwise, the item originates externally to the application.
receiving	Whether the item can be received by the destination (#t) or not (#f). Use the receiving-drag: keyword to specify whether this drag can move items to the destination (#t) or not (#f).

The <drag-flavor> Class

The <drag-flavor> class represents the data type of an item. You do not need to define a subclass of the <drag-flavor> class. You create a <drag-flavor> object for each kind of flavor in the drag item if the item is not part of a receiving drag; otherwise the Framework creates <drag-flavor> objects for you. Listing 25-5 shows the class definition for the <drag-flavor> class.

Listing 25-7 The <drag-flavor> class

```
define primary open class <drag-flavor> (<typed-data>)
  slot drag-item,
     setter: #f,
     required-init-keyword: item:;
  slot flavor-index :: <integer>;
```

```
  slot flavor-flags :: <integer>,
     init-value: 0,
     init-keyword: flavor-flags:;
end class;
```

Slot descriptions

`drag-item`	The item associated with the flavor of data.
`flavor-index`	The flavor index that identifies the flavor.
`flavor-flags`	The flavor flags, as defined in *Inside Macintosh*. You can use the values `$flavorSenderOnly`, `$flavorSenderTranslated`, `$flavorNotSaved`, and `$flavorSystemTranslated`.

Initialization

The `initialize` method for `<drag-flavor>` objects sets the flavor flags and adds the flavor (or its promise) to the drag. The `initialize` method's parameters are defined as follows:

```
define method initialize (flavor :: <drag-flavor>,
                    #key index: index,
                         promise: promise,
                         flavor-data: flavor-data) => ()
```

`flavor`	The flavor of data.
`promise`	The rules, such as a function, that specify now to set the data when it is received.
`flavor-data`	The data that can be received.

You cannot specify both the `promise:` and `flavor-data:` keywords.

Drag Event Classes

The `<drag-event>` class is an abstract class that represents a drag event. The Framework provides subclass that represent drag tracking receive dragging events. The `<drag-event>` class and its subclasses are internal to the Framework. You do not need to define additional subclasses of the `<drag-event>` class, nor do you need to create objects from these subclasses.

Listing 25-8 shows the class definition for the <drag-event>, <track-drag-event>, and <receive-drag-event> classes.

Listing 25-8 The <drag-event>, <track-drag-event>, and <receive-drag-event> classes

```
define primary sealed class <drag-event> (<mouse-event>)
  slot drag :: <drag>,
     setter: #f,
     required-init-keyword: drag:;
end class;

define primary sealed class <track-drag-event> (<drag-event>)
  slot track-drag-message,
     init-keyword: message:;
end class;

define primary sealed class <receive-drag-event> (<drag-event>)
end class;
```

Slot descriptions

`drag` The drag object associated with the event.

`track-drag-message` The status message from the Drag Manager.

Initialization

No `initialize` method is defined for <drag-event>, <track-drag-event>, or <receive-drag-event> objects.

Functions for Drag and Drop

Table 25-1 shows functions that can be used to support drag and drop operations.

Table 25-1 Functions for drag and drop

Function	Purpose	Call	Sp.
`drag-reference`	Retrieve the reference used by the Drag Manager given either an item reference or its flavor.	√	
`item-reference`	Retrieve the item reference given its flavor.	√	
`read-from`	Get the data given a flavor.		√
`data-length`	Determine the size of the data for a flavor.	√	
`write-to`	Set the data for a flavor.		√
`send-drag-data`	Internal. Dispatches the promise function.		
`make-object -reference-flavor`	Create a drag flavor from an object.	√	
`release-object -reference`	Release the drag flavor.	√	
`get-referenced -object`	Retrieve the object that is referenced as a flavor of data.	√	
`dispose`	Dispose of a drag, its items, and its flavors.	√	
`get-item-bounds`	Determine the bounding rectangle of a drag item.	√	
`set-item-bounds`	Set the bounding rectangle of a drag item.	√	
`new-item`	Create a drag item for a drag object.	√	

Table 25-1 Functions for drag and drop

Function	Purpose	Call	Sp.
`get-item`	Retrieve the referenced drag item from a drag object.	√	
`flavor-types`	Determine the flavors of data associated with a drag.	√	
`drag-mouse-moved?`	Determine if the mouse moved during a drag.	√	
`start-drag`	Initiate a drag from the application.	√	
`mouse-location`	Determine the current mouse location in a drag operation.	√	
`drag-origin`	Determine the starting mouse location in a drag operation.	√	
`modifiers`	Determine the keyboard modifiers during a drag operation.	√	

Table 25-2 shows functions that can be used to support drag and drop operations for views and windows.

Table 25-2 Functions that support drag and drop for views and windows

Function	Purpose	Call	Sp.
`drag-hilite`	Internal. Set drag highlighting on or off.		
`get-drag-hilite-region`	Internal. Determine the area that may be highlighted.		
`drag-enter-view`	Internal. Handles a drag entering a view.		
`drag-exit-view`	Internal. Handles a drag exiting a view.		
`handle-accept-drag?`	Internal. Allows behaviors and their event handlers to accept a drag.		

Table 25-2 Functions that support drag and drop for views and windows

Function	Purpose	Call	Sp.
`behavior-accept-drag?`	Specify whether the event handler associated with this behavior can accept a drag.		√
`accept-drag?`	Specify whether the event handler can accept a drag.		√
`handle-receive-drop`	Internal. Allows behaviors and their event handlers to receive the contents of a drag.		
`receive-drop`	Specify whether the event handler associated with this behavior can receive the contents of a drag.		√
`behavior-receive-drop`	Specify whether the event handler can receive the contents of a drag.		√

CHAPTER 26

Clipboard

Contents

About the Clipboard 587
Using the Clipboard 588
Setting Up the Clipboard Behavior 589
Specifying the Kind of Data to Paste 589
Setting up a Designator for Selected Data 590
Retrieving Data Specified by a Designator 591
Associating Clipboard Data with a View 591
Specifying the Selection 593
Specifying How to Delete Specified Data 593
Clipboard Objects Reference 594
The <clipboard-behavior> Class 594
The <clipboard> Class 594
The <clipboard-flavor> Class 595
Clipboard Command Classes 596
Functions for Clipboard Support 598

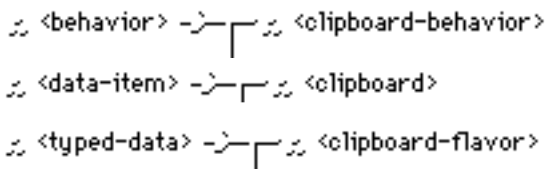
This chapter discusses the classes needed to implement clipboard support in your application. Transferring data between the application and the clipboard is a kind of data interchange, thus, you should be familiar with the classes described in Chapter 24, "Data Interchange," starting on page 551.

About the Clipboard

The Framework provides a behavior that implements clipboard support. This behavior handles the Copy, Cut, Paste, and Clear menu items. The Framework implements the behavior by creating Apple events and by responding to the events with commands. Most of the classes that implement this mechanism are internal to the Framework, however, and you do not need to use them directly to implement clipboard support. You only need to add the clipboard behavior to the views that you want to support clipboard operations, designate the data to be manipulated by the commands when they are executed, and specify the kind of data that can be pasted into a window's views.

Figure 26-1 shows the clipboard classes.

Figure 26-1 Clipboard classes

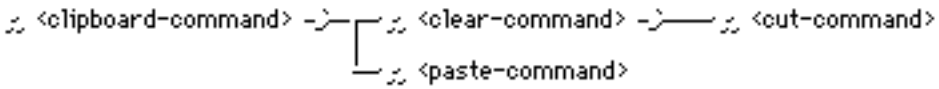


The classes shown in Figure 26-1 represent the following kinds of objects:

`<clipboard-behavior>`	Represents a behavior that handles clipboard operations of copy, cut, paste, and clear.
`<clipboard>`	Represents the clipboard.
`<clipboard-flavor>`	Represents the kind of data in the clipboard.

An Apple event may be sent to the application, or the Framework may create an event when a cut, paste, or clear menu item is selected. The Framework responds to the Apple event by creating a command object. The commands allow the operations to be undone and redone. Figure 26-2 shows the clipboard commands that the Framework creates in response to Apple events.

Figure 26-2 Clipboard command classes



The classes shown in Figure 26-2 represent the following kinds of objects:

`<clipboard-command>`	An abstract class that represents a command that handles a clipboard operation.
`<clear-command>`	Represents a command that clears the clipboard.
`<cut-command>`	Represents a command that cuts selected data from the clipboard.
`<paste-command>`	An abstract class that represents a command that pastes selected data into the clipboard.

Note
The Framework supports copying to the clipboard without creating a command object because this operation is not undoable. ◆

Using the Clipboard

All of the methods involved with clipboard operations are related to data interchange. The following sections show how to use these methods to implement clipboard support in your application. These tasks involve

- setting up the clipboard behavior for your view
- specifying the kind of data that can be pasted from the clipboard into a view.

- setting up a designator for selected data. This designator is used to specify the data that may be copied or cut to the clipboard.
- associating data with the designator
- retrieving data specified by the designator.
- setting the selection so that it visually reflects the clipboard operation.
- specifying how to delete the selected data.

The examples in the following sections show how the these tasks are handled by the Framework for the `<text-view>` class.

Setting Up the Clipboard Behavior

To implement Copy, Cut, Paste, and Clear menu items and their associated actions on the clipboard, you must add a clipboard behavior to each view that you wish to respond to these menu items. Listing 26-1 shows adding this behavior to the Framework's text view when the view is initialized.

Listing 26-1 Implementing a clipboard behavior

```
define method initialize (view :: <text-view>,
                          #key text: text-string,
                          text-style: style) => ()
    next-method();
    ...
    add-behavior(view, make(<clipboard-behavior>));
    ...
end method;
```

Specifying the Kind of Data to Paste

The Framework calls your view's `can-receive-data?` method to determine whether any of the flavors of data in the clipboard can be pasted into the view. The `can-receive-data?` method specifies the allowable flavors of data, which the Framework uses to build the data item to be pasted into the view.

For example, the Framework allows styled or non-styled text to be pasted into text views. Listing 26-2 shows how these flavors are specified.

Listing 26-2 The `can-receive-data?` method

```
define method can-receive-data? (view :: <text-view>,
                                 data-type :: <integer>,
                                 source :: <object>) => result :: <boolean>;
    ignore(view, source);

    data-type == ostype("TEXT") | data-type == ostype("styl");
end method;
```

Setting up a Designator for Selected Data

The Framework calls your view's `get-selection` method to determine what data is already selected so that the data can be restored when a paste, cut, or clear command is undone. You must specialize the `get-selection` method on your view so that the method returns a designator that specifies which data is currently selected.

Listing 26-3 shows the `get-selection` method for a text view.

Listing 26-3 Designating the current selection

```
define method get-selection (view :: <text-view>)
                                 => result :: <designator>;
    make(<range-designator>,
        container: view,
        start: view.te-handle.TERec$selStart,
        stop: view.te-handle.TERec$selEnd);
end method;
```

Retrieving Data Specified by a Designator

The Framework calls your view's `get-data` method so that the selected data can be removed from your view by a Cut or Clear command. The `get-data` method returns a `<data-item>` object. Listing 26-4 shows the `get-data` method specialized on the `<text-view>` class.

Listing 26-4 Retrieving data associated with a designator

```
define method get-data (view :: <text-view>,
                        designator :: <range-designator>)
                        => result :: <data-item>;
    let handle = get-text-range(view, designator.range-start,
                                designator.range-stop);
    let text-flavor = make(<typed-handle>,
                           data-type: ostype("TEXT"),
                           data-handle: handle);
    let flavor-list =
        if (use-styles?)
            ... // code to access styled text is not shown
            let scrap-handle = ...;
            list(make(<typed-handle>,
                data-type: ostype("styl"),
                data-handle: scrap-handle), text-flavor);
        else
            list(text-flavor);
        end if;

    make(<data-item>, flavors: flavor-list);
end method;
```

Associating Clipboard Data with a View

The Framework calls the view's `set-selection-data` method to associate data, for example data that has been pasted, with a view. The `set-selection-data` method specifies how to make the association; for example, by storing the data with other data for the view. The method returns a designator that specifies the

currently selected data after making the association. Listing 26-5 shows the `set-selection-data` method for text views.

Listing 26-5 Associating data with a view

```
define method set-selection-data (view :: <text-view>,
                                  data :: <data-item>)
                                => result :: <designator>;
    TEDelete(view.te-handle);

    let style-flavor =
        if (view.use-styles?)
            get-flavor(data, ostype("styl"));
        end if;

    local writer (ptr :: <machine-pointer>, length :: <integer>)
        if (style-flavor)
            TEStyleInsert(ptr, length, as(<StScrpHandle>,
                        get-data-handle(style-flavor)), view.te-handle);
        else
            TEInsert(ptr, length, view.te-handle);
        end if;
    end method;

    let (old-start, old-stop) = get-selection-range(view);
    ignore(old-stop);

    let text-flavor = get-flavor(data, ostype("TEXT"));
    fail-unless(text-flavor);
    write-to(text-flavor, writer, text-flavor.data-length);

    adjust-extent(view);

    make(<range-designator>,
        container: view,
        start: old-start,
        stop: old-start + text-flavor.data-length);
end method;
```

Specifying the Selection

The Framework calls the view's `set-selection` method to set the visual effects of the selection. Listing 26-6 shows the `set-selection` method for a text view, in which the selected range of text is being highlighted.

Listing 26-6 Setting the selection

```
define method set-selection (view :: <text-view>,
                             selection :: <range-designator>) => ()
    set-selection-range(view,
                        selection.range-start,
                        selection.range-stop);
end method;
```

Specifying How to Delete Specified Data

The Framework calls your view's `delete-data` method when selected data is cut or cleared. Listing 26-7 shows an example of this method.

Listing 26-7 Deleting data

```
define method delete-data ( view :: <text-view>,
                            designator :: <range-designator>) => ()
    ignore(view, designator);
    // Handled by delete-selection (special case)
end method;
```

This example is a special case due to the text being stored in a Macintosh text edit record, which makes it more efficient to specialize the `delete-selection` method instead of the `delete-data` method.

Clipboard Objects Reference

The following sections discuss the classes associated with the clipboard.

The <clipboard-behavior> Class

The `<clipboard-behavior>` class implements the behavior that responds to the Copy, Cut, Paste, and Clear menu items. You do not need to define subclasses from this class. You create a `<clipboard-behavior>` object and add it to the list of behaviors for each view that you want to support clipboard operations.

Listing 26-8 shows the class definition for the `<clipboard-behavior>` class.

Listing 26-8 The `<clipboard-behavior>` class

```
define primary open class <clipboard-behavior> (<behavior>)
  slot record-clipboard-commands :: <boolean>,
     init-value: #f,
     init-keyword: record-clipboard-commands:;
end class;
```

Slot descriptions

`record-clipboard-commands`
: Specifies whether you what to Cut, Paste, and Clear commands to be recordable (`#t`), or not (`#f`).

Initialization

No `initialize` method is defined for `<clipboard-behavior>` objects.

The <clipboard> Class

The `<clipboard>` class represents the contents of the clipboard. These contents are internal to the Framework. You do not need to define subclasses from this class, nor do you need to create objects from the class. The Framework creates a

<clipboard> object for you and associates it with the *clipboard* variable. You seldom need to access this variable.

Listing 26-8 shows the class definition for the <clipboard-behavior> class.

Listing 26-9 The <clipboard> class

```
define primary sealed class <clipboard> (<data-item>)
  slot clip-change-count :: <integer>,
     init-value: -1;
end class;
```

Slot descriptions

clip-change-count	The clipboard scrap count. The framework sets this slot for you. Do not change the value of this slot.

Initialization

No initialize method is defined for <clipboard> objects.

The <clipboard-flavor> Class

The <clipboard-flavor> class represents a flavor of data in the clipboard. You can define subclasses of the <clipboard-flavor> class and create objects from your subclass to handle transferring your flavor of data between your application and the clipboard.

Listing 26-10 The <clipboard-flavor> class

```
define primary open class <clipboard-flavor> (<typed-data>)
  slot data-length :: <integer>,
    required-init-keyword: data-length:;
end class;
```

Slot descriptions

data-length	The length of this flavor of data.

Initialization

No `initialize` method is defined for `<clipboard-flavor>` objects.

Clipboard Command Classes

The `<clipboard-command>` class is an abstract class that handles undo and redo for clipboard operations. The Framework provides classes that support undo and redo for the Cut, Paste, and Clear commands. These classes are `<cut-command>`, `<paste-command>`, and `<clear-command>` respectively. The Framework creates objects from these subclasses for you; you do not need to create them yourself.

You only need to define a subclass from the `<clipboard-command>` class if you want to support undo and redo for the Copy command. In this case, you also would need to create a `<copy-command>` object.

Listing 26-11 The `<clipboard-command>` class and its subclasses

```
define primary open class <clipboard-command> (<command>)
  slot target,
     required-init-keyword: target:;

  slot undo-selection :: <designator>;
  slot redo-selection :: <designator>;

  slot data-item :: <data-item>;
end class;

define primary open class <paste-command> (<clipboard-command>)
  inherited slot undo-name,
     init-value: $undo-paste-name;
  inherited slot redo-name,
     init-value: $redo-paste-name;
end class;

define primary open class <clear-command> (<clipboard-command>)
  inherited slot undo-name,
     init-value: $undo-clear-name;
```

```
  inherited slot redo-name,
     init-value: $redo-clear-name;
end class;

define primary open class <cut-command> (<clear-command>)
  inherited slot undo-name,
     init-value: $undo-cut-name;
  inherited slot redo-name,
     init-value: $redo-cut-name;
end class;
```

Slot descriptions

`target`	The event handler, such as a view, whose contents are affected by a clipboard command.
`undo-selection`	The designated data that is used to undo the command.
`redo-selection`	The designated data that is used to redo the command.
`data-item`	The clipboard contents.
`undo-name`	The name to appear when the Undo menu item is enabled.
`redo-name`	The name to appear when the Redo menu item is enabled.

Initialization

No `initialize` method is defined for clipboard command objects.

Functions for Clipboard Support

Table 26-1 shows functions that can be used to access data in the clipboard. These functions are all data interchange functions.

Table 26-1 Functions for clipboard support

Function	Purpose	Call	Sp.
`can-receive-data?`	Specifies whether an event handler can receive data of a particular type.		√
`get-data-handle`	Retrieve the clipboard flavor as a handle.	√	
`get-selection`	Retrieve the selection from the clipboard command's target object.		√
`set-selection-data`	Specify the data within a selection.		√
`set-selection`	Specify, perhaps visually, the selection within the clipboard command's target object.		√
`delete-data`	Delete data associated with the clipboard command's target object.		√
`get-data`	Retrieve data associated with the clipboard command's target object.		√

CHAPTER 27

Scripting Support

Contents

About Scripting Support 601
  Apple Events 602
  Data Structures 602
  Properties, Object Specifiers, and Descriptors 603
Using Scripting Support Classes 604
  Getting and Setting Property Values 604
  Responding to Apple Events 605
Scripting Support Objects Reference 607
  The Apple Event Classes 607
    The <ae-desc> class 607
    The <ae-list> class 608
    The <ae-record> class 609
    The <apple-event> class 610
    The <object-specifier> class 610
    Descriptor Classes 611
    The <token> class 612
  Scripting Event Classes 612
  Property Designator Classes 615
    The <property> Class 616
    The <document-property> Class 616
    The <file-property> Class 617
    The <text-view-property> Class 617
    The <text-view-selection-property> Class 618
    The <window-property> Class 618
  Scripting Support Functions 619

This chapter describes the scripting support available from the Framework's implementation of the Open Scripting Architecture (OSA). To use the scripting support, you must be familiar with OSA and Apple events as described in *Inside Macintosh: Interapplication Communication*. Apple event suites are described in the *Apple Event Registry*.

About Scripting Support

The Framework's scripting support includes the ability for:

- scripts or applications to send events to another application
- events to be created within the application to be sent and responded to by this application
- the application to respond to Apple events sent to it

The function that handles sending of Apple events is called `send-event`.

When an event is received, it is dispatched to the appropriate object by the main handler. If the object is dispatched to a window, the window's scripting object slot specifies the object that will respond to the event. If a scripting object is not specified, the event will be passed to the window's document. Thus, it is the responsibility of either a window or document to handle ab event sent to a window. For example, see "Using Scripting Support Classes" on page 604.

Apple Events

Your application can support any of the Apple events defined by the Framework and it can implement other Apple events or suites as well. Table 27-1 shows the events that are defined in the Framework.

Table 27-1 Apple event suites and events

Required Events	**Core Suite**	**Misc. Standards**
Open applicaiton	Clone	Copy
Open	Close	Cut
Print	Count elements	Make objects visible
Quit	Create element	Paste
	Delete	Revert
Core Suite	Do objects exist	Select
Move	Get class info	Redo
Save	Get data	Undo
Set data event	Get data size	
Get suite info	Get event info	

Data Structures

Many of the Framework's scripting support objects represent data structures in the Macintosh toolbox for locating data within a running application, such as

within a window or document. Table 27-2 summarizes the toolbox data structures and the corresponding Framework classes.

Table 27-2 Apple event data structures represented by Framework classes

Data Structure	Framework Class
`AEDesc`	`<ae-desc>`
`AEDescList`	`<ae-list>`
`AERecord`	`<ae-record>`
`AppleEvent`	`<apple-event>`

Properties, Object Specifiers, and Descriptors

A property designates the kind of data that an Apple event can access or change. The Framework handles access to the following properties for you:

- documents
- files
- windows
- text views
- text view selections

You can define subclasses of the Framework's `<property>` class and use objects from your subclasses to handle other properties. For an example, see "Getting and Setting Property Values" on page 604.

You can also locate data that is specified in a more complex way than by specifying only the property. The Framework provides the `<object-specifier>` class, whose objects allow you to locate data in an Apple event container. The Framework also provides descriptor classes that allow you to describe the relationship between specifiers. Using object specifiers with the descriptor objects, you can build a containment hierarchy. For information about the `<object-specifier>` class, see "The <object-specifier> class" on page 610. For information about descriptor classes, see "Descriptor Classes" on page 611.

Using Scripting Support Classes

The Framework supports the Open Scripting Architecture by allowing you to send and receive Apple events. The following sections show how to

- convert a Dylan object to and from an Apple event descriptor
- respond to the get and set data Apple events

Figure 27-1 shows a script that sends get data and set data Apple events to retrieve the contents of the front window of the application and set its contents. This script can be used to test your implementation of for responding to the events.

Figure 27-1 An Apple script

```
tell application "Application Nub"
    set message to the contents of the front window
    set dialogResult to display dialog "Set hello message to:" default answer message
    if button returned of dialogResult = "OK" then
        set the contents of the front window to the text returned of dialogResult
    end if
end tell
```

Getting and Setting Property Values

Listing 27-1 shows the `<hello-message-property>` subclass and the conversion methods for text in a static text view.

The `get-property-value` method specifies how convert a Dylan object to an Apple event descriptor record. The actual conversion is performed by the `make-descriptor` method. The `set-property-value` method specifies how convert an Apple event descriptor record to a Dylan object and sets the property. The conversion is performed by the `as-string` method.

The Framework provides functions for conversion of Apple event descriptor records to various kinds of Dylan objects. For a list of these conversion functions, see Table 27-5 on page 620.

Listing 27-1 Converting between Dylan objects and Apple event descriptor records

```
define class <hello-message-property> (<property>)
    slot text-view :: <view>,
    required-init-keyword: text-view:;
end class;

define method get-property-value (property :: <hello-message-property>,
                                  property-type == $pContents)
                                  => value :: <ae-desc>;
    ignore(property-type);

    make-descriptor(property.text-view.text);
end method;

define method set-property-value (property :: <hello-message-property>,
                                   property-type == $pContents,
                                   data :: <ae-desc>) => ()
    ignore(property-type);

    property.text-view.text := as-string(data);
    invalidate-all(property.text-view);
end method;
```

Note
The property's container is also invalidated in the `set-property-value` method so that the contents will be redrawn. ◆

Responding to Apple Events

Listing 27-2 shows the `<hello-scripting-object>` object that implements the response to the get and set data Apple events. The `get-property` method is executed when a get or set Apple event occurs and the window is in the target chain. This method creates the property object that specifies how the Apple event descriptor record is converted and handled.

Listing 27-2 Responding to an Apple event

```
define class <hello-scripting-object> (<object>)
    slot text-view :: <view>,
    required-init-keyword: text-view:;
end class;
```

Listing 27-3 shows the `get-property` method that is executed when a get or set Apple event occurs and the window associated with the scripting object is in the target chain. This method creates the property object that specifies how the Apple event descriptor record is converted and handled.

Listing 27-3 Responding to an Apple event

```
define method get-property (object :: <hello-scripting-object>,
                            property-type == $pContents)
                            => result :: <property>;
    make(<hello-message-property>,
        property-type: property-type,
        text-view: object.text-view,
        container: get-window(object.text-view));
end method;
```

The object is associated with a window, as shown in Listing 27-4.

Listing 27-4 Associating a scripting object with a window

```
define constant make-hello-window =
    method ()
        let static-text = make(<static-text>,
                               identifier: #"hello-text",
                               location: point(10, 10),
                               extent: point(200, 50),
                               text: get-string($hello-string-id));

        let window = make-scrolling-window(list(static-text),
                                location: point(50, 50),
```

```
                            extent: point(300, 200),
                            title: get-string($hello-window-title-id),
                            resizable: #t,
                            zoomable: #t,
                            closable: #t,
                            target-view: static-text);

        window.scripting-object := make(<hello-scripting-object>,
                                        text-view: static-text);

        open(window);
end method;
```

Scripting Support Objects Reference

The following sections discuss the classes that are used to implement scripting and identifies the functions that can be used to implement scripting

The Apple Event Classes

The following sections describe classes that are directly related to Apple event data structures. For more information about the data structures used with Apple events, see *Inside Macintosh: Interapplication Communication*.

The <ae-desc> class

The `<ae-desc>` class is an abstract class that represents an Apple event descriptor record. You can define subclasses of from the `<ae-desc>` class, although the Framework provides many subclasses from which you can create descriptor objects. You do not create objects from this subclass.

Listing 27-5 shows the `<ae-desc>` class definition.

Listing 27-5 The <ae-desc> class

```
define primary sealed class <ae-desc> (<object>)
  slot desc-type :: <integer>,  // really a DescType
     init-keyword: desc-type:;
  slot desc-handle :: <machine-pointer>,  // really a Handle
     init-value: $null-machine-pointer,
     init-keyword: desc-handle:;
end class;
```

Slot descriptions

desc-type	The kind of data.
desc-handle	A handle to the data itself.

Initialization

The initialize method for <ae-desc> objects allows Dylan to garbage collect the object as soon as it is no longer needed. The initialize method's parameters are defined as follows:

```
define method initialize (ae-desc :: <ae-desc>,
                          #key terminate: terminate) => ()
```

ae-desc	The Apple event descriptor object.
terminate	Specifies whether termination (#t) should be performed when the object is no longer used or not (#f).

The <ae-list> class

The <ae-list> class represents a list of Apple event descriptor records, such as those that specify the names to be opened by an Open Documents Apple event. You do not need to define subclasses from the <ae-list> class. You can create <ae-list> objects. You use <ae-list> objects in the same way that you use AEDescList data type in the Toolbox.

Listing 27-6 shows the <ae-list> class definition.

Listing 27-6 The <ae-list> class definition

```
define primary sealed class <ae-list> (<vector>, <ae-desc>)
end class;
```

Initialization

The initialize method for <ae-list> objects creates an Apple event descriptor list. The initialize method's parameters are defined as follows:

```
define method initialize (list :: <ae-list>,
                          #key ae-desc: ae-desc) => ()
```

list	The Apple event descriptor list object.
ae-desc	An Apple event descriptor record. Use the ae-desc: keyword to specify the kind of descriptor and the list's contents; otherwise, an empty descriptor list is created.

The <ae-record> class

The <ae-record> class represents a list of Apple event descriptor records that you want to refer to as a single object. You do not need to define subclasses from the <ae-record> class. You can create <ae-record> objects. You use <ae-record> objects in the same way that you use AERecord data type in the Toolbox.

Listing 27-7 shows the <ae-record> class definition.

Listing 27-7 The <ae-record> class definition

```
define primary sealed class <ae-record> (<ae-list>)
end class;
```

Initialization

No initialize method is defined for <ae-record> objects.

The <apple-event> class

The `<apple-event>` class represents a full-fledged Apple event. You do not need to define subclasses from the `<apple-event>` class. You can create `<apple-event>` objects. You use `<apple-event>` objects in the same way that you use `AppleEvent` data type in the Toolbox. The Framework creates and dispatches Apple Event objects that are received by the application automatically.

Listing 27-8 shows the `<apple-event>` class definition.

Listing 27-8 The `<apple-event>` class definition

```
define primary sealed class <apple-event> (<ae-record>)
end class;
```

Initialization

No `initialize` method is defined for `<apple-event>` objects.

The <object-specifier> class

The `<object-specifier>` class represents instructions for finding data (called Apple event objects) within a container. You do not need to define subclasses from the `<object-specifier>` class. You can create `<object-specifier>` objects. You use `<object-specifier>` objects in the same way that you use `typeObjectSpecifier` data type in the Toolbox. Typically, you call `make-object-specifier` to make an object specifier.

Listing 27-9 shows the `<object-specifier>` class definition.

Listing 27-9 The `<object-specifier>` class definition

```
define primary sealed class <object-specifier> (<ae-desc>)
end class;
```

Initialization

No `initialize` method is defined for `<object-specifier>` objects.

Descriptor Classes

The descriptor classes represent the form of the data within a container. You do not need to define subclasses from these descriptor classes. You can create objects. Typically, you one of the functions that create a descriptor object, such as `create-offset-descriptor` or `create-logical-descriptor`. For a complete list of these functions, see Table 27-6 on page 621.

Listing 27-10 shows the class definitions for the descriptor classes.

Listing 27-10 Apple event descriptor class definitions

```
define primary sealed class <relative-descriptor> (<ae-desc>)
end class;

define primary sealed class <index-descriptor> (<ae-desc>)
end class;

define primary sealed class <range-descriptor> (<ae-desc>)
end class;

define primary sealed class <logical-descriptor> (<ae-desc>)
end class;

define primary sealed class <comparison-descriptor> (<ae-desc>)
end class;

define primary sealed class <file-descriptor> (<ae-desc>)
end class;

define class primary sealed <null-descriptor> (<ae-desc>)
end class;
```

Initialization

No `initialize` method is defined for these descriptor objects.

The <token> class

The `<token>` class represents an Apple event descriptor record that identifies either an element in a container or a property of an Apple event object. You do not need to define subclasses from the `<token>` class. You can create `<token>` objects. Typically, you call `make-token` from your specialization of the `get-element` function to return a `<token>` object.

Listing 27-11 shows the `<token>` class definition.

Listing 27-11 The `<token>` subclass

```
define primary sealed class <token> (<ae-desc>)
end class;
```

Initialization

The `initialize` method for `<token>` objects sets up an entry in a table to track the token's external reference. The `initialize` method's parameters are defined as follows:

```
define method initialize (token :: <token>,
                          #key object: object,
                          token-class: the-class) => ()
```

`token`	The token object.
`object`	The Framework object that the token represents.
`the-class`	The Apple event class associated with the token.

Scripting Event Classes

The Framework defines an abstract class, `<scripting-event>`, that represents the occurrence of an Apple event. The Framework defines subclasses for core Apple events and other Apple events, as shown in Listing 27-13 on page 614. You can create objects from the subclasses provided by the Framework or from the classes you define.

Listing 27-12 shows the `<scripting-event>` class definition.

Listing 27-12 The <scripting-event> subclass

```
define primary open class <scripting-event> (<event>)
  slot message :: <ae-desc>,
     init-keyword: message:;

  slot reply :: <ae-desc>,
     init-keyword: reply:;
end class;
```

Slot descriptions

message	The data, representing a message, in an Apple event descriptor record.
reply	The data, representing a reply to a message, in an Apple event descriptor record.

Initialization

The initialize method for <scripting-event> objects creates an Apple event that corresponds to the <scripting-event> object. It also creates an <ae-desc> object for the target application if the target is not this application. The initialize method's parameters are defined as follows:

```
define method initialize (event :: <scripting-event>,
                          #key target: target-psn,
                          direct-object: direct-object) => ()
```

target-psn	The process serial number of the target machine's process. Use the target: serial number to specify the process; otherwise, the scripting event is directed towards this application.
direct-object	A direct parameter containing information for use by the server application.

You can define additional subclasses using the define-scripting-event macro. Three parameters must be specified, as follows:

1. The class name
2. The event class

3. The event ID

Listing 27-13 shows the definitions for the scripting event subclasses provided by the Framework.

Listing 27-13 Scripting event subclasses

```
define-scripting-event(<open-application-event>,
                       $kCoreEventClass, $kAEOpenApplication)

define-scripting-event(<open-event>, $kCoreEventClass, $kAEOpen)

define-scripting-event(<print-event>, $kCoreEventClass, $kAEPrint)

define-scripting-event(<quit-event>,
                       $kCoreEventClass, $kAEQuitApplication)

define-scripting-event(<clone-event>, $kAECoreSuite, $kAEClone)

define-scripting-event(<close-event>, $kAECoreSuite, $kAEClose)

define-scripting-event(<count-elements-event>,
                       $kAECoreSuite, $kAECountElements)

define-scripting-event(<create-element-event>,
                       $kAECoreSuite, $kAECreateElement)

define-scripting-event(<delete-event>, $kAECoreSuite, $kAEDelete)

define-scripting-event(<do-objects-exist-event>,
                       $kAECoreSuite, $kAEDoObjectsExist)

define-scripting-event(<get-class-info-event>,
                       $kAECoreSuite, $kAEGetClassInfo)

define-scripting-event(<get-data-event>, $kAECoreSuite, $kAEGetData)

define-scripting-event(<get-data-size-event>,
                       $kAECoreSuite, $kAEGetDataSize)
```

```
define-scripting-event(<get-event-info-event>,
                       $kAECoreSuite, $kAEGetEventInfo)

define-scripting-event(<get-suite-info-event>,
                       $kAECoreSuite, $kAEGetSuiteInfo)

define-scripting-event(<move-event>, $kAECoreSuite, $kAEMove)

define-scripting-event(<save-event>, $kAECoreSuite, $kAESave)

define-scripting-event(<set-data-event>, $kAECoreSuite, $kAESetData)

define-scripting-event(<copy-event>, $kAEMiscStandards, $kAECopy)

define-scripting-event(<cut-event>, $kAEMiscStandards, $kAECut)

define-scripting-event(<make-objects-visible-event>,
                       $kAEMiscStandards, $kAEMakeObjectsVisible)

define-scripting-event(<paste-event>, $kAEMiscStandards, $kAEPaste)

define-scripting-event(<redo-event>, $kAEMiscStandards, $kAERedo)

define-scripting-event(<revert-event>, $kAEMiscStandards, $kAERevert)

define-scripting-event(<select-event>, $kAEMiscStandards, $kAESelect)

define-scripting-event(<undo-event>, $kAEMiscStandards, $kAEUndo)
```

Property Designator Classes

Property classes are designators that identify a property of an Apple event. The Framework provides an abstract `<property>` class from which subclasses can be defined. In addition, the Framework provides subclasses for the following properties:

- document
- file

- text view
- text view selection
- window

The following sections describe the property classes provided by the Framework.

The <property> Class

The `<property>` class represents a property of an Apple event element. You can define subclasses from the `<property>` class and create objects from your subclass when you need to designate a property that is not handled by the Framework. The `class-type` method for this class returns `$cProperty`.

Listing 27-14 shows the definition of the `<property>` class.

Listing 27-14 The `<property>` class

```
define primary open class <property> (<designator>)
  slot property-type :: <integer>,
     setter: #f,
     required-init-keyword: property-type:;
end class;
```

Slot descriptions

`property-type` A property of an Apple event class, such as `$pFont` for the font property of a `$cParagraph` class.

Initialization

No `initialize` method is defined for `<property>` objects.

The <document-property> Class

The `<document-property>` class represents a document as a property of an Apple event element. You do not need to define subclasses from the `<document-property>` class, you can create a `<document-property>` object when you need to designate a document.

Listing 27-15 shows the definition of the <document-property> class.

Listing 27-15 The <document-property> class

```
define primary open class <document-property> (<property>)
  slot parent-type :: <integer>,
     init-value: $cDocument;
end class;
```

Initialization

No initialize method is defined for <document-property> objects.

The <file-property> Class

The <file-property> class represents a file as a property of an Apple event element. You do not need to define subclasses from the <file-property> class, you can create a <file-property> object when you need to designate a file.

Listing 27-16 shows the definition of the <file-property> class.

Listing 27-16 The <file-property> class

```
define primary open class <file-property> (<property>)
end class;
```

Initialization

No initialize method is defined for <file-property> objects.

The <text-view-property> Class

The <text-view-property> class represents a text view as a property of an Apple event element. You do not need to define subclasses from the <text-view-property> class, you can create a <text-view-property> object when you need to designate a text view.

Listing 27-17 shows the definition of the <text-view-property> class.

Listing 27-17 The <text-view-property> class

```
define primary open class <text-view-property> (<property>)
  slot view :: <text-view>,
     required-init-keyword: view:;
end class;
```

Initialization

No initialize method is defined for <text-view-property> objects.

The <text-view-selection-property> Class

The <text-view-selection-property> class represents a selection in a text view as a property of an Apple event element. You do not need to define subclasses from the <text-view-selection-property> class, you can create a <text-view-selection-property> object when you need to designate a selection of text.

Listing 27-18 shows the definition of the <text-view-selection-property> class.

Listing 27-18 The <text-view-selection-property> class

```
define primary open class <text-view-selection-property> (<property>)
  slot view :: <text-view>,
     required-init-keyword: view:;
end class;
```

Initialization

No initialize method is defined for <text-view-selection-property> objects.

The <window-property> Class

The <window-property> class represents a window as a property of an Apple event element. You do not need to define subclasses from the <window-property> class, you can create a <window-property> object when you need to designate a window.

Listing 27-19 shows the definition of the <window-property> class.

Listing 27-19 The `<window-property>` class

```
define primary open class <window-property> (<property>)
  slot parent-type :: <integer>,
     init-value: $cWindow;
end class;
```

Initialization

No `initialize` method is defined for `<window-property>` objects.

Scripting Support Functions

Table 27-3 shows functions that can be used with property objects to locate data identified by object specifier records.

Table 27-3 Object model functions

Function	Purpose	Call	Sp.
`osa-class`	Determine the Open Scripting Architecture class for an object.	√	√
`make-object -specifier`	Locate data within an Apple event object.	√	
`get-element`	Return a token to the specified kind of data.	√	√
`get-property`	Return the specified property.		√
`count-elements`	Specify how to count the number of elements associated with a token.		√
`compare-elements`	Specify how to compare the contents associated with two tokens.		√
`get-container`	Return the container of an object specifier.	√	
`get-property-value`	Return a property's value as an `<ae-desc>` object.	√	√
`set-property-value`	Set a property's value.	√	√

Table 27-4 shows functions that can be used with scripting events.

Table 27-4 Scripting event functions

Function	Purpose	Call	Sp.
`has-reply?`	Determine whether the scripting event has a reply associated with it.	√	
`send-event`	Send an event for dispatching	√	
`get-parameter`	Retrieve a parameter associated with and event.	√	
`set-parameter`	Set a parameter associated with and event.	√	
`set-response`	Set the contents of the response parameter in an Apple event.	√	
`read-long`	Retrieve the contents of a parameter as a long word.	√	
`write-long`	Set the contents of a parameter from a long word.	√	

Table 27-5 shows functions that can be used with Apple event descriptor objects.

Table 27-5 Apple event descriptor functions

Function	Purpose	Call	Sp.
`get-ae-desc`	Create an `<ae-desc>` object and retrieve the data associated with which it was created.	√	
`make-ae-desc`	Create an `<ae-desc>` object.	√	
`set-ae-desc`	Set the data associated with an `<ae-desc>` object.	√	
`as-integer`	Return the contents of an `<ae-desc>` object as an integer.	√	

Table 27-5 Apple event descriptor functions

Function	Purpose	Call	Sp.
`as-boolean`	Return the contents of an `<ae-desc>` object as a boolean.	√	
`as-point`	Return the contents of an `<ae-desc>` object as a point.	√	
`as-rect`	Return the contents of an `<ae-desc>` object as a rectangle.	√	
`as-enumerated`	Return the contents of an `<ae-desc>` object as an enumeration.	√	
`as-file`	Return the contents of an `<ae-desc>` object as a file.	√	
`make-descriptor`	Create an `<ae-desc>` object from a value in a specified data type.	√	√
`make-enumerated -descriptor`	Create an `<ae-desc>` object as an enumeration of a value or values.	√	√
`make-ae-list`	Create an `<ae-desc>` object from a collection.	√	

Table 27-6 shows functions that can be used with Apple event descriptor list objects and Apple event record objects.

Table 27-6 Apple event descriptor list and record functions

Function	Purpose	Call	Sp.
`make-ae-list`	Create an `<ae-desc>` object from a collection.	√	

Table 27-6 Apple event descriptor list and record functions

Function	Purpose	Call	Sp.
`get-key`	Retrieve the key that allows you to locate a descriptor given an `<ae-record>` or `<apple-event>` object.	√	
`set-key`	Set the value of a key that allows you to locate a descriptor given an `<ae-record>` or `<apple-event>` object.	√	
`as-record`	Return the contents of an `<ae-desc>` object as an `<ae-record>` object.	√ß	

Table 27-7 shows functions that can be used with Apple event descriptor list objects and Apple event record objects.

Table 27-7 Apple event object specifier and descriptor functions

Function	Purpose	Call	Sp.
`resolve-specifier`	Internal. Locate the data specified by a descriptor.		
`create-offset -descriptor`	Create an `<ae-desc>` object for an offset position descriptor.	√	
`make-offset -specifier`	Create an `<object-specifier>` object for an offset position descriptor.	√	
`create-comparison -descriptor`	Create an `<ae-desc>` object for a comparison descriptor.	√	
`make-relative -specifier`	Create an `<object-specifier>` object a relative position descriptor.	√	
`create-logical -descriptor`	Create an `<ae-desc>` object for a logical descriptor.	√	

Table 27-7 Apple event object specifier and descriptor functions

Function	Purpose	Call	Sp.
`create-range-descriptor`	Create an `<ae-desc>` object for a range descriptor.	√	
`make-range-specifier`	Create an `<object-specifier>` object for a range descriptor.	√	
`create-object-specifier`	Create an `<object-specifier>` object.	√	

Table 27-6 shows functions that can be used with token objects.

Table 27-8 Apple event descriptor list and record functions

Function	Purpose	Call	Sp.
`make-token`	Create a `<token>` object	√ß	
`token-class`	Determine the Apple event class of a token.	√	
`object-index`	Internal. Get or set the index in an external reference table for a token.		
`tokenized-object`	Retrieve the object associated with a token.	√	
`valid-token?`	Determine if a token is valid.	√	

Scripting Support

CHAPTER 28

Environment and Utilities

Contents

About Environment and Utility Classes 627
- About Region Classes 628
- About String Classes 628
- About Condition Classes 628
- About the Busy Cursor Class 629
- About the Framework Library Class 629
- About External Reference Tables 630

Using Environment and Utility Classes 630
- Using Rectangles 630
- Using String Classes 631
  - Using Handle Strings 631
  - Using Resource Strings 631
- Using Condition Classes 632
  - Using Framework Error Conditions 632
  - Using OS Error Conditions 632
- Using the Busy Cursor Class 633
- Using the Framework Library Class 633
- Using the External Reference Table Class 634

Geometry Objects Reference 635
- The <region> Class 635
- The <rect> Class 635
- Geometry Functions 637

String Objects Reference 639
- The <handle-string> Class 639
- The <resource-string> Class 640
- String Resource Functions 641

Condition Objects Reference 642
The <framework-error> Class 642
The <os-error> Class 643
Error Condition Functions 644
Debugger Behavior Reference 645
The <debugger-behavior> Class 645
Busy Cursor Reference 646
The <busy-cursor> Class 646
Busy Cursor Functions 648
Library Objects Reference 649
The <framework-library> Class 649
Framework Library Functions 651
External Reference Table Reference 651
The <external-reference-table> Class 652
External Reference Functions 653
Miscellaneous Function Reference 654

This chapter discusses several kinds of classes that are utilitarian or related to the operating environment. Utility classes augment classes provided by the Dylan language. The kinds of Framework utility classes are as follows:

- geometry-related classes that represent regions and rectangles
- string classes that represent strings stored in handles or resources
- condition classes for reporting application-related or operating system-reported errors

Environment-related classes allow you to determine the state of the environment and control it. The kinds of Framework environment classes are as follows:

- a class that supports cursor animation when the application cannot respond to an event
- a class that represents code or other resources that can be used by the application
- a class that represents a debugger
- a class that maps Framework objects inside the application to resources or events in the operating environment

This chapter also identifies some functions provided by the Framework that fall into the category of miscellaneous utilities.

About Environment and Utility Classes

The following sections provides an introduction to the following kinds of classes:

- regions
- strings
- conditions
- busy cursor
- debugger behavior
- framework library

- external reference table

About Region Classes

The region class allows you to perform geometric operations on an area. The Framework provides an abstract `<region>` class and subclasses for rectangles and QuickDraw regions and rectangles. The `<region>` class that may be used to specialize methods. The use of the general purpose `<rect>` subclass of the region class is discussed in the section "Using Rectangles" on page 630. The use of the QuickDraw region and rectangle subclasses are described on "The QuickDraw Region Class" on page 324.

About String Classes

The Framework augments the Dylan `<string>` class by providing classes that represent strings in handles or resources. The Framework's classes are the `<handle-string>` and `<resource-string>` classes, respectively.

The `<handle-string>` class represents one or more strings in a handle. The location of a string is determined by an offset into the handle. The size of the string is determined by its first byte, which is the byte at the offset into the handle. You do not need to traverse the handle yourself; you can call the `element` function to retrieve the string and the `size` function to determine its size.

The `<resource-string>` class is a subclass of `<handle-string>`. When strings are retrieved, they are loaded from the specified library and resource.

For examples of using handle and resource strings, see "Using String Classes" on page 631. For a description of the `<handle-string>` class, see "The <handle-string> Class" on page 639. For a description of the `<resource-string>` class, see "The <resource-string> Class" on page 640.

About Condition Classes

The Framework provides subclasses of the Dylan `<error>` class. The Framework's `<framework-error>` class maintains slots for the error message strings and for strings (similar to Macintosh `ParamText` fields) associated with

the message, the reason string, operation string, and recovery string. The Framework substitutes the reason string, operation string, and recovery string into the message string when the message is reported.

The Framework provides a subclass of the `<framework-error>` class to handle operating system-related errors. This class, called `<os-error>`, typically is the class that you will use to report error codes returned from calls to the Macintosh toolbox.

For examples that use the Framework's condition classes, see "Using Condition Classes" on page 632. For information about the `<framework-error>` class, see "The <framework-error> Class" on page 642. For information about the `<os-error>` class, see "The <os-error> Class" on page 643.

About the Busy Cursor Class

The Framework provides a `<busy-cursor>` class that animates cursor movement between events. The Framework creates a `<busy-cursor>` object for you from either an `'acur'` resource for a color cursor or a black and white cursor, depending on the operating environment.

The Framework starts the busy cursor after 60 ticks (about one second) unless you specify not to do so. You can control this threshold value as well as other characteristics. For an example of controlling the busy cursor, see "Using the Busy Cursor Class" on page 633. For information about the `<busy-cursor>` class, see "The <busy-cursor> Class" on page 646.

About the Framework Library Class

Objects created from the `<framework-library>` class are used to initialize and reset the application's runtime environment. You must create a `<framework-library>` object for each library that your application uses. For an example of how to create and manipulate a `<framework-library>` object, see "Using the Framework Library Class" on page 633. For information about the `<framework-library>` class, see "The <framework-library> Class" on page 649.

About External Reference Tables

The Framework provides an external reference table to maintain objects that represent items from outside the Framework, such as Macintosh windows or Apple events. The object cannot be deleted (garbage collected) if it is stored in a table while the item it represents is in use. You can use the Framework's external reference if you have objects that you do not want garbage collected, or you can create your own reference table. For more information, see "Using the External Reference Table Class" on page 634 and "The <external-reference-table> Class" on page 652.

Using Environment and Utility Classes

The following sections show how to use the various environment-related and utility classes.

Using Rectangles

The `<rect>` class represents a rectangle with four slots: `top`, `bottom`, `left`, and `right`. You can create a rectangle as follows:

```
let my-rect = make(<rect>)
```

The function call creates an empty rectangle and is equivalent to the following function calls:

```
let my-rect = make(<rect>, top: 0, bottom: 0, left: 0, right: 0);
let my-rect = rect(0, 0, 0, 0);
```

There are many functions that operate on rectangles and convert between `<rect>` objects and QuickDraw rectangle objects. For a list of functions associated with rectangles, see the tables "Point functions" on page 637 and "Rectangle functions" on page 637. For information about QuickDraw rectangle objects, see "The QuickDraw Region Class" on page 324.

Using String Classes

The Framework provides string classes that represent strings that are stored in handles and resources. The following sections describes how to use these classes.

Using Handle Strings

A `<handle-string>` object specifies the location and size of a string within a handle. These objects are used by the Framework to identify strings within a stream. Listing 28-1 shows the implementation of `read-handle-string` which can be used to retrieve a string from a handle, such as the following error message string:

```
let reason-string :: <string> = read-handle-string(stream);
```

Listing 28-1 Using a handle string

```
define method read-handle-string (stream :: <handle-stream>)
  => result :: <handle-string>;
  let handle-string = make(<handle-string>,
                       handle: stream.stream-handle,
                       offset: stream.stream-position);

  let length = size(handle-string);
  stream.stream-position := stream.stream-position + length + 1;

  handle-string;
end method;
```

Using Resource Strings

You use resource strings when you call the `get-string` or `get-indexed-string` macros. The macros call `get-string-from-library` and `get-indexed-string-from-library`, respectively, which you can use if you want to specify the library. If you use these macros and functions, it is not necessary to explicitly create a `<resource-string>` object. The string is not actually loaded until you access an element of the string or you explicitly call `load-string`.

Using Condition Classes

The following sections describe the condition classes that you can use to handle errors during execution of the application.

Using Framework Error Conditions

You can create `<framework-error>` objects or define a subclass of `<framework-error>` and create objects from it when you want to handle an exception.

You can specialize the `report-error` function to perform actions when an error occurs. The `report-error` function for the main handler displays an error message in a modal dialog.

Using OS Error Conditions

The easiest way to detect an operating system-reported error is to call `fail-os-error` with the operating system return code as its argument. Consider the following example:

```
fail-os-error(FSWrite(file.data-ref-num, byte-count, buffer));
```

The `fail-os-error` function creates an `<os-error>` object and calls the Dylan `error` function unless `FSWrite` returns `$noErr`.

Note
You can specify the keyword parameter `dont-break` to be true (`dont-break: #t`) if you want to debug the application before calling error. The call to `error` occurs, however, as soon as execution resumes. ◆

The Framework provides a variety of functions for specialized errors, such as `fail-mem-error` to detect memory errors. For a list of these functions, see Table 28-1 on page 637.

Using the Busy Cursor Class

The Framework creates a `<busy-cursor>` object for you that allows the cursor to be animated during periods of processing that cannot be interrupted.

By default, the 'acur' resource with ID 256 is used if the operating environment supports color QuickDraw; otherwise, resource ID 257 is used. The default threshold is 60 ticks (about one second) before the threshold is invoked and the animation changes every 10 ticks. You can change these values by changing the slots associated with the object assigned to `*busy-cursor*` or you can create a new `<busy-cursor>` object and assign it to the `*busy-cursor*` variable.

If you know that length of an operation will exceed the busy cursor's threshold, you can call `start-busy-cursor` to start animating the cursor immediately.

Because the busy cursor will automatically start its animation after the threshold is exceeded. you can use the `without-busy-cursor` macro to prevent animation. Listing 28-2 shows an example.

Listing 28-2 Using the `without-busy-cursor` macro

```
without-busy-cursor()
StandardGetFile(as(<FileFilterUPP>, 0), type-count, type-list, reply);
end;
```

Using the Framework Library Class

You must create a `<framework-library>` object for each library that you want to include in your application. The Framework itself creates a `<framework-library>` object for the `dylan-framework` library. To create the object, you should use the `define-framework-library` macro in a top-level form, as in the following example:

```
define-framework-library("hello-app");
```

In addition to creating a `<framework-library>` object, the `define-framework-library` macro adds the object to the application's list of libraries.

You can specify how to initialize and reset the library with the `set-library-init-function` and `set-library-reset-function` macros, respectively.

Listing 28-3 shows an example that uses the `set-library-init-function` macro to initialize the `hello-app` library with the `init-hello` function.

Listing 28-3 Initializing a library

```
define method init-hello () => ()
    install-menu(get-resource-menu($file-menu-id));
    install-menu(get-resource-menu($edit-menu-id));
    add-behavior (*main-handler*, make(<hello-behavior>));
end method;

set-library-init-function(init-hello)
```

Several Framework functions operate implicitly on framework libraries:

- `start` initializes libraries and initiates the main event loop
- `reset` closes all libraries
- `restart` restarts the application without closing libraries

These functions are typically invoked from the Listener.

Using the External Reference Table Class

The Framework provides an external reference table that you can use to maintain references between Framework objects and items external to the Framework. The object cannot be garbage-collected while it is in the table, thus the object persists until it is removed.

For example, the Framework calls `make-extenal-reference` when a scroll bar is created and `dispose-external-reference` to remove it from the table when the window that contains the scroll bar is closed. The `make-extenal-reference` returns an index to the object's position in the table, which must be kept in order to remove the object later. A convenient place to keep the index is the `refCon` field of a Macintosh record that represents the item.

Geometry Objects Reference

The following sections discuss these classes:

- `<region>`
- `<rect>`

The <region> Class

The `<region>` class is an abstract class that represents an area. You do not need to define subclasses of the `<region>` class unless you are implementing a graphics system that the Framework does not directly support, such as QuickDraw GX. You do not need to create objects from the class directly. You can specialize methods on the `<region>` class if you wish.

The Framework provides two subclasses of the `<region>` class for you to use. The `<qd-region>` class represents a Macintosh QuickDraw region. It is described in the section "The QuickDraw Region Class" on page 324. The `<rect>` class represents a rectangle. It is described in the following section.

Listing 28-4 shows the class definition for the `<region>` class.

Listing 28-4 The `<region>` class definition

```
define primary sealed abstract class <region> (<object>)
end class;
```

Initialization

No `initialize` method is defined for `<region>` objects.

The <rect> Class

The `<rect>` class represents an area defined by a rectangle. You do not need to define subclasses of the `<rect>` class. You can create `<rect>` objects as needed. The Framework provides the `rect` method that you can use to create rectangles.

Listing 28-5 shows the class definition for the <rect> class.

Listing 28-5 The <rect> class definition

```
define primary sealed class <rect> (<region>)
  slot top :: <integer>,
     init-value: 0,
     init-keyword: top:;

  slot left :: <integer>,
     init-value: 0,
     init-keyword: left:;

  slot bottom :: <integer>,
     init-value: 0,
     init-keyword: bottom:;

  slot right :: <integer>,
     init-value: 0,
     init-keyword: right:;
end class;
```

Slot descriptions

top	The coordinate that specifies the top of the rectangle.
left	The coordinate that specifies the left-most side of the rectangle.
bottom	The coordinate that specifies the bottom of the rectangle.
right	The coordinate that specifies the right-most side of the rectangle.

Initialization

No initialize method is defined for <rect> objects.

Geometry Functions

Table 28-1 shows functions that you can use with points.

Table 28-1 Point functions

Function	Purpose	Call	Sp.
`point`	Create a `<point>` object.	√	
`get-point`	Converts a QuickDraw point or an integer to a `<point>` object.	√	
`set-point`	Sets a QuickDraw point to the coordinates specified by a `<point>` object.	√	
`as-point`	Converts an object into a `<point>` object.	√	√

Table 28-2 shows functions that you can use with `<rect>` objects.

Table 28-2 Rectangle functions

Function	Purpose	Call	Sp.
`rect`	Create a `<rect>` object.	√	
`get-rect`	Converts a QuickDraw rectangle to a `<rect>` object.	√	
`set-rect`	Sets a QuickDraw rectangle to the coordinates specified by a `<rect>` object.	√	
`width`	Determines the width of a `<rect>` object.	√	
`height`	Determines the height of a `<rect>` object.	√	
`top-left`	Returns the point a the top-left corner of a `<rect>` object.	√	
`bottom-right`	Returns the point a the bottom-right corner of a `<rect>` object.	√	

Table 28-2 Rectangle functions

Function	Purpose	Call	Sp.
`set-rect-from-points`	Set the top-left and bottom-right points of a rectangle.	√	
`as-rect`	Converts an object into a `<rect>` object	√	√
`frame-region`	Draw a border just inside a rectangle's bounds.	√	
`fill-region`	Fill a rectangle with the specified pattern.	√	

Table 28-3 shows functions that you can use with `<rect>` objects or `<region>` objects.

Table 28-3 Rectangle or region functions

Function	Purpose	Call	Sp.
`empty-region?`	Determines if the region is empty; a rectangle is empty if it is actually a line.	√	
`set-empty-region`	Set a region to be empty.	√	
`set-rect-region`	Change the structure of a region as specified by a rectangle.	√	
`inset-region`	Shrink or expand a region.	√	
`offset-region`	Move a region.	√	
`point-in-region?`	Determine whether a point is in a region.	√	
`intersect-region`	Determine the intersection of two regions.	√	
`regions-intersect?`	Determine whether two regions intersect.	√	
`difference-region`	Subtract one region from another.	√	

Table 28-3 Rectangle or region functions

Function	Purpose	Call	Sp.
`xor-region`	Calculate the difference between the union and the intersection of two regions.	√	
`union-region`	Create a region from two regions.	√	
`add-to-region`	Expand a region.	√	
`subtract-from-region`	Shrink a region.	√	
`intersect-with-region`	Create a region that is the smaller of two other regions.	√	

String Objects Reference

The following sections discuss these classes:

- `<handle-string>`
- `<resource-string>`

The <handle-string> Class

The `<handle-string>` class represents a string that is stored in a handle. You do not need to define a subclass from the `<handle-string>` class. You can create `<handle-string>` objects directly. Listing 28-6 shows the definition of the `<handle-string>` class.

Listing 28-6 The `<handle-string>` class definition

```
define primary sealed class <handle-string> (<string>, <vector>)
  slot string-handle :: <machine-pointer>,
     init-value: $null-machine-pointer,
     init-keyword: handle:;
```

```
  slot string-offset :: <integer>,
     init-value: 0,
     init-keyword: offset:;
end class;
```

Slot descriptions

`string-handle`	The handle that contains the characters.
`string-offset`	The start of the string in the handle.

Initialization

No `initialize` method is defined for `<handle-string>` objects.

The <resource-string> Class

The `<resource-string>` class represents a string that is stored in a resource. The resource is accessed via a Framework library. You do not need to define a subclass from the `<resource-string>` class. You can create `<resource-string>` objects directly. Listing 28-7 shows the definition of the `<resource-string>` class.

Listing 28-7 The `<resource-string>` class definition

```
define primary sealed class <resource-string> (<handle-string>)

  slot string-library :: <framework-library>,
     required-init-keyword: library:;

  slot string-id :: <integer>,
     required-init-keyword: id:;

  slot string-index :: <integer>,
     init-value: 0,
     init-keyword: index:;

end class;
```

Slot descriptions

`string-library`	The library that contains the string resource.
`string-id`	The resource ID.
`string-index`	The index within the ID.

Initialization

No `initialize` method is defined for `<resource-string>` objects.

String Resource Functions

Table 28-4 shows functions that you can use with string resource objects.

Table 28-4 String functions

Function	Purpose	Call	Sp.
`load-string`	Load a string from a resource.	√	
`size`	Determine the size of a string.	√	
`get-string`	Create a `<resource-string>` object from the specified string in the `$library` library. Macro.	√	
`get-indexed-string`	Create a `<resource-string>` object from the specified string at the specified index in the `$library` library. Macro.	√	
`get-string-offset`	Determine the location in a string in a resource as specified by its index.	√	
`element`	Retrieve a character in a string at the specified position.	√	
`get-string-from-library`	Create a `<resource-string>` object from the specified string in the specified library.	√	
`get-indexed-string-from-library`	Create a `<resource-string>` object from the specified string at the specified index in the specified library.	√	

Condition Objects Reference

Conditions are classes which are built into the Dylan language. The Framework extends these conditions by providing subclasses of the `<error>` class that represent application-level and operating system-generated error conditions. The following sections describe these classes:

- `<framework-error>`
- `<os-error>`

The <framework-error> Class

The `<framework-error>` class represents an application-level error. The Framework defines a subclass for operating system-generated errors. (See the following section for information about this subclass.) You may define a subclass of the `<framework-error>` class or create `<framework-error>` objects directly. Listing 28-8 shows the definition for the `<framework-error>` class.

Listing 28-8 The `<framework-error>` class definition

```
define primary open class <framework-error> (<error>)
  slot message-string :: <string>,
     init-function: default-message-string;

  slot reason-string :: <string>,
     init-function: default-reason-string;

  slot operation-string :: <string>,
     init-function: default-operation-string;

  slot recovery-string :: <string>,
     init-function: default-recovery-string;
end class;
```

Slot descriptions

`message-string`	The message, such as “Could not ... because ...”.
`reason-string`	The reason (because clause).
`operation-string`	The action, for example “complete your request”.
`recovery-string`	Suggestions for recovery.

Initialization

No `initialize` method is defined for `<framework-error>` objects.

The <os-error> Class

The `<os-error>` subclass represents an error condition reported by the Macintosh Operating System. You do not need to define a subclass of `<os-error>`; you create `<os-error>` objects as they are needed. Listing 28-9 shows the definition for the `<os-error>` class.

Listing 28-9 The `<os-error>` class definition

```
define primary sealed class <os-error> (<framework-error>)
  slot error-number :: <integer>,
     required-init-keyword: error-number:;
end class;
```

Slot descriptions

`error-number`	The Macintosh `OSErr` number.

Initialization

The `initialize` method for `<os-error>` objects sets the reason and recovery strings to correspond to the error number. The `initialize` method’s parameters are defined as follows:

```
define method initialize (error :: <os-error>, #key) => ()
```

`error`	The error object.

Error Condition Functions

Table 28-5 shows functions that you can use with error condition objects.

Table 28-5 Error condition functions

Function	Purpose	Call	Sp.
`default-message-string`	Return the default message string.	√	
`default-operation-string`	Return the default operation string.	√	
`default-recovery-string`	Return the default recovery string.	√	
`report-error`	Display the error message.	√	√
`error-dialog`	Internal. Create an error message dialog box.		
`make-error-string`	Create a string that contains the specified error message.	√	

Table 28-6 shows functions that you can use to manipulate error strings.

Table 28-6 Error string functions

Function	Purpose	Call	Sp.
`replace-string-parameters`	Replaces parameters in param text.	√	
`read-error-list-resource`	Internal.		
`read-error-lists`	Internal.		

Table 28-7 shows functions that you can use with <os-error> objects.

Table 28-7 Functions for <os-error> objects

Function	Purpose	Call	Sp.
make-os-error	Create an <os-error> object.	√	√
fail-os-error	Signal that a nonrecoverable operating system error occurred.	√	
fail-nil	Signal that a nonrecoverable attempt to manipulate a pointer occurred.	√	
fail-unless	Signal that a nonrecoverable attempt was made to manipulate the wrong object.	√	
fail-res-error	Signal that a nonrecoverable resource error occurred.	√	
fail-nil-resource	Signal that a nonrecoverable attempt to manipulate a resource occurred.	√	
fail-mem-error	Signal that a nonrecoverable memory error occurred	√	

Debugger Behavior Reference

The Framework provides an application-level debugger. It is implemented as a behavior, as described in the following section.

The <debugger-behavior> Class

The <debugger-behavior> class represents a prototype application-level debugger that allows you to see the target chain. The Framework creates a <debugger-behavior> object for you and installs a menu to which the behavior responds. If you do not want the debugger menu to appear, set the

install-debug-menu variable to false (#f). Listing 28-10 shows the <debugger-behavior> class definition.

Listing 28-10 The <debugger-behavior> class definition

```
define primary sealed class <debugger-behavior> (<behavior>)
end class;
```

Initialization

There is no initialize method for <debugger-behavior> objects.

Busy Cursor Reference

The Framework allows you to implement a busy cursor with a class that provides cursor animation. The cursor can be either color or black and white. The busy cursor is associated with an 'acur' resource. The following section describes the class that implements busy-cursor support.

The <busy-cursor> Class

The <busy-cursor> class represents an animated (moving) cursor that can be displayed while the application is doing non-interruptible work. You do not need to define a subclass of <busy-cursor>, nor do you need to create a <busy-cursor> object. The Framework creates a <busy-cursor> object for you using the 'acur' resource. If the operating environment supports color QuickDraw, resource ID 256 is used to create the object; otherwise, resource ID 257 is used. If you want to use a different <busy-cursor> object, you can create it and assign it to *busy-cursor*.

Listing 28-11 The <busy-cursor> class definition

```
define primary sealed class <busy-cursor> (<object>)
  slot busy? :: <boolean>,
     init-value: #f;
```

```
  slot use-color? :: <boolean>,
     init-value: #f,
     init-keyword: use-color:;

  slot disable-count :: <integer>,
     init-value: 1;

  slot cursor-count :: <integer>;

  slot cursors :: <vector>;

  // ticks without clear until we get busy
  slot busy-threashold :: <integer>,
     init-value: 60,
     init-keyword: busy-threashold:;

  // ticks since the last time we were cleared
  slot last-clear :: <integer>,
     init-value: #xffffffff;

  // ticks since the last time we animated the cursor
  slot last-animate :: <integer>,
     init-value: 0;

  // ticks since the last time we animated the cursor
  slot animate-frequency :: <integer>,
     init-value: 10,
     init-keyword: animate-frequency:;

  // current cursor in the animation sequence
  slot current-cursor-index :: <integer>,
     init-value: 0;
end class;
```

Slot descriptions

`busy?`	Whether the busy cursor is active (`#t`), or not (`#f`).
`use-color?`	Whether the busy cursor is a color cursor (`#t`), or not (`#f`).
`disable-count`	The number of times the busy cursor has been disabled.

`cursor-count` The number of cursors in the resource, which provides the unique cursor appearances when the cursor is animated.

`busy-threashold` The number of ticks without `clear-busy-cursor` being called before the busy cursor is activated.

`last-clear` The tick count when clear-busy-cursor was last called.

`last-antimate` The tick count when the busy cursor changed its appearance.

`antimate-frequency` The number of ticks desired between changes of appearance.

`current-cursor-index` The index that specifies the cursor's current appearance.

Initialization

The `initialize` method for `<busy-cursor>` objects sets up the unique cursor appearances that display during animation. The `initialize` method's parameters are defined as follows:

```
define method initialize (busy-cursor :: <busy-cursor>,
                          #key acur-resource)
```

`busy-cursor` The busy cursor object.

`acur-resource` The resource that contains the cursor appearances.

Busy Cursor Functions

Table 28-8 shows functions that you can use with `<busy-cursor>` objects.

Table 28-8 Busy cursor functions

Function	Purpose	Call	Sp.
`disable-busy-cursor`	Turn off animation of the cursor.	√	
`reenable-busy-cursor`	Restart cursor animation.	√	
`start-busy-cursor`	Start animation of the cursor.	√	

Table 28-8 Busy cursor functions

Function	Purpose	Call	Sp.
`clear-busy-cursor`	Reset the interval before animation starts.	√	
`poll-busy-cursor`	Internal.		
`set-busy-cursor`	Internal. Set up the busy cursor.		
`init-busy-cursor`	Initialize the busy cursor.	√	
`without-busy-cursor`	Specify the actions to take while animation is off. Macro.	√	

Library Objects Reference

The Framework allows you to include multiple libraries in your application. Each library is represented by a `<library-framework>` object. The following section describes this class.

The <framework-library> Class

The `<framework-library>` class represents a library of compiled code and resources. You do not need to define a subclass of the `<framework-library>` class, rather you create a `<framework-library>` object for each library.

Listing 28-12 The `<framework-library>` class definition

```
define primary sealed class <framework-library> (<object>)
  slot library-name :: <symbol>,
     required-init-keyword: name:;

  slot init-function,
     init-value: #f,
     init-keyword: init-function:;
```

```
  slot reset-function,
     init-value: #f,
     init-keyword: reset-function:;

  slot library-resource-files :: <list>,
     init-value: #();

  slot library-initialized? :: <boolean>,
     init-value: #f;
end class;
```

Slot descriptions

`library-name` A library associated with the application.

`init-function` The function the specifies how to set up the library.

`reset-function` The reset function associated with the library.

`library-resource-files` The resource files associated with the application.

`library-initialized` True (`#t`) if the library is initialized; otherwise false (`#f`).

Initialization

The `initialize` method for `<framework-library>` objects adds the library to the Framework's list of libraries. The `initialize` method's parameters are defined as follows:

```
define method initialize (lib :: <framework-library>, #key) => ()
```

`lib` The library.

Framework Library Functions

Table 28-9 shows functions that you can use to initialize, start, reset, and restart your application. These functions are not specialized on a `<library-framework>` object because they operate on all libraries associated with the application.

Table 28-9 Framework library functions

Function	Purpose	Call	Sp.
`reset-framework`	Internal. Reset Framework and environment.		
`init-framework`	Internal. Set up global variables.		
`initialize-framework`	Internal. Set up libraries.		
`reset`	Reset the Framework so that calling Start will reset libraries and global variables.	√	
`start`	Start the application.	√	
`run`	Internal. Run the application.		
`restart`	Redraw all windows and start the application.	√	
`define-framework-library`	Define the Framework's library. Macro.	√	
`set-library-init-function`	Specify the function that initializes the application's libraries. Macro.	√	
`set-library-reset-function`	Specify the function that resets the application's libraries. Macro.	√	

External Reference Table Reference

The following section describes the external reference table class.

The <external-reference-table> Class

The `<external-reference-table>` class represents a table of associations between Dylan objects within the application and external items, such as Macintosh windows. You do not need to define a subclass of the `<external-reference-table>` class.

The Framework provides a single `<external-reference-table>` object and methods to access it. The Framework handles references between objects defined it its classes and external items for you. You can use the access methods to add or remove your own items, or you can create your own table and manage its contents using general-purpose methods.

Listing 28-13 The `<external-reference-table>` class definition

```
define primary sealed class <external-reference-table> (<object>)
  slot object-array :: <stretchy-vector>,
     setter: #f,
     init-function: curry(make, <stretchy-vector>, size: 100);

  slot free-list :: <integer>,
     init-value: 0;
end class;
```

Slot descriptions

`object-array` Internal.

`free-list` Internal.

Initialization

The `initialize` method for `<external-reference-table>` objects. The `initialize` method's parameters are defined as follows:

```
define method initialize (table :: <external-reference-table>,
                          #key) => ()
```

`table` The external reference table.

External Reference Functions

Table 28-10 shows functions that you can use with external reference tables.

Table 28-10 External reference functions

Function	Purpose	Call	Sp.
`add-object-to-table`	Internal. Insert an object into the table.		
`remove-object-from-table`	Internal. Remove an object from the table.		
`get-object-from-table`	Internal. Retrieve an object from the table.		
`make-external-reference`	Create a reference and store it in the Framework's external reference table.	√	
`dispose-external-reference`	Dispose of a reference in the Framework's external reference table.	√	
`resolve-external-reference`	Retrieve the object specified by a reference it in the Framework's external reference table.	√	
`init-external-reference`	Internal. Create the Framework's external reference table.		
`clear-external-reference`	Internal. Dispose of the Framework's external reference table.		

Miscellaneous Function Reference

Table 28-11 shows functions that you can use to initialize the Framework.

Table 28-11 Environment functions

Function	Purpose	Call	Sp.
`init-environment`	Internal. Set up the Macintosh environment.		
`application-in-front?`	Determine if the application is the frontmost process.	√	
`bring-application-to-front`	Bring the application to front.	√	

Table 28-12 shows miscellaneous utility functions.

Table 28-12 Miscellaneous utility functions

Function	Purpose	Call	Sp.
`assert`	Display a message and break the event loop if the object is not the specified one.	√	
`insert`	Insert an object into a list at the specified position	√	
`first-that`	Retrieve the first object in a collection that matches the specified criteria.	√	
`copy-handle`	Copy the contents of a handle into a new handle.	√	
`beep`	Execute the Macintosh SysBeep function.	√	

Table 28-12 Miscellaneous utility functions

Function	Purpose	Call	Sp.
`string-to-handle`	Copy a string of bytes to a handle that contains the string.	√	
`handle-to-string`	Copy the contents of a handle to a byte string.	√	
`key-down`	Determine if the specified key was pressed during the last event.	√	

Table 28-13 shows miscellaneous goodie functions.

Table 28-13 Miscellaneous goodie functions

Function	Purpose	Call	Sp.
`choose-document`	Allow the user to choose one of the specified documents.	√	
`save-changes-dialog`	Prompt the user to save changes to a document.	√	
`request-file-name`	Prompt the user for a file.	√	
`yes-no-dialog`	Prompt the user for a Yes or No response.	√	
`ask-dialog`	Prompt the user for a confirmation.	√	
`tell-dialog`	Present information in a modal dialog box.	√	
`make-scrolling -window`	Create a scrolling window.	√	
`make-palette-window`	Create a palette window.	√	
`make-edit-menu`	Create an Edit menu.	√	

CHAPTER 28

Environment and Utilities

Index

Symbols

$default-determiner constant 358
$flavorNotSaved constant 580
$flavorSenderOnly constant 580
$flavorSenderTranslated constant 580
$flavorSystemTranslated constant 580
$null-identifier constant 363
$read-write-permission constant 499
$zero-point constant 311
%columns slot 392
%command-key slot 296
%dimmed? slot 442
%enabled? slot 294
%event slot 295
%floats? slot 410
%hilited? slot 442
%icon slot 296
%justification slot 439
%mark slot 296
%max slot 448
%min slot 448
%modal? slot 410
%on? slot 442
%root-view slot 412
%rows slot 392
%style slot 296
%super-view slot 359
%target-handler slot 266
%title slot 294, 297, 412, 486
%translation slot 472
%value slot 448
%visible? slot 359, 412
<activate-event> class 272
<adorner> class 360
<ae-desc> class 607
<ae-list> class 608
<ae-record> class 609
<apple-event> class 610
<apple-menu> class 300
<background-mouse-down-event> class 273
<become-target-event> class 274
<behavior> class 266
<black-background-adorner> class 360
<busy-cursor> class 646
<button> class 443
<button-event> class 454
<character-designator> class 563
<check-box> class 443
<check-box-event> class 454
<class-spec> class 522
<clear-command> class 596
<clipboard> class 594
<clipboard-behavior> class 594
<clipboard-command> class 596
<clipboard-flavor> class 595
<clone-event> class 614
<close-event> class 614
<cluster> class 437
<command> class 540
<command-context> class 539
<comparison-descriptor> class 611
<control> class 440
<control-event> class 454
<copy-event> class 615
<count-elements-event> class 614
<create-element-event> class 614
<C-string> class 108
<ctl-mgr-button> class 448
<ctl-mgr-check-box> class 449
<ctl-mgr-control> class 446
<ctl-mgr-popup> class 452
<ctl-mgr-radio> class 450
<ctl-mgr-scroll-bar> class 451
<cut-command> class 596
<cut-event> class 615

<data-item> class 560
<debugger-behavior> class 645
<default-button-adorner> class 455
<delete-event> class 614
<designator> class 561
<dialog-behavior> class 433
<disk-event> class 270
<document> class 485
<document-property> class 616
<do-objects-exist-event> class 614
<drag> class 577
<drag-event> class 580
<drag-flavor> class 579
<drag-item> class 578
<draw-adorner> class 360
<edit-text> class 378
<edit-text-dialog-filter> class 436
<event> class 267
<event-handler> class 263
<external-reference-table> class 652
<file> class 496
<file-descriptor> class 611
<file-object-stream> class 521
<file-property> class 617
<file-stream> class 518
<font-command> class 545
<font-size-command> class 545
<font-style-command> class 545
<frame> class 319
<frame-adorner> class 360
<framework-error> class 642
<framework-library> class 649
<generic-mouse-down-event> class 273
<get-class-info-event> class 614
<get-data-event> class 614
<get-data-size-event> class 614
<get-event-info-event> class 615
<get-suite-info-event> class 615
<graphics-port> class 320
<grid-arrow-key-behavior> class 396
<grid-select-behavior> class 396
<grid-view> class 391
<grow-icon> class 414
<grow-icon-determiner> class 414
<handle-object-stream> class 521
<handle-stream> class 519
<handle-string> class 639
<has-super-view-bounds> class 362
<hilite-adorner> class 360
<horiz-scroll-bar-determiner> class 363
<horiz-size-super-view-determiner>
class 362
<index-descriptor> class 611
<key-down-event> class 270
<key-event> class 270
<key-up-event> class 270
<list-view> class 394
<logical-descriptor> class 611
<machine-pointer> class 105
<main-handler> class 265
<make-objects-visible-event> class 615
<menu> class 297
<menu-element> class 292
<menu-event> class 301
<menu-item> class 294
<mouse-down-event> class 273
<mouse-event> class 273
<mouse-moved-event> class 273
<mouse-up-event> class 273
<move-event> class 615
<null-descriptor> class 611
<number-text> class 379
<object-specifier> class 610
<object-stream> class 521
<offset-designator> class 564
<open-application-event> class 614
<open-event> class 614
<os-error> class 643
<Pascal-string> class 110
<paste-command> class 596
<paste-event> class 615
<popup> class 445
<popup-event> class 454
<print-event> class 614
<property> class 616
<qd-graphics-buffer> class 320, 348
<qd-graphics-port> class 320
<qd-region> class 324

<qd-text-style> class 322
<quit-event> class 614
<radio> class 443
<radio-event> class 454
<range-descriptor> class 611
<range-designator> class 564
<receive-drag-event> class 581
<rect> class 635
<redo-event> class 615
<region> class 635
<relative-descriptor> class 611
<resign-target-event> class 275
<resource-string> class 640
<resume-event> class 271
<revert-event> class 615
<rgb-color> class 324
<save-event> class 615
<scripting-event> class 612
<scroll-bar> class 444
<scroll-bar-event> class 454
<scroller> class 471
<select-event> class 615
<selection-designator> class 562
<separator-item> class 297
<set-data-event> class 615
<simple-icon> class 438
<slot-spec> class 523
<statically-typed-pointer> class 106
<static-text> class 439
<stream> class 517
<super-view-horiz-relative-determiner> class 363
<super-view-relative-determiner> class 362
<super-view-vert-relative-determiner> class 363
<suspend-event> class 271
<system-event> class 271
<tabber> class 435
<text-designator> class 563
<text-grid-view> class 394
<text-list-view> class 395
<text-style> class 322
<text-style-command> class 544
<text-view> class 375
<text-view-property> class 617
<text-view-selection-property> class 618
<token> class 612
<toolbox-event> class 268
<track-drag-event> class 581
<tracker> class 473
<tracker-adorner> class 474
<typed-data> class 558
<typed-handle> class 560
<typed-pointer> class 559
<typing-command> class 542
<undo-event> class 615
<update-event> class 272
<vert-scroll-bar-determiner> class 363
<vert-size-super-view-determiner> class 363
<view> class 345, 355
<view-determiner> class 353, 362
<window> class 407
<window-context> class 406
<window-event> class 271
<window-property> class 618

A

about-item slot 300
About menu item 292
accept-drag? method 584
access paths 121
action-proc slot 447
activate-it? slot 272
activate method 358, 366, 456
active? method 358
active? slot 358, 410
active-hilite slot 358
add-adorner method 357, 368
add-behavior method 264, 278
add-character method 548
add-child method 547
add-graphics-cache method 365
add-menu-item-from-toolbox-menu method 303
add-menu-item method 294, 298, 303

add-object-to-table method 653
add-resource-to-file method 496, 502
add-sub-view method 357, 364, 475
add-to-region method 329, 639
add-view-to-port method 367
add-window method 415
adjust-extent method 366, 381
adjust-te-rects method 381
adorner-draw method 351, 368
adorner-hilite method 351, 368
adorner-list slot 357
align-word method 514, 527
allocation-size slot 520
all-slots slot 523
always-enabled? slot 299
antimate-frequency slot 648
append-text method 381
Apple events 602
apple-menu variable 292, 300
Apple Shared Library Manager Access Path 129
argument-type options 34
arrays as structure members 97
arrow (=>) options 59
as-boolean method 621
as-enumerated method 621
as-file method 621
as-integer method 620
ASLM shared libraries 131
as-point method 621, 637
as-qd-region method 328
as-record method 622
as-rect method 328, 621, 638
automatic name mapping 55
automatic termination 208, 211
auto-scroll? slot 473
available-types method 567

B

back-color slot 358
behavior-accept-drag? method 575, 584
behavior-close method 278
behavior-event method 261, 278
behavior-idle method 278
behavior-list slot 264
behavior-open method 278
behavior-receive-drop method 576, 584
behaviors 254
  adding and removing 262
  clipboard 589
  implementing 260
behavior-set-cursor method 262, 278, 365
behavior-setup-menus method 278, 289, 303
behavior-update method 278
blue slot 325
bold? method 330
bottom-right method 637
bottom slot 636
break-request-key-chord variable 183
break-request variable 183
buffer-position slot 519
buffer-size slot 518
buffer slot 518
busy? slot 647
busy cursor 629, 633
busy-cursor variable 646
busy-threashold slot 648
bytes-in-buffer slot 518
bytes-per-chunk-expansion variable 201
bytes-per-chunk variable 201
bytes-per-LOS-cluster-expansion
    variable 201
bytes-per-LOS-cluster variable 201
by-value types 91

C

C 12
cached-local-bounds slot 359
callback clauses 22, 38, 41, 51
callback functions 72, 79
callback macros 72, 80
callout clauses 22, 38, 41
callout functions 72, 78, 140

cancel-item slot 434
can-receive-data? method 382, 567, 589, 598
causes-change? slot 541
C constants 14
C containers 12, 16
C definitions 12
cell-bounds method 397
cell-in-rect? method 398
cell-selected? method 398
CFM-library-oldest-version option 123
CFM-library option 123
CFM-library options 21
CFM-library-version option 123
CFM shared libraries 126
C functions 14
change counts 480
change-count slot 486
character-at method 156
character-at-setter method 157
C header files 12, 16, 23
children slot 539
choose-document method 500
class specifications 507
  default objects 512
  setting up 508
  shared objects 512
class-streaming-version method 528
class-symbol slot 523
class-to-symbol method 528
class-type method 568
clauses 11, 18, 39
clear-buffer method 328
clear-busy-cursor method 649
clear-external-reference method 653
click-loop method 381
clipboard behavior 589
clipboard support 587
  and views 591
  deleting the selection 593
  designating selected data 590
  retrieving data 591
  specifying paste data 589
clip-change-count slot 595
clip-region slot 360
clip-to-super-view method 366
clobber-oldspace variable 202
clonable? method 279
cloneable-behaviors method 279
clone-behavior-list method 279
clone method 328, 511, 527
clone-slot? slot 525
cloning-getter slot 524
cloning objects 511
closable? slot 410
close-data-fork method 500
close-files method 488, 502
close method 276, 327, 364, 415, 487, 500, 547
close-resource-fork method 500
close-with-last-window? slot 265, 407
Code Fragment Manager Access Path 126
code resources 135
color 316
color-table slot 322
columns method 392
columns-setter method 392
columns slot 395
column-to-offset method 397
column-width method 397
command-key? slot 269
command-key method 296
command-key-setter method 296
command-queue slot 265
commands 531
  and documents 480
  committing 536
  completing 537
  defining 534
  doing 535
  performing 537
  undoing and redoing 535
commit method 536, 548
commit-on-save? slot 486
compacting algorithm 195
compare-elements method 619
completed method 537, 548
compute-clip method 367
compute-visible-bounds method 357, 366
conceptual model 219

condense? method 330
condition classes 628
constant clauses 22, 37, 41
constant options 22, 37, 46
container clauses 19, 56
container options 20, 43
containers 16
container slot 562
content view 337
context slot 541
control-char slot 437
control events 423
control-handle slot 447
control-key? slot 269
control-max method 448, 456
control-min method 448, 456
control-mode slot 441
controls 419
  highlighting 424
control slot 454
control-value method 448, 456
control view classes 421
convert-clipboard? slot 271
coordinate systems 313
copy-buffer method 328
copying algorithm 195
count-elements method 619
count-resources-in-file method 502
create-comparison-descriptor method 622
create-logical-descriptor method 622
create-menu-event method 279
create method 500
create-object-specifier method 623
create-offset-descriptor method 622
create-range-descriptor method 623
cross-language calls 71
c-string-at method 168
c-string-at-setter method 169
C structure members 15
C structures 16
C union members 15
C unions 16
current-choice-id method 457
current-choice-item method 457
current-choice-string method 457
current-cursor-index slot 648
current-hilite method 358, 365
current-view slot 577
cursor-count slot 648
cursor region 344
cursors 344
  busy cursor 629, 633
  implementing as a behavior 262
C variables 15

D

data flavors 551
data-fork-open? method 498, 500
data-handle slot 560
data interchange 551
data item classes 552
data items 555
data-item slot 597
data-length method 566, 582
data-length slot 559, 595
data-permission slot 499
data-pointer slot 559
data-ref-num slot 498
data-type slot 559
default-<text-grid-view>-text-getter method 394
default-<text-list-view>-text-getter method 395
default-column-width slot 393
default importing behavior 11
default-instance method 512
default-item slot 434
default-message-string method 644
default name mapping 55
default-operation-string method 644
default-recovery-string method 644
default-row-height slot 393
define-framework-library macro 633
define-framework-library method 651
define options 21, 24

define-scripting-event macro 613
delete-all-resources-in-file method 496, 502
delete-data method 381, 567, 593, 598
delete method 500
delete-resource-from-file method 502
delete-selection method 381, 567
depth slot 322
descriptor classes 611
deselect-cell method 398
designators 551
  classes for 553
  creating 557
  defining 557
  property classes 615
determiner-list slot 358
dialog behaviors 419
dialog box, creating 425
dialogs 419
  dismissers 429
  windows for 429
did-command method 547
difference-region method 329, 638
dimmed? method 442
dimmed?-setter method 442
Direct Pointer Access Path 135
dir-id slot 498
disable-buffering method 367
disable-busy-cursor method 648
disable-count slot 647
dismisser slot 434
dismisses-dialog? slot 441
dispatch-event method 276
dispose-external-reference method 653
dispose method 327, 582
dispose-on-close? slot 411
do-content-click method 277
documents 479
  and commands 480
  creating 482
  defining subclasses 481
  files for 492
  file types 484
  menu items for 480
  opening 479, 482
  registering 482
  windows for 483
documents and data interchange classes 236
document-type-list slot 265
do-drag method 277
do-each-sub-view method 364
do-enabling method 290, 303
do-event method 260, 276, 487
do-go-away method 277
do-initial-state method 487
do-invalidation method 368
doit method 535, 548
do-marking method 290, 303
done? slot 541
done-drawing method 366
done-drawing-region method 366
done-typing method 382, 548
do-scroll-action method 457
do-scroll method 457
do-set-cursor method 365
do-setup-menus method 276, 299, 303, 487
double-click? method 277
do-update method 277
do-validation method 368
do-view-and-subviews method 364
do-zoom method 277
drag and drop support 571
  accepting a drag 575
  promise functions 574
  receiving a drop 576
  setting up a drag 572
drag-enter-view method 583
drag-exit-view method 583
draggable? method 415
drag-hilite method 583
drag-item slot 580
drag-mouse-moved? method 583
drag-origin method 583
drag-reference method 582
drag-reference slot 577
drag slot 579, 581
draw-adorners method 365
draw-buffer method 327, 350

draw-cell-border? slot 393
draw-cell-border method 398
draw-cell method 398
draw-feedback-rect method 475
drawing in views 342, 348
draw-item method 398
draw method 348, 365, 456
dynamic-menu? slot 445

E

eldest generation 194
element method 641
empty-region? method 328, 638
enable-automatic-termination variable 211
enable-buffering method 367
enable-compacting-garbage-collection variable 201
enabled? method 294
enabled?-setter method 294
enabled? slot 264
enable-garbage-collection variable 201
enable-item method 290, 303
end-of-stream? method 525
ephemeral garbage collection 194
erase-region method 329
error conditions 632
error-dialog method 644
error-number slot 643
event checking 151
event handlers 246
  adding to target chain 259
  performing an event 260
event handling 242
event handling classes 223
event-loop method 279
event method 295
event-not-handled method 277
events
  control 423
  kinds of 242
  menu 284, 290
  responding to 250
event-setter method 295
event-time slot 269
event-what slot 269
event-window-ptr slot 272
exclude options 20, 26
explicit name mapping 56
explicit type mapping 89, 93, 100
exporting values 101
export-temporary-value function 102
export-value function 102
extend? method 330
extent 312, 317
extent-changed method 366
extent slot 320
External Module Access Path 133
external-module file option 123
external-module options 21
external modules 133
external reference table 630, 634

F

fail-mem-error method 645
fail-nil method 645
fail-nil-resource method 645
fail-os-error method 632, 645
fail-res-error method 645
fail-unless method 645
file clauses 19, 23, 40
file-creator slot 499
file-name slot 497
file options 21, 42
files 491
  and documents 492
  reading from 493
  reading from resource fork 495
  types of 484
  updating resource forks 495
  writing to 494
file slot 518
file-type slot 499

fill-region method 329, 638
find-adorner method 368
find-behavior method 278
find-menu-item method 303
find-sub-view method 357, 364
first-click? slot 411
first-target slot 435
flavor-available? method 567
flavor-flags slot 580
flavor-index slot 580
flavors of data 551
flavors slot 561
flavor-types method 583
floats? method 411
floats?-setter method 411
flush-slot-cache method 528
flush-stream method 526
font-number slot 323
fonts, getting and setting styles 374
font-size slot 323
font slot 323
font-style slot 323
fore-color slot 358
foundation classes 220
found-current-target slot 436
frame-region method 329, 638
frames 310
framework library 629, 633
free-list slot 652
function clauses 22, 34, 41
function options 22, 34, 49
function pointers 77

G

garbage 193
garbage collection 193
generations 194
get-ae-desc method 620
get-class-spec method 527
get-clip-region method 367
get-command-context method 415, 538, 547
get-container method 619
get-data-handle method 566, 598
get-data method 382, 567, 591, 598
get-default-cursor-region method 365
get-document method 487
get-drag-hilite-region method 583
get-element method 619
get-end-of-file method 493, 501
get-file-position method 501
get-flavor method 567
get-graphics-cache method 365
get-indexed-string-from-library method 641
get-indexed-string method 641
get-item-bounds method 582
get-item method 583
get-ith-resource-from-file method 495, 501
get-key method 622
get-menu method 302
get-mouse method 366
get-named-resource-from-file method 502
get-object-from-table method 653
get-parameter method 620
get-point method 637
get-property method 605, 619
get-property-value method 604, 619
get-rect method 637
get-referenced-object method 582
get-region method 328
get-resource-from-file method 501
get-scroller method 475
get-selected-text method 382
get-selection-data method 567
get-selection-font method 382
get-selection-font-size method 382
get-selection-font-sizes method 382
get-selection-fonts method 382
get-selection-font-style-handle method 383
get-selection-font-style method 383
get-selection-font-styles method 383
get-selection method 381, 567, 590, 598
get-selection-range method 382
get-string-from-library method 641

get-string method 641
get-string-offset method 641
get-style-handle method 383, 496
getter-function slot 524
getter options 35, 60
get-text-handle method 381
get-text method 381
get-window-color method 415
get-window method 347, 364
global coordinates 313
global-mouse slot 269
global-to-local method 326
graf-port slot 321
graphics buffers 309, 348
graphics caches 310
graphics concepts 307
graphics ports 308
graphics-port slot 359
green slot 325
grid view behaviors 388
grid views 387
  column widths 390
  drawing 389
  row height 391
  selecting cells 390
  setting up 389

H

handle-accept-drag? method 583
handle-draw method 365
handle-event method 276
handle-receive-drop method 584
handles, typed 554
handle-set-cursor method 365
handle-setup-menus method 277
handle-size slot 520
handle streams 513
handle strings 631
handle-update method 277
has-disk-file? method 487
has-reply? method 620
height method 637
hide-on-deactivate? slot 451
highlighting 343, 424
hilite-adorners method 365
hilite-cell method 398
hilited? method 442
hilited?-setter method 442
hilite-item method 398
hilite method 365
hilite-region slot 577
horizontal-size-sub-views slot 472

I

icon method 296
icon-setter method 296
identifier slot 264, 267, 361, 363, 414, 455
ID slot 299
implicit type mapping 89
imported functions 71, 75
imported members 72
imported objects 12
imported variables 71
importing pointer-to-function types 99
importing pointer-to-structure types 98
importing structures 95
importing type definitions 87
importing values 101
import options 20, 26, 56
import-value function 102
inaccessible objects 193
inactive-hilite slot 358
index slot 294
init-busy-cursor method 649
init-external-reference method 653
init-framework method 651
init-function slot 650
initialize-framework method 651
inline-constants options 21, 28
Inline Machine Code Access Path 124
inline options 37
input-argument options 34

input-output-argument options 34
inset-bounds method 359
inset-region method 328, 638
inset slot 359, 377
install-debug-menu variable 646
installed? slot 299
installed-menus method 299, 302
install-menu method 299, 302
interface definition clauses 11, 39
interface definition options 11
interface definitions 11, 18
interface importation 11
intersect-region method 329, 638
intersect-with-region method 329, 639
invalidate-all method 366
invalidate-all-on-resize? slot 359, 440, 448
invalidate-caches method 367
invalidate-cell method 398
invalidate-cursor-region method 367
invalidate-extent-changed-all method 364
invalidate-extent-changed method 364
invalidate-item method 398
invalidate-location-changed method 364
invalidate method 366
invert-region method 329
is-empty? method 558, 568
is-slot-editable? method 527, 528
is-sub-view-of? method 365
italic? method 330
item-bounds method 398
item-list slot 577
item-reference method 582
item-reference slot 578
items slot 298

J

justification method 381, 439
justification-setter method 439

K

key-character slot 270

L

language file option 122
language options 21
large-object spaces 196
last-antimate slot 648
last-clear slot 648
last-command slot 539
last-user-item slot 300
left slot 636
library, framework 629, 633
library-name slot 650
library-resource-files slot 650
list views 387
load-string method 641
local? slot 577
local-bounds method 313, 326, 359
local coordinates 313
local-mouse slot 274
local-slots slot 523
local-to-global method 326
local-to-root method 364
local-to-super method 326
location 311, 317
location-changed method 366
location slot 319
lock-buffer method 327
lock-count slot 322
locked? slot 358
low-level value 94
low-level values 101

M

machine pointers 138, 146, 152
main-file-creator slot 486
main-file slot 486

main-file-type method 484, 488, 502
main-file-type slot 486
main-handler variable 259
main-window? slot 411
main-window slot 406
make-ae-desc method 620
make-ae-list method 621
make-alias method 501
make-class-spec top-level form 507, 528
make-descriptor method 604, 621
make-enumerated-descriptor method 621
make-error-string method 644
make-external-reference method 653
make-graphics-cache method 367
make-mouse-down-event method 279
make-object-reference-flavor method 572, 582
make-object-specifier method 619
make-offset-specifier method 622
make-os-error method 645
make-palette-window method 404
make-pict method 327
make-range-specifier method 623
make-region method 328
make-relative-specifier method 622
make-scrolling-window method 347, 402
make-slot-spec top-level form 507, 528
make-te-handle method 381
make-token method 623
make-toolbox-event method 279
make-typing-command method 382
make-windows method 346, 407, 483, 487
mark-item method 290, 303
mark method 296
mark-setter method 296
maximum-extent slot 412
max-text-length slot 377
max-value slot 380
memory-available function 200
memory management 193
memory-shortage-level variable 202
memory-shortage variable 202
menu-bar slot 293
menu-bar variable 293
menu classes 285
menu events 290
menu items 283
  About item 292
  creating 287
  for documents 480
menu-item slot 301
menus 283
  creating 287
  popup 431
  setting up 288
menu slot 453
message slot 269, 613
message-string slot 643
microseconds 187
middle generation 194
minimum-extent slot 412
minimum-free-block variable 201
minimum-free-bytes variable 201
min-value slot 380
modal? method 410
modal?-setter method 410
modal dialog boxes 419
  creating 425
  opening 432
modifiers method 583
modifiers slot 269
mouse events 245, 252
mouse-location method 583
multiple-selection? slot 396

N

name conflicts 55
name-mapper options 21, 62
name mapping 17, 55
name-mapping functions 62
naming conventions 55
new-item method 582
new-selection-start slot 543
new-selection-stop slot 543
new-style slot 543, 545

new-target slot 275
new-text slot 543
next-behavior method 278
next-handler slot 264
next-target slot 435
normal? method 330
notifiy-super-view-extent-changed method 365
notifiy-super-view-location-changed method 365
notify-sub-view-extent-changed method 366
notify-sub-view-location-changed method 365
number text, creating 430

O

object-array slot 652
object generations 194
object-index method 623
object model support classes 226
object streams 514
offset-in-super method 326
offset-region method 328, 638
offset slot 565
offset-to-column method 397
offset-to-row method 397
old-selection-start slot 543
old-selection-stop slot 543
old-style slot 543, 545
old-target slot 275
on? method 442
on?-setter method 442
open? slot 486
open-data-fork method 500
open-files method 488, 502
opening documents 479, 482
open-initially? slot 411
open method 276, 364, 415, 487, 500
open-resource-fork method 500
Open Scripting Architecture support, see scripting support 601
operation-string slot 643
option-key? slot 269
options 11, 18
originating-view slot 577
osa-class method 619
ostype-name-at method 167
ostype-name-at-setter method 167
outline? method 330
output-argument options 34
own-handle? slot 520
owning-window slot 321

P

page-increment slot 445
paint-region method 329
parent-menu slot 293
part-code slot 274
Pascal-string-at method 170
Pascal-string-at-setter method 170
paste data 589
peek-byte method 525
perform-command method 547
pixels slot 322
pix-map slot 322
pointer-at method 165
pointer-at-setter method 166
pointers, typed 555
pointer-to-function types 77, 99
pointer-to-structure types 98
point-in-region? method 328, 638
point method 637
point-to-cell method 397
poll-busy-cursor method 649
poll-event method 279
popup menus 431
popup-style slot 453
port-is-window? slot 321
pose-modally method 415, 432
post-command method 537, 548
post-draw method 367

pre-draw method 367
preemption 151
preemption status 178
prefix name mapping 56, 60
prefix options 21, 61
prepare-to-draw method 366
prepare-to-draw-region method 366
print-views method 415
proc-id slot 447, 449, 450, 451, 452
promise functions 572, 574
promises slot 579
property-type slot 616

Q

qd-graphics-port slot 411
qd-graphics-system variable 360
quickdraw-port method 368

R

range-start slot 564
range-stop slot 564
read-buffer method 526
read-byte method 525
read-bytes method 525
read-data method 493, 500
read-error-list-resource method 644
read-error-lists method 644
read-from-file method 488, 493, 502
read-from method 566, 582
read-from-stream method 516, 527
read-handle-string method 514, 527
read-long-integer method 526
read-long method 620
read-object method 516, 527
read-only options 21, 29, 35
read-pstring method 526
read-short-integer method 514, 526
read-string method 526
read-symbol method 526
reallocate method 520, 527
reason-string slot 643
recalc-text method 381
receive-drop method 584
record-clipboard-commands slot 594
record-editing? slot 377, 378
recovery-string slot 643
rectangles 630
  creating 317
rect method 637
recurring? slot 541
redid-command method 547
redoit method 536, 548
redo-name slot 541, 543, 546, 597
redo-selection slot 597
redraw-all method 366
redraw method 366
red slot 325
reenable-busy-cursor method 648
ref-con slot 447
region classes 314, 628
regions 317
regions-intersect? method 329, 638
registering documents 482
release-object-reference method 582
remove-adorner method 357, 368
remove-behavior method 264, 278
remove-child method 547
remove-graphics-cache method 365
remove-menu method 302
remove-object-from-table method 653
remove-sub-view method 357, 364, 475
remove-view-from-port method 367
remove-window method 415
rename options 21, 57
replace-string-parameters method 644
reply slot 613
report-error method 644
report-garbage-collections variable 202
requested garbage collection 194, 197
request-file-name method 500
require-rsrc-fork? slot 499
reset-framework method 651
reset-function slot 650

reset method 651
reset-stream method 526
resizable? slot 410
resize-handle method 566
res-menu-type slot 453
resolve-external-reference method 653
resolve-specifier method 622
resource-id slot 438
resources 492
  reading 495
  strings 631
  updating 495
restart method 651
restore-buffer-port method 327
result-type options 34
resurrecting objects 210
reveal-selection method 367, 375
revert method 487
RGB colors 316, 318
rgn-handle slot 324
right slot 636
root coordinates 313
root-to-local method 364
root view 337
root-view method 412
root-view-setter method 412
row-heigth method 397
rows method 392
rows-setter method 392
row-to-offset method 397
rsrc-fork-open? method 498
rsrc-permission slot 499
rsrc-ref-num slot 498
run method 651

S

same-window? slot 275
scripting-object slot 412
scripting support 601
  descriptors 603
  getting and setting property values 604
  object specifiers 603
  properties 603
  responding to Apple events 605
scroll-increment slot 444
scrolling 461
  setting up views 465
scroll-max method 448, 456
scroll-min method 448, 456
scroll-value method 457
seal-functions options 21, 29
seal options 34, 35
search-recursively slot 436
select-cell method 398
selected-cells slot 393
selected-entire-contents? slot 544
selection-rect method 367
selection-start slot 545
selection-stop slot 545
send-drag-data method 582
sender slot 455
send-event method 620
set-ae-desc method 620
set-bounds method 326
set-buffer-port method 327
set-busy-cursor method 649
set-ctl-mgr-color method 456
set-current-resource-file method 501
set-empty-region method 328, 638
set-end-of-file method 494, 501
set-event-record method 277
set-file-position method 501
set-fs-spec method 501
set-item-bounds method 582
set-key method 622
set-library-init-function method 634, 651
set-library-reset-function method 651
set-origin method 327
set-parameter method 620
set-point method 637
set-port-style method 330
set-property-value method 604, 619
set-rect-as-region method 328
set-rect-from-points method 638
set-rect method 637

set-rect-region method 638
set-response method 620
set-selected-text method 382
set-selection-data method 381, 567, 591, 598
set-selection-font method 382
set-selection-font-size method 383
set-selection-font-style-handle method 383
set-selection-font-style method 383
set-selection method 381, 567, 593, 598
set-selection-range method 382
set-source-bounds method 328
set-style-handle method 383
set-target method 347, 409
set-target-selection method 366
setter-function slot 524
setter options 35, 60
set-text method 381
setup-file-permissions method 488, 502
set-window-color method 415
shadow? method 330
shareable? method 528
shareable-instance slot 523
shift-key? slot 269
signed-byte-at method 153
signed-byte-at-setter method 154
signed-long-at method 162
signed-long-at-setter method 163
signed-short-at method 158
signed-short-at-setter method 158
size method 641
slot-list method 528
slot specifications 507
  getters and setters 509
  setting up 508
slot-symbol slot 524
slot-tile method 527
specify-from-alias method 501
specify-from-fs-spec method 501
specify-from-path-name method 501
spontaneous garbage collection 193, 197
start-busy-cursor method 648
start-drag method 583
start method 651
start-point slot 474
statically-typed-pointer types 93, 96
static generation 194
static text, creating 430
streamable? method 279
streamable-behaviors method 279
stream-handle slot 520
stream-position method 526
stream-position slot 519, 520
streams 505
  and primitive data types 516
  handle streams 513
  object streams 514
  reading from 515
  writing to 515
stream-slot? slot 525
string classes 628
string-handle slot 640
string-id slot 641
string-index slot 641
string-library slot 641
string-offset slot 640
strings 100
  handle strings 631
  resoure strings 631
structure clauses 21, 31, 41
structure members 15
structures 16
style-action slot 545
style method 296
style-setter method 296
subtract-from-region method 329, 639
sub-view-extent-changed method 369
sub-view-extent slot 472
sub-view-location-changed method 369
sub-views slot 357
super-bounds method 326, 359
super coordinates 313
super-frame method 326
super-to-local method 326
super-view-extent-changed method 354, 369
super-view-location-changed method 369
super-view method 359
super-view-setter method 359

symbol-cache slot 521
symbol-to-class method 528

T

tabbing behavior 432
target 249
  changing 263
target chain 248
  adding an event handler 259
  and event handling 251
  changing 254
target-handler method 266
target-handler-setter method 266
target slot 597
target-view slot 409
te-handle slot 376
terminate method 328, 526, 566
termination 207
text 373
  edit 378
  number 379
  number text 430
  static text 430
text-changed? slot 544
text-getter slot 394, 395
text-length method 381
text slot 439
text styles 315, 318
text-style slot 359
text view classes 373
text-view slot 543, 545
the-class slot 522
the-current-point slot 474
the-last-point slot 474
the-tracker slot 474
title method 294, 412, 486
title-setter method 294, 412, 486
title slot 437, 441
token-class method 623
tokenized-object method 623
toolbox-enabled? slot 294
toolbox-mark slot 295
toolbox-menu slot 299
top-left method 637
top slot 636
track-begin method 469, 475
track-constrain method 470, 475
track-drag-message slot 581
track-end method 469, 475
tracker-scroller slot 473
tracker-view slot 473
track-feedback method 471, 475
tracking 463
track-mouse method 468, 475
transfer vectors 136
translation 313
translation method 472
translation-setter method 472
transparent? slot 358, 438
triple-click? method 277
truncate? slot 439
typed data classes 551
typed handles 554
typed pointers 555
type mapping 17, 87, 89
type mapping strings 100
type options 21, 35
typing-command slot 376

U

undefine options 21, 24
underline? method 330
undid-command method 547
undoable? slot 541
undoit method 548
undo-name slot 541, 543, 546, 597
undone? slot 541
undo-redo method 547
undo-selection slot 597
unicode-character-at method 161
unicode-character-at-setter method 161
union clauses 21, 31, 41

union members 15
union-region method 329, 639
unions 16
universal options 51
unlock-buffer method 327
unoit method 535
unsigned-byte-at method 155
unsigned-byte-at-setter method 155
unsigned-long-at method 164
unsigned-long-at-setter method 164
unsigned-short-at method 159
unsigned-short-at-setter method 160
untyped-pointer types 92
update-coordinates method 367
update-origin method 367
use-color? slot 647
use-data-fork? slot 498
user interface classes 228
use-rsrc-fork? slot 498
uses-color? slot 447
use-styles? slot 377

V

validate-all method 366
validate-cell method 398
validate-item method 398
validate method 366
valid-token? method 623
value options 37
value translation 17, 87, 94
variable clauses 22, 35, 41, 60
variable-column-widths? slot 393
variable options 22, 35, 47
variable-row-heights? slot 393
vertical-size-sub-views slot 472
view classes 336
view determiner classes 341
view hierarchy 337
view-only? slot 377
views
  clipboard 591
  creating text views 374
  for controls 421
  grid 387
  list 387
  scrolling 465
  text 373
view-to-port method 367
view-to-port-offset slot 360
visible? method 327, 357
visible-bounds method 357
visible-bounds slot 357
visible method 359, 412
visible-setter method 359, 412
vol-ref-num slot 498

W

wants-to-be-target? slot 358, 377
was-visible? slot 412
width method 637
window-content-color slot 412
window-kind slot 409
window-ptr slot 409
windows 401
  creating 402
  floating 404
  for dialog boxes 404, 429
  for documents 483
with-focused-view macro 349
without-busy-cursor method 649
word-wrap? slot 376, 379
world slot 321
write-buffer method 526
write-byte method 525
write-bytes method 525
write-data method 494, 500
write-external-modules function 134
write-long-integer method 526
write-long method 620
write-object method 515, 527
write-pstring method 526
write-short-integer method 526

write-slot method 527
write-string method 526
write-symbol method 526
write-to-file method 488, 494, 502
write-to method 566, 582
write-to-stream method 516, 527
ws-template slot 523, 524

X

xor-region method 329, 639

Y

youngest generation 194

Z

zoomable? slot 410

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final pages were created on a Docutek. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

PROJECT MANAGER
Trish Eastman

LEAD WRITER
Linda Kyrnitszke

WRITERS
Marq Laube, Gary McCue

ILLUSTRATOR
Sandee Karr

PRODUCTION EDITOR
Lorraine Findlay

SPECIAL THANKS TO
Kim Barrett, Bob Cassels, Phil Kania, Ross Knights, Mike Lockwood, David Moon, Andrew Shalit, David Sotkowitz, Steve Strassman