

Rhapsody

Developer's Guide

Rhapsody

Developer's Guide

Jesse Feiler



AP PROFESSIONAL

AP Professional is a division of Academic Press

Boston San Diego New York
London Sydney Tokyo Toronto

Find us on the Web! <http://www.apnet.com>

This book is printed on acid-free paper. (∞)

Copyright © 1997 by Academic Press.

All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Excerpts from Chartsmith are copyright © 1997 by Blacksmith, Inc. All rights reserved.

Excerpts from OpenBase are copyright © 1997 by OpenBase International, Ltd. All rights reserved.

Excerpts from Create are copyright © 1997 by Stone Design, Inc. All rights reserved.

Excerpts from OmniWeb are copyright © 1997 by Omni Development, Inc. All rights reserved.

Excerpts from TIFFany are copyright © 1997 by Caffeine Software. All rights reserved.

All brand names and product names mentioned in this book are trademarks or registered trademarks of their respective companies.

Academic Press

525 B Street, Suite 1900, San Diego, CA 92101-4495

1300 Boylston Street, Chestnut Hill, MA 02167

United Kingdom Edition published by

ACADEMIC PRESS LIMITED

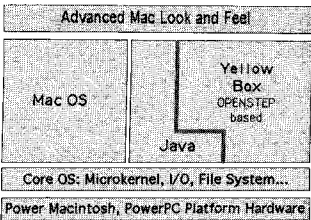
24-28 Oval Road, London NW1 7DX

ISBN 0-12-251334-7

Printed in the United States of America

97 98 99 00 CP 9 8 7 6 5 4 3 2 1

Table of Contents



Semantics Tables	xv
Preface	xxi
Why Rhapsody?	xxii
Redefining Ease of Use	xxiii
Opportunities for Developers	xxiv
About the Book	xxv
Three Useful Internet Addresses	xxv
Acknowledgments	xxvi
Introduction	i
The Search for Better Ways to Write Software	3

Improving the Production of Code	3
Reusing Code	6
System Software	8
Abstraction	8
The Evolution of Software.....	9
Increasing Complexity of Software	9
Software for the Twenty-first Century	9
Summary	15

Part I: Introducing Rhapsody

I. Rhapsody Architecture Overview	19
The Evolution of Rhapsody	20
The Beginnings of Operating Systems	20
Common Interfaces	24
Managing Changes	25
The Object-Oriented/Flat World Boundary	27
Splitting the Operating System	29
The Blue Box	31
Windows	33
Rhapsody on Mac OS	35
Writing for Rhapsody	37
Summary	38
 2. A Modern OS—The Mach Microkernel	 39
The Story So Far... ..	40
PersonalComputer Operating Systems	41
Sharing	42
Modern Operating Systems	43
Security	46
Process Management	47
Memory Management	50
Communication Between and Among Processes	50
Failure and Exception Handling	51
Microkernel Architecture	52
Mach	53
Tasks and Threads	53
Memory Management	54

Communications Between and Among Tasks	54
Mach and Other Operating Systems	55
Summary.....	56
3. Object-Oriented Programming	57
Object-Oriented Programming.....	58
Why Is It Liked?	59
Where Are the Benefits?	60
The Learning Curve	61
Objects	62
Object-Oriented Design Issues.....	64
Inheritance	64
Polymorphism	64
Data Hiding and Encapsulation	65
What It Means—No If Statements	66
An Example	67
Performance	70
Run-Time Issues.....	70
Dynamism	70
Managing Objects in Memory	71
Frameworks.....	72
Summary.....	72
4. The Languages of Rhapsody—Objective-C	73
Object-Oriented Programming Languages: Two Directions.....	74
Objects and Messages	75
The Look of the Languages	75
Modern Syntax	76
Dynamism.....	76
Dynamic Typing	76
Dynamic Binding	77
Dynamic Linking	80
Class and Instance Objects.....	80
Creating New Instances	80
Using Class Objects	81
Protocols	82
Keeping It Running	83
Syntaxes	84
Classic Syntax	84
Modern Syntax	84

Summary	85
5. The Languages of Rhapsody—Java	87
The Look of Java	88
Unicode	88
Java Isn't C	89
Everything Is an Object	90
Object Class	90
Some Java Syntax	92
Inheritance and Organization in Java.....	93
Packages	94
Frameworks and Java	94
Exception Handling	95
Summary	95
6. The Languages of Rhapsody—C++	97
The C++ Objective	98
Consequences of Compile-Time Typing.....	100
Small Objects Are Not Expensive	100
The Fragile Base Class Problem	102
Run-Time Type Identification and Exception Handling	102
Multiple Inheritance.....	103
The History of Multiple Inheritance in C++	103
C++ Delegation	104
Rhapsody and C++.....	107
Summary	109

Part II: The Rhapsody Frameworks

7. Rhapsody Frameworks Overview	113
Development Tools	114
Project Builder	114
Interface Builder	114
The Development Components	115
Nibs	115
Bundles	116
NSApplication	117

Resources and Dictionaries	117
About This Part of the Book	117
Functionalities	118
Subject Areas	119
Semantics Tables.....	120
Fabrication	121
Accessors	122
Grouped Methods	123
Optional Arguments	123
Class Methods	123
Summary	124

8. Terminology and Basic Concepts 125

Programming Design Terminology.....	126
Objects	127
Events	127
Actions	128
Responders	128
Responder Chain	129
Delegates	130
Outlets	131
Connections	131
Interface Terminology.....	132
Cells	132
Box	133
Browsers	133
Panels	134
Types	134
NSRange	134
NSComparisonResult	134
NSObject.....	135
Architectural Overview	135
NSObject Functionalities	136
NSInvocation	145
NSProxy.....	146
Summary	147

9. Building Blocks and the Application Object 149

Data Types	150
Simple Data	150

Pointers and Typed Data	155
Dates and Times	160
Error Handling.....	167
Exceptions	168
Assertions	170
Programming Tools.....	171
Arrays and Enumerators	172
Dictionaries	177
Text.....	181
Characters and Strings	181
Scanning Text	190
OpenStep Applications.....	192
NSApplication	192
NSBundleAdditions	205
NSNibAwaking[P]	206
Summary.....	206

10. Visualization 207

Color	208
NSColor	209
NSColorList	213
Controls.....	215
NSBox	215
NSCell	216
Images.....	228
NSImage	229
NSImageRep	235
Programming Tools.....	237
NSView	237
NSScreen	259
NSWindow	259
Window Delegate Methods	280
Summary.....	281

11. Interface Design and Controls 283

Commands	284
Menus	284
Drag-and-Drop	296
Panels	301
Controls.....	303

NSControl	303
Buttons	308
Summary	314
12. Living in a Shared Environment	317
Archiving and Distribution	318
Coding	319
Archiving	325
Distribution	328
Controls.....	329
Notification	330
Selection	336
Data Types.....	337
Data Links	338
Display.....	349
NSPasteBoard	349
Programming Tools.....	355
Copying	356
Synchronism.....	357
Kinds of Synchronism	358
Opportunities for Synchronism	360
Locking	361
Threading	366
Tasks	367
Connections	369
Summary.....	372
13. Files and Documents	375
Controls.....	376
Opening and Saving Files	376
Programming Tools.....	382
NSBundle	382
NSFileWrapper	385
Summary.....	390
14. Interaction, Events, and Responders	391
Programming Tools.....	392
NSEvent	392
NSResponder	398

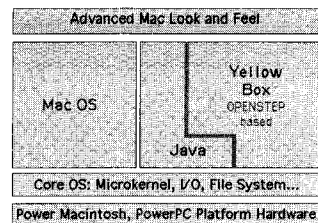
NSServicesRequest[P]	408
Summary	408
15. Printing	409
Controls.....	410
NSPrintPanel	410
Programming Tools.....	412
NSPrintInfo	413
NSPrintOperation	416
NSPrinter	420
Summary	422

Part III: Using Rhapsody

16. Dynamic Binding and Loading Resources with TIFFany	425
Image Processing.....	426
Placing Markers in Ruler Views	428
Setting the Markers	428
The View's Declarations	430
Caffeine Software	432
17. Dynamic Loading with Create	433
Create	434
Creating an Animated Star	434
Loading an Object and Its Code at Run Time	437
When to Load the Animator	438
Stone Design.....	440
18. Providing a Service and Using Distributed Objects with CHaRTSMITH	441
CHaRTSMITH	442
CHaRTSMITH at Work	443
Providing a Service	444
Setting up the Services Menu	444
Registering the Service at Startup	447
Implementing the Functionality	447

Using Distributed Objects	449
Making a Connection (from the Client)	451
Allowing a Connection (from the Server)	453
Symmetry of Distributed Objects	454
CHaRTSMITH	455
19. Using Services with OmniWeb	457
OmniWeb	459
OmniWeb Services	460
Adding a News Reader Service to OmniWeb.....	461
Registering New Functionality	462
The Code	463
Using Services	470
Omni Development	470
20. Notifications with OpenBase	471
Using Databases	472
Starting a Database with the Database Manager	473
Connecting to a Database from a Client	475
Notifications of Database Changes	477
OpenBase	480
Index.	483

Semantics Tables



The Rhapsody framework that is based on OpenStep is described in Part II. The semantics of the classes and methods are described in a language-neutral manner in the semantics tables. For further information about the semantics tables, see “Semantics Tables” on page 120.

8. Terminology and Basic Concepts	125
NSObject Fabrication Methods	138
NSObject Introspection Methods	139
NSObject Identification Methods	140
NSObject Messaging Methods	140
NSObject Garbage Collection and Memory Management Methods	142
NSObject Archiving Methods	143
NSObject Error Handling Method	144

NSObject Dynamism Methods	145
NSInvocation Methods	145
NSProxy Methods	146
NSInvocation Methods	148

9. Building Blocks and the Application Object 149

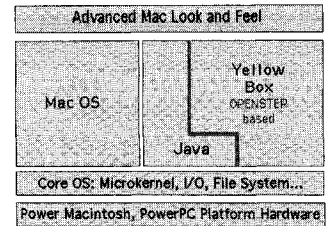
NSData Fabrication Methods	151
NSData Accessor Methods	152
NSData Utility Methods	153
NSMutableData Fabrication Methods	154
NSMutableData Utility Methods	155
NSValue Methods	156
NSNumber Methods	157
NSDecimalNumber Methods	159
NSDate Fabrication Methods	161
NSDate Utility Methods	163
NSDateCalendarDate Methods	164
NSTimeZone Methods	166
NSTimeZoneDetail Methods	167
NSException Methods	170
NSAssertionHandler Methods	171
NSArray Fabrication Methods	172
NSArray Utility Methods	173
NSMutableArray Methods	176
NSEnumerator Method	177
NSDictionary Fabrication Methods	178
NSDictionary Utility Methods	179
NSMutableDictionary Methods	180
NSCharacterSet Methods	182
NSMutableCharacterSet Methods	183
NSString Methods	183
NSString Encoding Methods	185
NSString String Manipulation Methods	185
NSString File Path Name Methods	189
NSMutableString Methods	189
NSScanner Utility Methods	190
NSApplication Management	194
NSApplication Events and Actions	197
NSApplication Window Management	199
NSApplication Modality	201
NSApplication Panels	202

NSApplication Managing Menus	204
NSBundleAdditions Methods	205
NSNibAwaking Method	206
10. Visualization	207
NSColor Methods	210
NSColorList Methods	213
NSBox Methods	215
NSCell Fabrication Methods	218
NSCell Action and Responder Methods	218
NSCell General Accessors	221
NSCell Accessors for Text	224
NSCell Image Accessors	227
NSCell Action Accessors	228
NSImage Methods	229
NSImageRep Methods	235
NSView Fabrication Method	239
NSView Hierarchy Methods	240
NSView Geometry Methods	242
NSView Notification Methods	245
NSView Size, Focus, and Display Methods	247
NSView Drawing Methods	250
NSView Event and Mouse Handling Methods	252
NSView Menus and Scrolling Methods	254
NSView Paper Handling Methods	256
NSScreen Methods	259
NSWindow Fabrication Methods	261
NSWindow Geometry Methods	261
NSWindow Window Management Methods	266
NSWindow Drawing Methods	269
NSWindow Window Server and Screen Methods	271
NSWindow Event and Key View Methods	273
NSWindow Editing Methods	275
NSWindow More Window Management Methods	276
NSWindow Menus, Cursor, and Dragging Methods	278
NSWindow Final Methods	279
NSWindow Delegate Methods	281

11. Interface Design and Controls	283
NSMenu Methods	289
NSMenuValidation [P] Method	293
NSMenuItem [P] Methods	294
NSDraggingSource [P] Methods	298
NSDraggingDestination [P] Methods	299
NSDraggingInfo [P] Methods	301
NSPanel Methods	302
NSControl Methods	304
NSControl Methods That Are Described in NSCell	307
NSButton Methods	309
NSButton Methods Described Elsewhere	312
NSButtonCell Methods	312
NSButtonCell Methods Described Elsewhere	314
12. Living in a Shared Environment.....	317
NSCoding [P] Methods	320
NSCoder Coding Methods	322
NSCoder Object Coding Methods	324
NSCoder Utility Methods	324
NSArchiver Methods	325
NSUnarchiver Methods	327
NSPortCoder Methods	329
NSNotification Methods	333
NSNotificationCenter Methods	334
NSSelection Methods	337
NSDataLink Methods	340
NSDataLinkManager Methods	343
NSDataLinkManager Delegate Methods	345
NSDataLinkPanel Methods	348
NSPasteboard Methods	350
Pasteboard Owner's Methods	354
NSCopying [P] Methods	357
NSMutableCopying [P] Methods	357
NSLock Method	363
NSConditionLock Methods	364
NSRecursiveLock Method	365
NSLocking [P] Methods	366
NSThread Methods	366

NSTask Methods	368
NSConnection Methods	370
13. Files and Documents.....	375
NSSavePanel Fabrication and Accessor Methods	377
NSSavePanel Utility Methods	379
NSSavePanel Delegate Methods	381
NSOpenPanel Methods	382
NSBundle Methods	383
NSFileWrapper Fabrication Methods	386
NSFileWrapper Utility Methods	387
NSFileWrapper Methods for Directories	388
14. Interaction, Events, and Responders.....	391
NSEventTypes and Flags Constants for NSEvents	394
NSEvent Fabrication Methods	395
NSEvent Accessors	397
NSResponder—First and Next Responder Methods	399
NSResponder Message and Command Management Methods	400
NSResponder Mouse Event Methods	402
NSResponder Keyboard and Text-Related Event Methods	403
NSResponder Other Event Methods	406
NSServicesRequest Methods	408
15. Printing.....	409
NSPrintPanel Methods	411
NSPrintInfo Methods	414
NSPrintOperation Methods	416
NSPrinter Methods	420

Preface



This book is a first look at Apple's new operating system code named Rhapsody. Rhapsody's first year is punctuated by three releases:

- *Developer Release is the first release of Rhapsody, seeded to Apple developers to let them become familiar with the technology.*
- *Premiere Release is the first public release of Rhapsody. Still very much a work in progress, Premiere Release lets users explore the new technology. Developers who have started with Developer Release can release early versions of Rhapsody software.*

- *Unified Release pulls together all of the pieces of Rhapsody (including the Mac OS compatibility component).*

Rhapsody itself is built on technologies developed at NeXT and at Apple. Much of the technology is part of the OpenStep (formerly NextStep) object-oriented framework, which has been ported over to Rhapsody. Additions to OpenStep implement Apple technologies (AppleScript, the QuickTime Media Layer, ColorSync, etc.). Other frameworks that were NeXT products (the Enterprise Objects Framework and WebObjects, for example) are also joining the suite of Rhapsody frameworks.

Since the entire system is object oriented, changes (particularly interface changes) that would be unthinkable in other environments can easily be made both in Rhapsody and in application programs. For example, the Rhapsody user experience design in both Developer and Premiere is considered preliminary. This shouldn't be of too much concern to developers: the frameworks take care of everything. It doesn't matter where the menus are or how they are displayed as long as you properly create them and enable their items.

The Rhapsody architecture is well established. Some parts of the final versions of Rhapsody are incomplete or missing in Developer Release—and in this book. Apple's documentation (on line and on paper) tracks the releases of Rhapsody in detail. This book provides an overview of the architecture and details of the basic building blocks. If you master these basics, learning how to use other pieces of the system as they become available should not be difficult.

This book represents the version of Rhapsody (pre-Developer) that was presented to Apple developers at the World Wide Developers Conference in May 1997.

Why Rhapsody?

The vast majority of people in the world today do not use computers. Computers don't do the things that these people want to do, and they are too hard to use. Making computers do things that matter to people is something for you, the de-

veloper, to address. Making them easier to use is a joint effort by you and operating system designers.

Redefining Ease of Use

“Ease of use” means not just intuitive and powerful interfaces, but also—and more important—reliable and stable software. Consider the simple admonition, “Make sure to save your document periodically.” How much the user must know about in order to cope with the undependability of application programs, system software, and hardware! How strange it is to save something that’s not done—most people are used to putting things away when they’re finished with them, when their work is complete. Save half a paragraph? Store an unfinished database entry? Absurd, unintuitive, and unnatural for most people (that vast majority of the world’s population referred to previously).

Necessary and understandable for those few people who have adjusted to using the awkward and capricious devices called computers.

The adventures you experience while attempting to use a computer to do something you care about (be it work or play) are not limited to the crashing of hardware or software: these are just the most dramatic. Two other common adventures make life unpredictable for computer users.

First, the computer sometimes takes off on a journey of its own, leaving you staring wide-eyed at an unresponsive machine. Incoming faxes are a notorious culprit, often seriously degrading—if not stopping—all other processes. Automated processes—disk backups, for example—manage to automatically start at precisely the worst moment, leaving you high and dry. A computer that cannot easily and almost invisibly handle multiprocessing can never be totally under your control.

Second, the barriers between your data as stored in one format or another and as processed by one application or another

are formidable. At some point, someone will point out that it is your data, and that you have bought and paid for each of those applications: shouldn't you have some sort of control? Shouldn't you be able to force them to get along together? A computer that cannot easily exchange messages and data among its various applications will frustrate its users whenever they try to do anything that involves putting things together (which is what people do best). A computer that itself cannot fit its data and applications into a shared environment with other computers (and other people) can never be part of the kind of collaborative efforts that make great projects possible.

Rhapsody does all this. Its messaging, sharing, and multiprocessing features are precisely those features that people have expected in computers and have rarely found. The groundwork is laid with Rhapsody for a new generation of software where "ease of use" takes on a much broader meaning.

Opportunities for Developers

Not only is it too hard to use computers, it is too hard to program them. Rhapsody brings a new type of programming to computers—the graphical Project Builder and Interface Builder as well as pervasive object-oriented programming.

Building on this newfound ease of programming, you can develop traditional applications far more quickly; you can expand the power of existing applications significantly; and you can do things you haven't done before. Things that, perhaps, no one has ever done before.

Projections are that early adopters of Rhapsody will be power users who will be running servers and doing very computation-intensive tasks. That may well be true, and they will need very heavy duty software to help them (which will be easier to write in Rhapsody than in other environments). But don't make the mistake of thinking that even these high-powered early adopters of Rhapsody won't also need a host of other applications. Whether on their own (like games) or integrated

with other applications as Rhapsody services, these other applications offer major opportunities for developers large and small.

If it costs far less to develop software for Rhapsody, presumably that cost saving can be passed on to consumers. Affordable and reliable software may be in reach for all of those people who don't now use computers. As noted before, that is an enormous market—and it is not a market that is readily available to people who cannot deliver powerful and dependable software like the software that runs on Rhapsody.

About the Book

The first part of the book describes the basic Rhapsody environment. The second part provides an in-depth look at the main framework of Rhapsody. The final part gives you case studies (with screenshots and code) from several Rhapsody products. These examples demonstrate specific Rhapsody functionalities, particularly those that are unique to its architecture. You may find it useful to read this part of the book before Part II.

Three Useful Internet Addresses

For more news about Rhapsody the primary source is Apple:

<http://www.apple.com>

<http://devworld.apple.com>

For updates to this book:

<http://www.philmontmill.com/rhapsody>

For more news about AP Professional:

<http://www.apnet.com/approfessional>

Acknowledgments

Writing a book on technology such as *Rhapsody*—particularly at a very early stage of the project—is impossible without the help of many people. At AP Professional, Ken Morton was (as always) a pleasure to work with in making the book happen; Mike Williams was absolutely indispensable to the production process and was helpful in the wording of these acknowledgments. Samantha Libby and Julie Champagne were equally efficient and helpful, as was Mary Prescott, whose copy editing made the book easier to read.

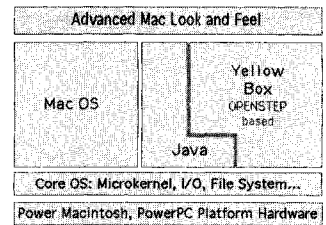
John Wendt provided the technical edit of the book. This was an enormous task, accomplished under severe time constraints. His comments and suggestions were invaluable and always helpful.

At Apple, a number of people have been very helpful on this book and its predecessors. Among them are Patrick Collins, Janie Phillips, Terry Donoghue, Ricardo Gonzales, Ron Hayden, Henri Lamiroux, Matt Morris, Steve Naroff, and Alan Samuels.

Jim Black organized a session at the end of the 1997 World Wide Developers Conference that was called “A View from the Trenches.” It featured a number of developers who had written software for OpenStep; they demonstrated their products and described the process they had gone through to compile them on the very first version of *Rhapsody*. This session was one of the most impressive of the conference and served as the basis for the last part of this book. Jim and the five developers who have allowed their code to be used as examples in the last part of this book (Stan Jirman, Scott Keith, Andrew Stone, Greg Titus, and Tom Trainer) have made an enormous contribution to the book.

Notwithstanding the assistance of all of these people, any errors that remain are the author’s handiwork alone.

Introduction



Apple's new operating system (code named "Rhapsody") is a departure from previous operating systems both on mainframes and on personal computers. Its structure is thoroughly modern—it is based on a small microkernel that isolates the most critical aspects of the operating system, and it is object oriented to a degree that has not been seen in a mainstream operating system. Its user experience has been developed based not only on Apple's years of research and development in this area but also on the powerful processors of today and tomorrow that can provide a rich, sophisticated, and delightful user experience of greater elegance and power than their earlier incarnations could provide.

This book is for Rhapsody developers—for people who are starting to explore Rhapsody to find out what's there and what the opportunities are. It is about writing software and about how Rhapsody makes the developer's job much easier—and very different.

Writing computer software can be very rewarding—both intellectually and financially. Yet it requires scrupulous attention to detail and often consists of continuing repetitions of nearly similar tasks—tasks that must be carried out with attention and accuracy, despite their likenesses. Errors can cause immediate problems, or they can lie in wait until a particular combination of circumstances conspires to loose their destructive forces on unsuspecting people.

The work is extraordinarily labor intensive, consuming large amounts of the lives of often highly paid people while requiring them to repeat (not mindlessly but mindfully) the same tasks over and over, at the same time exacting high prices for even the slightest flaws or inattention to these complex, repetitive processes.

For nearly 50 years, people have looked at the process of programming computers and tried to improve it. The impetus for this comes from programmers themselves as well as from their managers. Management, of course, would like to reduce costs, improve the reliability of code, and produce software faster. These incentives are nothing compared to the goals of programmers—people who find themselves writing sort routines, natural language parsers, report formatters, and mouse-tracking procedures over and over and over again. Each version is almost like the last one—similar enough to pose little imaginative challenge, yet different enough to be riddled with snares for the inattentive.

Meanwhile, the nature of computer software has been changing, with the focus moving from mainframes to personal computers and then to networks of machines. As the nature of software has changed, enormous advances in hardware (processing speed, memory, and storage) coupled with dramatic decreases in their costs have allowed developers to experiment with new techniques that previously would have been impractical to implement.

The Search for Better Ways to Write Software

Perhaps more than the economic motives of management, the sense among many programmers that they are wasting enormous amounts of time repeating and repeating the same old code has inspired a decades-long search for better ways to develop software.

This search for better ways to write software has centered on three primary areas:

- the process of producing code
- the reuse of existing code
- the abstraction of code

Improving the Production of Code

In this strategy to improve the production of software, the focus is on the actual production of code. Compilers and code generators automate its production while techniques of coding such as structured programming improve the quality of the code. Nontextual programmer tools bring the power of graphical user interfaces to programmers (at last!).

Compilers and High-Level Languages Strange as it may seem today, the possibility of writing a compiler was not widely accepted in the early 1950s. Compilers and high-level languages (starting with Fortran, Algol, and Cobol in the late 1950s) have grown far beyond their early implementations.

The idea of compilers is to allow people to program in ways that are more natural to them than the peculiar instructions of individual computers. In 1978, Dr. Grace Hopper, one of the most important early compiler developers (she led the group that developed A-0 for the UNIVAC), recalled in an address on the origins of compilers:

[T]he primary purposes [of building compilers] were not to develop a programming language, and

*we didn't give a hoot about commas and colons....
We were trying to solve problems and get answers....*

We were also endeavoring to provide a means that people could use, not that programmers or programming language designers could use. But rather, plain, ordinary people, who had problems they wanted to solve. Some were engineers; some were business people. And we tried to meet the needs of the various communities. I think we somewhat lost track of that, too. I'm hoping that the development of the microcomputer will bring us back to reality and to recognizing that we have a large variety of people out there who want to solve problems, some of whom are symbol-oriented, some of whom are word-oriented, and that they are going to need different kinds of languages rather than trying to force them all into the pattern of the mathematical logician.¹

The “tough-guy” early programmers who struggled with machine code and then with assembly language often looked down on the softies who used compilers and high-level languages. The code generated by compilers was never so efficient as that written directly in assembler or machine code (or so they said). With today's highly optimized compilers and blazingly fast processors, such arguments surely have little basis in practical reality. (But a reminder such as this is always welcome in some quarters.)

Code Generators Code generators represent another approach to the problem of improving the production of software. Code generators—which in fact antedate compilers—

1. *History of Programming Languages*, Richard L. Wexelblat, editor. Academic Press, 1981, p. 11. The address was given as the keynote speech at the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978.

let you specify the nature of the problem you want to solve (a file to be sorted, a report to be produced, etc.) and they then produce the code to do the job. This code can be low- or high-level code, depending on the particular code generator. Some generators are quite complex—Burroughs Corporation's LINC system in the 1980s allowed users to specify screen layouts, report formats, and database designs, and it generated Cobol code as output. The code that is produced can often be modified manually to fine-tune the end product.

Code generators are a very efficient means of producing basic code with little customization. Rhapsody's Project Builder and Interface Builder allow you to create the basic files of an application automatically; stubs for classes and methods are generated automatically as you specify them graphically.

Structured Programming and Other Techniques In addition to compilers and code generators, a third technique for improving the production of code was developed early on. It is in essence the "good faith" efforts of programmers to police themselves and to write understandable code. Whereas computers and compilers are quite persnickety about the code they read and are intolerant of even the smallest faux pas, they have infinite patience when it comes to deciphering the complexity of the code. People, however, are just the reverse. In reading code (or any text, for that matter), the eye and brain often correct minor errors without your noticing it. (That is why you can so often not find a compiler error that consists of a single misplaced letter.) By the same token, following a convoluted logical process quickly taxes the poor human brain.

Realizing that the challenge is often to produce code that can easily be read and understood by people (who are modifying it at some time in the future), programming styles and techniques have been developed to make it easier for people to read code.

These techniques include the notion of structured programming (from which goto statements are banished) and more re-

cently the development of object-oriented programming (from which conditional statements are largely excluded). The results of these two stylistic decisions are programs that consist largely of declarative statements that are executed sequentially in the same way each time. Such programs are much easier to write and debug than they would otherwise be.

Graphical Tools for Programmers Programmers often do not get to use the fruits of their own labors. Recognized productivity improvements for end users such as databases and graphical user interfaces have been slow in coming to the programmer's environment. With Rhapsody, developers will find an environment that is heavily graphically oriented. Other environments have provided more or less successful interfaces to text-based tools; with Interface Builder, you will find a true graphical programming tool. And since its primary purpose is to build graphical interfaces, that is totally appropriate.

In fact, experienced programmers who used Interface Builder when it was a product of NeXT, refer to "ASCII code"—the text-based code that is written to augment their graphically written code. No longer is it assumed that "code" means text.

Reusing Code

The advantage of reusing code has been obvious to programmers for decades. Every programmer has reused code, whether from a previous project, from a colleague, from code samples, or from books and magazines.

Reusing code provides obvious and immediate benefits, but at a cost. For code to be reused, it must be directly applicable to the problem at hand. Routines that calculate mathematical functions (sines, cosines, etc.) are fairly easy to reuse. As soon as the code starts to be customized for a specific purpose, however, its reuse becomes problematic. Of course, it is precisely such customized code that people want to reuse; in do-

ing so, all too often the advantage of reusing old code outweighs the disadvantage of its not being quite right for the job. More often than not, it is the user who suffers with an interface that just isn't quite right.

In order to deal with the problem of reusing code that isn't exactly right, two approaches have been developed to make the situation better.

Parameter-Driven Code For many years now, code has been factored into functionally separate parts. The interface including variables that change from time to time is divorced as much as possible from the operational side of the program. As needed, the program queries the interface (or a file) for the parameters it needs to carry out its work.

The use of resources on the Mac OS is a prime example of parameter-driven code. The appearance of windows is specified in resources that can be changed without recompiling the program itself. Text can be changed from one language to another without touching source code. Over time, even fairly complex issues such as centering of windows on monitors have been moved into resources rather than application-specific code.

In Rhapsody, the parameters and data are located in files of their own. Whereas on the Mac OS resources need to be placed in the program file, customization and localization in Rhapsody requires no change at all in the application file.

Object-Oriented Programming You can consider object-oriented programming as an issue of programming style as well as a strategy to reuse code. It allows the reuse of existing code while allowing its modification (overriding). The modification of preexisting code can be done at compile time as well as at run time with dynamic linking.

Frameworks extend the benefits of object-oriented programming into a more generalized sphere. Until recently, frameworks were viewed much as high-level languages and

compilers were by assembly-language programmers. They were considered inefficient tools for sissies who didn't want to do "real" programming. Of course, nothing could be further from the truth, and programmers (and managers) gradually became aware that there were large sections of critical pieces of software that duplicated one another and that were so patched up that it was almost suicidal to make any changes to them. The issue today with regard to frameworks is which one to use, not whether or not to use one.

System Software

One final approach to reusing code is the gradual incorporation of common functionality into the operating system. Originally, operating systems provided the minimal functionality necessary to run application programs on a computer. Gradually, functions that were written—over and over—by application programmers have migrated to the system software level.

Thus, in mature operating systems programmers can expect to find utility routines that manage arithmetic, trigonometric, and date functions; communications between and among programs and computers are managed increasingly by the system software and not by the programs themselves.

Abstraction

The last main approach to improving the development of software has been the increasing use of abstraction. Initially, computer code had to be written for the specific computer on which it was to be run. Even as late as the 1960s, programmers needed to know the number of the tape drive used for standard input on their computer and the number of the tape drive used for standard output. Programs written for a computer whose tape drives were numbered differently would not run.

From the earliest days of computing, people realized that by providing an abstract concept of a "virtual machine" or a "pseudomachine," code could be written to interact with these imaginary machines—and at run time an operating sys-

tem could concretize the code to the machine on which it was running. This abstraction meant that programmers needed to learn how to write for the abstract machine (a general concept) rather than for individual machines.

The cost of abstraction lies in the additional work that the operating system must do at run time to concretize the code. As the power of computers has dramatically increased, this cost has been worth bearing—particularly when weighed against the advantages derived from allowing programmers to write in a more general way. In Rhapsody, the architecture of the system (which incorporates an abstraction away from the hardware layer) is based on the notion of hardware-independent abstractions; no additional run-time cost is incurred.

The Evolution of Software

While people have been dealing with the issues surrounding improvements in the production of software, the nature of software has changed. In response to the development of personal computers and the rise of networking (including the use of the Internet), the nature of software has changed substantially.

Increasing Complexity of Software

Along with the increasing power of computers, the complexity of commonly used software has increased. The increasing complexity of software is not merely a problem for users: for developers it has become a nightmare. As many applications have encompassed more and more functionality (often far beyond what their original designs considered), the code has become more fragile with each additional modification.

Software for the Twenty-first Century

Fewer than 25 percent of U.S. citizens use computers. Many of them use computers only sporadically—for periodic word processing, to access the Internet, or to do bookkeeping. Only a relatively few people actually use computers routinely and

regularly throughout the day. Most computers today are woefully underutilized, yet few of their users would willingly increase their use of the machines.

It has been suggested that all the major software that needs to be written has already been produced. Word processing, spreadsheets, communications, bookkeeping, databases, desktop publishing, etc.—sometimes it seems as if only relatively minor enhancements (and simplifications!) are needed in these categories. From this point of view, new software will follow along in these molds, customizing data structures, being applied to specific markets, and otherwise being refined—but staying the way that it is.

On the other hand, many people see an industry just waiting to take off. It is hard to underestimate the deleterious effects of the crisis in development of the mid-1990s. Software is just too complicated to produce in anything like a timely, efficient, and cost-effective manner—at least using the old techniques.

With a solution on the horizon for the difficulties of developing software, and with the dramatically increased power of computers, the software industry could indeed take off... but to where?

Software and the Internet The Internet is everyone's favorite destination these days, and its impact on the world of software can scarcely be overestimated. Internet protocols are relatively simple, yet they allow people to communicate and to share information with relatively little difficulty. The computer industry has seen in the Internet a way to disseminate quickly information about its products and to provide technical support to bewildered users (who are not quite so bewildered that they can't get to the Internet).

The Internet is also seen by many as a marketplace—a tool that can deliver digital products almost instantaneously anywhere in the world. Not surprisingly, one of the first digital products to be sold over the Internet was software.

Perhaps the Internet's greatest impact is, in a perverse way, due to its simplicity. People stymied by their word processors are finally getting up the nerve to say, "If I can connect to a computer halfway around the world, why can't I get a header to print on my report?"

Empowered by their connections to remote computers, they are starting to realize that the problem with printing headers might actually not be their fault—the software might be too complex. How few programmers realized that empowering the masses to surf the Net with a click of a mouse would turn people into demanding users, no longer content to adjust their activities to the arbitrary restrictions of computer software.

Activity-Centered Computing Increasingly, the focus of software developers is on people and the activities that they are carrying out with the aid (and occasionally the hindrance) of their computers. As computer software addresses more and more routine and mundane aspects of people's lives, the reluctance to invest substantial amounts of time and money in learning how to use the software grows.

The meadows of pastel notes taped to the sides of monitors around the world testify to the fact that it remains simpler to write a memorandum on a scrap of paper than to use a computer as an *aide-mémoire*. Any hope of expanding the market for computers and software must rely on increasing the simplicity of software products. It is true that the big things are done (probably), but the myriad pesky details of life that a computer could assist with remain. And until they are easier to handle using a computer, people will continue to save scraps of annotated paper, tape notes to doors, and keep an extra sum in their checking account to avoid having to strike a balance once a month.

Software for Work and Play One of the biggest areas of growth in the software industry for the twenty-first century is in all of those domains that remain relatively untouched by

today's software products. Computers and software have found a place in offices; in homes, they have become established in the office-like activities of a home (everything from a home business, to personal accounting, to the office-like activities of students preparing research papers).

A relatively small market does exist for computer games and diversions—but often these remain in the office mode. The vast range of activities that could use an assist from computers but that are carried on in places too noisy or dirty for computers (garages and workshops), or in places too dignified and quiet for computers, represent a vast untapped market for computers and their software.

The computer industry has learned how to put computers on desktops and on the laps of business travelers. It is taking the first few baby steps toward putting computers in briefcases, backpacks, and purses. The software that will help the industry break out of its relatively confined mold (and will provide fame and fortune for its developers and promoters) must also break out of the office model.

The U.S. Federal Communications Commission has estimated that two-thirds of the people alive today have never placed a telephone call. When considering the opportunities for computers, remember this point. You may be looking at a market that is close to being saturated in a very few areas of relative wealth and sophistication, but there is an almost unimaginable market out there for software that does the things the vast majority of the Earth's population want to do.

Chronology Versus Complexity Most people today have grown up to a certain extent with the growing up of the computer industry. In its evolution and development, the industry has provided ever-richer data types and more powerful processors. For people who have experienced this process, it is easy to fall into the trap of thinking that software has become more complex. In fact, it hasn't; the chronology is being confused with complexity.

Half a century ago, the popular impression of what a computer would be (as seen in the press and movies) included video, speech recognition and synthesis, and near-instantaneous communication among people around the world. The available products, constrained by limitations of existing hardware and software, fell far short of those expectations. Despite having lived through advances of hardware (and the concomitant reductions in the compromises users had to make to deal with the machines), you should remember that it is the current data types that are natural. Communicating with anything (person or machine) by typing in only capital letters (and expressing a gloss on the content by typing marks of punctuation sideways [:)]) is unnatural and confusing. The so-called advanced technologies are the basic ones.

Graphics aren't decoration; multimedia isn't optional. They are parts of our lives—and have been for millennia. Anyone who feels that all the basic software has been written is living in a two-dimensional, text-based world that is very quiet and still. Adventuresome people and those whose jobs have required assistance from computers have adapted to live in that world. These people constitute only a third of the population of the United States (and for many of them, their patience is wearing thin). In fact, there is a grain of truth to the notion that all the major software has already been written. It is highly likely that all the major software of the old kind has been written. The software for the vast majority of people in the world has yet to be written. It requires new ways of thinking about software and its development, and it requires a new tool.

OpenStep OpenStep is the basic framework that provides a robust, powerful, cross-platform development environment with the tools needed to develop for Rhapsody. For some, it will be your first experience with using an object-oriented framework. Still others will be exploring the world of object-oriented programming for the first time.

All of these are sophisticated concepts that ultimately make the job of developing—and maintaining—software easier. Re-

member, however, that the more new concepts you have to learn with OpenStep, the longer it may take.

The biggest challenge most programmers have in learning to use a framework like OpenStep is learning to trust it. It's there to be used—and while at first it may seem easier to go off and write some code of your own, in the long run it makes great sense to take a moment and see if OpenStep already has code to do what you want to do.

Almost all experienced programmers taking a one-week introductory course in MacApp exhibit similar behavior. On the first two days, they roll their eyes and complain about having to use the “complicated” framework. Instead of being allowed to write a dozen lines of code, they complain, they are forced to look through the framework to find the classes to use and the methods to override. All of this is woefully inefficient (they say).

Some time during the third day of most introductory MacApp classes, the experienced programmers get very quiet. As the examples get more complex and the lab exercises become closer to real-life situations, a hush generally descends on the class. And then things start to get noisy again. The programmers who were complaining about having to use an “awkward clumsy framework” to do something they could have done in “a few lines of code” are now excited that by overriding only one method they are able to implement dragging, undoable commands, or multiple selections. It is that moment—the moment when the power of the framework is unlocked—that convinces people that they are on the right track.

If you are new to frameworks, remember this. It will take a little while for you to get to the point where you trust the framework. Once you do, you'll be on the road to a powerful way to develop the exciting software that is part of the Rhapsody world.

Summary

Rhapsody provides an extraordinary computing experience for users. Its highly reliable performance is unfortunately a rarity on the desktop; it is as robust as many high-end main-frame systems. Its integrated graphics and networking make a strong statement to users and developers alike: this is a serious, twenty-first century computer.

The cross-platform features of Rhapsody (it runs on both PowerPC and Intel chips; applications written to it can be compiled to run on Windows and Mac OS) make it an attractive environment to decision makers at software development organizations.

But there is something else.

The process of developing applications for Rhapsody is unlike anything else. Developers and programmers get the same treatment that early users of the Macintosh got: intuitive, simple, highly productive tools. If you are old enough to remember typewriters, remember how liberating it was to use a word processor—no longer did you slow down at the bottom of a page, living in dread of a typo on the last line that would cause you to have to retype the whole page. As you start to develop for Rhapsody, you will experience the same sort of liberation.

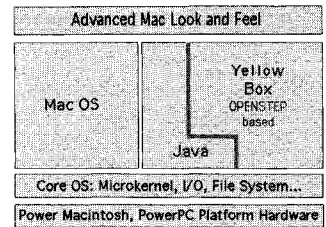
The system embodies the contemporary ideas about software development that have been touched on in this chapter. It is not Cobol or Assembler hidden under a mask of trendy buzzwords—it is new and exciting, and it can help you to remember that programming can be fun.

Whether your emphasis is on the act of programming or on the results that you deliver to your users, you will find the power of the Rhapsody environment invigorating. You may choose to use that power to shorten drastically your develop-

ment and maintenance cycles—or you may choose to use it to tackle projects that you never would have dreamed of doing before.

Programmers have been among the last to profit from the new technologies that they themselves develop for others. There may be a silver lining to that cloud. If you look at the development of programs as a generic task, the tools that developers have for that task are now at least as good as—if not better than—the tools anyone has to do anything on a computer.

Part I: Introducing Rhapsody



Rhapsody, the new operating system from Apple Computer, is built on firm foundations. These include the principles of modern operating systems (including multithreading, multiprocessing, a microkernel, and memory protection) as well as object-oriented programming and component software.

Rhapsody is also built on three proven, highly regarded, and very popular products—the Mac OS, the OpenStep (formerly NextStep framework), and the Mac look-and-feel, which has evolved from the “computer for the rest of us” in 1984 to today’s industry-leading PowerPC processors.

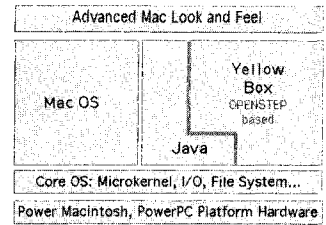
This part of the book provides a summary of the Rhapsody architecture, as well as the principles of modern operating systems and object-oriented programming.

Chapter I

Rhapsody

Architecture

Overview



This chapter covers the Rhapsody architecture: it is a unified, robust architecture that has several implementations. Rhapsody itself—with its own distinctive look, feel, and functionality—runs on PowerPC and Intel processors. In addition, applications written for Rhapsody can run on their own on traditional Mac OS and Windows; when they do, they bring parts of the Rhapsody experience along with them, but they adhere to the look, feel, and functionality of the platform on which they are running.

You can find the day-to-day record of how Apple arrived at this architecture in old magazine and newspaper clippings; there were many false leads and turns in the road at Apple and NeXT as well as throughout the computer industry and academe over the last half-

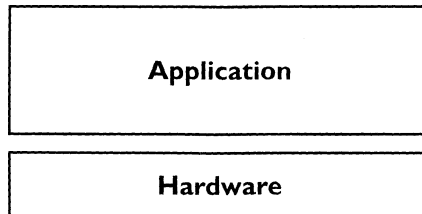
century. At times, the direction in which the computer world was headed was not clear; in retrospect, it is possible to trace the evolution of the architecture of Rhapsody without the digressions and distractions that in fact occurred on the way here.

In other words, this coherent story is what actually happened—although few people realized at the time where the adventure would end up.

The Evolution of Rhapsody

The first computers had no operating systems. They were rooms full of electronic components that were programmed with circuit boards and switches. Management of the computer and its resources was done by someone armed with a sign-up sheet; bills for the use of the machine were sent out by a secretary. A single application program ran the computer and did whatever it wanted to do, as shown schematically in Figure 1-1. The functionality delivered to the user consisted of the application and the hardware—the total area within the two boxes.

FIGURE 1-1. The Beginning: Program and Hardware



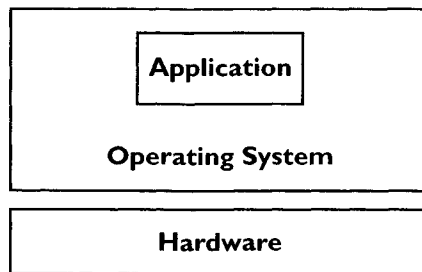
The Beginnings of Operating Systems

Operating systems evolved for a number of reasons—scheduling and billing for computer resources being among the primary ones. As it became possible for computers to run more than one job at a time, this feat of time-sharing was accom-

plished with operating systems. And so, by the end of the 1960s, operating systems were born out of the necessities to schedule, share, and bill for computer resources.

Figure 1-2 presents a schematic of the computer environment at this time: the computer hardware, an operating system, and within the operating system an individual application program.

FIGURE 1-2. A Basic OS



Increasing Functionality Figures 1-1 and 1-2 are drawn to a similar conceptual scale. Note that the application in Figure 1-2 is smaller than the single application in Figure 1-1, which had to do everything. Notice also in Figure 1-2 that the total functionality that could be provided to users is larger than that in Figure 1-1. The OS itself provides functionality (albeit in the early days limited); that functionality falls into three general areas:

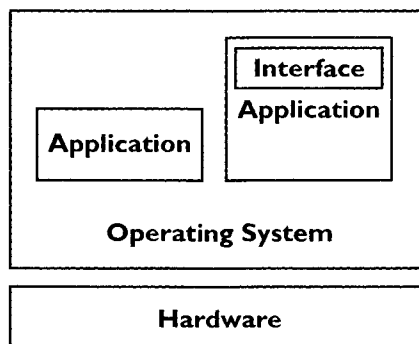
1. The OS can take advantage of economies of scale in development and deployment; code that otherwise would have had to be written inside individual applications (if it had been written at all) can now be written once and shared by various applications. For example, starting in very early times, application programmers left the mechanics of positioning tape and disk drives to the operating system. For this reason,

the individual applications are smaller.

2. On the other hand, the OS itself can provide direct services to the user: billing and scheduling of time-sharing jobs were some of the first. Thus, the total box of services encompassing both application programs and OS suggests more functionality than was provided by the single application program shown in Figure 1-1.
3. Finally, the OS provides services to itself that only it needs. This is usually a significant chunk of the processing of the OS. Operating systems are voracious consumers of computer resources in many environments. Mainframe systems running time-sharing often spend well over 50 percent of their processing power managing the OS itself.

With operating systems such as these, it was possible to run several application programs at the same time as shown in Figure 1-3.

FIGURE 1-3. Multiple Application Programs in an Operating System



The operating system has grown compared with the early one shown in Figure 1-2. Note also that the application program

on the right has grown in size: adding functionality is not confined to operating systems—users of programs want and need new features, too. Here a specific type of new functionality has been added—a sophisticated interface to the program.

Added processing power made it possible to write more sophisticated user interfaces; the increasing numbers of people using computers made such interfaces necessary. With the advent of graphical user interfaces and the widespread deployment of the Macintosh computer, people began to appreciate not only the advantages of a graphical user interface but also the benefits of a consistent graphical user interface across many applications. Standards for interfaces were promulgated and programmers (more or less) tried to adhere to them.

Isolation and Dependencies of System Components The first operating systems were relatively unconcerned with the internals of the applications that they ran. They started and stopped the jobs, sent the bills, and that was that.

Operating systems performed another valuable service. Without an operating system, the application program itself had to interact directly with the hardware. Any change to the hardware—even so slight a change as the renumbering of tape drives to accommodate a new device—required changing the programs.

As you can see from Figures 1-2 and 1-3, the application programs are removed from the hardware; under normal circumstances, application programs communicate with their operating system, which in turn deals with the hardware. Relatively large changes to the hardware can be made without having to change the application programs.

The boundaries between the objects in these diagrams are like the Earth's great tectonic plates: there is tremendous pressure and friction at these points. Two plates in the Earth's crust may fit together more because of the pressures holding them

in place than because of any simple fit. Geologic events that are relatively contained within a single plate can be magnified to catastrophic proportions when they confront the boundary between two plates.

Similarly, the boundaries in these diagrams are fraught with danger. Whereas the application running within an OS may be relatively immune to changes in the hardware level, the OS itself which abuts the hardware level, must change constantly in response to hardware changes. (On very rare occasions the hardware changes in response to software requirements.)

Thus, although the OS itself normally shields the application program's structure and code from hardware changes, the fact that the OS itself must change in response to hardware changes does mean that the application's environment is at least potentially disrupted by changes in hardware.

Common Interfaces

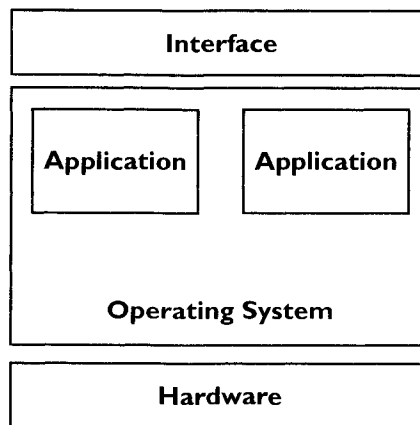
In 1984, the Macintosh presented developers with an environment like that shown in Figure 1-4.

The interface design and its implementation, which in the Figure 1-3 design were done by individual applications if they were done at all, are now basic parts of the system architecture. By not relying on individual applications to implement the interface, each application can become smaller (note the relative decrease in size from the previous diagram). Furthermore, since the interface itself is a system service, it should be more consistent: the notion of a single type of interface for the entire environment carries with it significant benefits for end users in the areas of training, software evaluation, and general usability.

Remember that every dividing line in Figure 1-4 represents the computer equivalent of a fault line in the Earth: pressures build up, and there are periodic cataclysmic shifts as one or the other side of the line shifts. Compared with Figure 1-1, there are far more fault lines in Figure 1-4. Changes in hard-

ware can affect the operating system; changes in the operating system can affect application programs; and changes in the application programs can affect the interfaces. And just to make life more interesting, remember that the changes can move in the other direction: advances in interface design can require changes in the operating system, which then requires changes to applications—and sometimes even to hardware.

FIGURE I-4. Operating System with a Graphical User Interface



Managing Changes

Changes to application programs come from two primary sources:

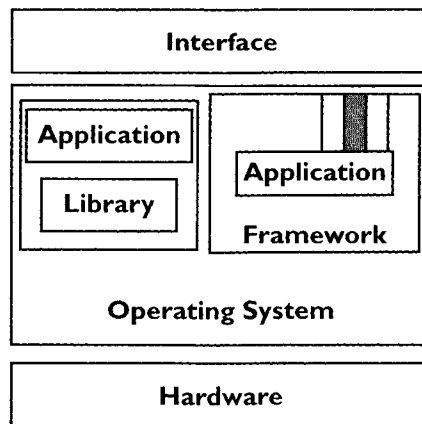
1. Many changes come from changes in user requirements or additions to the application's functionality.
2. A significant number of changes also come from external forces in the program's environment: changes in the operating system are one of the most common causes of application program changes.

Managing these changes has been a challenge for developers and managers for many years. Two strategies have been used repeatedly (and successfully) to address these issues:

1. Libraries of common code have been made available to applications. This lets separate programs use and reuse code that has been developed, tested, and documented for both programmers and users.
2. Object-oriented programming and reusable frameworks have allowed applications to be developed as more manageable collections of objects that can often be reused. More important, individual objects within an application can often be replaced with fewer side effects.

Figure 1-5 displays the applications from Figure 1-4, schematically showing the one on the left taking advantage of a reusable library and the one on the right taking advantage of an object-oriented framework.

FIGURE 1-5. Libraries and Object-Oriented Frameworks at Work



As is common throughout this saga, the application programs themselves provide greater functionality when they incorporate either a library or a framework; the specifically written code, however, is smaller. (This is not a necessary condition:

nothing prevents you from using a library or a framework and coming up with a less functional application than you could otherwise have done. However, this is the most common scenario.)

If you consider the boundaries between application, operating system, library, and framework together, there are many, many fault lines in this diagram. However, although they are theoretically more prone to disruption, the smaller size of each of the components suggests that these disruptions may be less severe than in the earlier diagrams. Certainly in Figure 1-1 (with simply hardware and an application) almost any change to the hardware is likely to cause some change in the software.

In the object-oriented world of a framework, there should exist individual components (such the gray shaded box in Figure 1-5) that should be relatively easily replaceable. Although a two-dimensional schematic suggests far fewer boundaries than actually exist, still it stands to reason that a change requiring a modification only to the shaded box should have a limited impact on the rest of the system.

This has been the popular theory of the past few years. It does make sense and people who have used frameworks and libraries (particularly shared libraries that can be used by several applications at run time) can testify to a very real sense that they are saving programming effort in the development and maintenance of their applications.

But the payoff isn't as big as it should be. Something's not quite right.

The Object-Oriented/ Flat World Boundary

Part of the problem with frameworks is that while they do an excellent job at shielding the custom-written application code from changes in the operating system and beyond, and although their object-oriented modularity interacts well with the object-oriented modularity of applications built using

them, the boundary between the world of the objects and the outside world (the traditional operating system) is a major obstacle. Most operating systems deal with procedure calls and data structures rather than objects. Frameworks, on the other hand, live in a world of objects. Every call across that boundary needs to move from one world to the other. And while there is a reasonable amount of control of changes within the object-oriented world of a framework and object-oriented applications, as soon as this boundary to the flat world is reached, the efficiencies are greatly reduced.

As a result, it is almost impossible for an application framework to keep pace even with one moderately sophisticated operating system and interface—the flat, native, non-object-oriented code of the operating system is updated before the frameworks.

In fact, it is quite rare for a framework to incorporate all of an operating system's application programming interface (API) in its classes and methods, much less to keep pace with changes. The task is enormous, and it is always faster in any individual case to code directly to changed platform APIs rather than to change the framework—or wait for a third-party vendor to do so.

The solution, of course, is to make the operating system object oriented itself. Then, there would be no need to synchronize API changes in the operating system's flat world with changes to an object-oriented framework: they would be one and the same.

The fly in this particular ointment, however, is that there is a certain degree of inefficiency in object-oriented systems. Normally this is not significant, but an operating system's tasks are often extremely time critical. This is a risk that few would be willing to take.

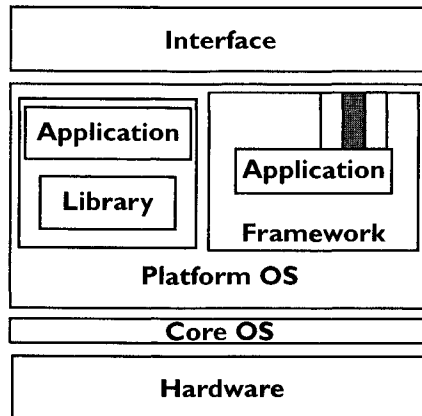
Splitting the Operating System

Meanwhile, the notion of a “modern OS” began to take shape in the minds of many. (See the next chapter for more details.) One of the notions behind a modern OS is that of a microkernel—a very small, fast section of code that does the direct hardware interaction; the rest of the OS is implemented as services above the level of the microkernel.

If you combine this notion with the issues in the previous section, you might see a way out—as did the people at NeXT. The operating system itself can be split into two parts: one can be the critical microkernel running as fast as it can, and the other can be an object-oriented OS.

Figure 1-6 presents a schematic of this stage of evolution: the splitting of the OS into two parts.

FIGURE 1-6. A Two-Part OS



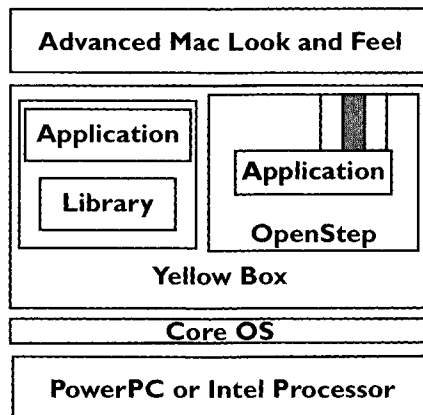
This is the basic design of Rhapsody. The core OS is Mach 4—a microkernel-based modern operating system that is not object oriented. The platform OS is thoroughly object oriented. The barrier between the object-oriented world and the flat world has moved down in the diagram: that barrier is now be-

tween the core OS and the platform OS. Within the platform OS, object-orientedness is the rule of the day.

The application on the left of the diagrams here contains a reusable library; it is assumed not to be object oriented and not to make use of a framework. Living in an object-oriented platform OS, it faces the same problems that the framework faced previously: its flat world, non-object-oriented data structures and procedure calls must be translated into object structures. This is not just a matter of turnabout being fair play. The object-oriented paradigm is being used increasingly—new languages such as Java are just adding to the momentum. Asking applications to learn to function in an object-oriented world is perfectly reasonable. (But, as you will see, Rhapsody does not require legacy applications to do so directly.)

Figure 1-7 is simply Figure 1-6 with Rhapsody names replacing the schematic names.

FIGURE 1-7. Rhapsody Architecture Diagram—Step One



As you can see from the label on the hardware box, this can run on both PowerPC and Intel processors. In both cases, the small core OS (the Mach 4 microkernel) interacts with the

hardware in question; the object-oriented and relatively hardware-independent Yellow Box runs on top of the core OS.

This is Rhapsody.

The Blue Box

This diagram is fine as far as it goes. From Apple's point of view, it leaves out a vital point: applications that run in the Yellow Box need to be compiled for the Yellow Box. Many, many Mac OS legacy applications are in regular use around the world. Requiring all of them to be recompiled for Rhapsody is unacceptable.

One of the things that an operating system is good at is running and scheduling tasks. The platform OS in the previous figure has run two applications—the library-based traditional application and the framework-based, object-oriented application.

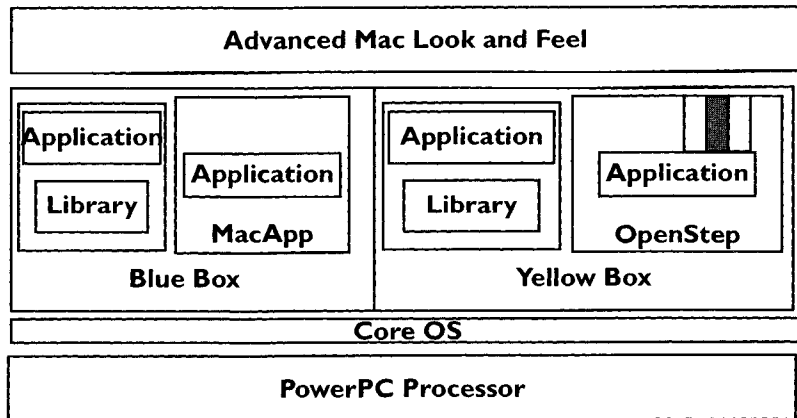
Running and scheduling tasks is in fact delegated to the core OS: the platform OS asks it to start tasks as needed. If this is the case, nothing prevents the core OS from running two platform OSs. The full Rhapsody implementation on PowerPC does precisely that: the core OS runs one platform OS called the Yellow Box and another one called the Blue Box.

Yellow Box is the fully object-oriented platform OS that has been described here. The boundary between objects and the flat world of data structures and functions is between Yellow Box and the core OS.

On the other hand, Blue Box *is* Mac OS. The boundary between objects and the flat world of data structure and functions is between object-oriented applications and those elements of the Blue Box (Mac OS) that the application touches. Applications that run in the object-oriented Yellow Box must be compiled for it, while those that run in the non-object-oriented Blue Box must be compiled for it—that is, they must be compiled for Mac OS (which is what all of those existing applications have been compiled for already).

And so, you get the full diagram of Rhapsody as shown in Figure 1-8: Advanced Mac Look and Feel, Blue and Yellow Boxes, Core OS, and PowerPC.

FIGURE 1-8. Full Rhapsody Diagram



This diagram includes suggestions of some applications that can run in the Blue and Yellow Boxes. The basic architecture (Advanced Mac Look and Feel, Blue Box, Yellow Box, Core OS, and hardware) is constant. What you choose to run—and write—is up to you. Here are suggestions for four typical applications (from left to right in Figure 1-8):

- A non-object-oriented Mac OS application that happens to use a shared library is shown on the left.
- Next, a Mac OS application built using the MacApp object-oriented framework is shown. Within the MacApp framework, everything is object oriented. However, the boundary between the world of objects and the flat world of nonobjects lies at the edge of the application. The Blue Box (Mac OS) is not object oriented.

- An application written for the Yellow Box is shown next. Such applications can use libraries; they can be written in Objective-C, C, C++, or Java. Full access to the Yellow Box API (which is dynamic and object oriented) is available only from Objective-C and from Java, but code that doesn't use the API directly (such as the vast amounts of legacy computational and application-specific code you have) can be written in any language.
- Finally, on the right you see a Yellow Box application written with the native Yellow Box framework, OpenStep. This is the preferred way of writing Yellow Box applications.

This full version of Rhapsody runs only on PowerPC. Remember the evolution of Rhapsody as summarized previously. The applications written for Mac OS were accustomed to dealing with a unified OS—not a microkernel and a platform OS. Some of them even touched the PowerPC hardware directly. There are dependencies on PowerPC throughout Mac OS applications; retrofitting these applications automatically to an environment that could run on Intel is not feasible. Thus, full Rhapsody—including the Mac OS/Blue Box—is only available on PowerPC. But that's OK, because only people with PowerPCs have been used to running Mac OS applications. They will run full Rhapsody as shown in Figure 1-8. New users of Rhapsody on the Intel platform—who have no legacy Mac OS applications—will run Rhapsody as shown in Figure 1-7.

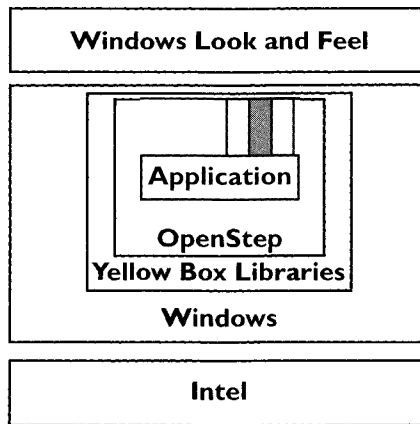
Windows

Owners of Intel processors do not have legacy Mac OS software, but they very well may have other legacy applications from the Windows world that they need to run. Theoretically, it is possible to create yet another platform OS that would run Windows, but that poses a number of logistical problems that start from the fact that whereas Apple owns and can use the Mac OS source code as it sees fit, it does not own the Windows source code.

So while you might think that the solution is to put a Windows platform OS on top of the core OS, that isn't feasible. Another solution is needed—one that builds on Windows itself.

Since the solution cannot be at the OS level, it must be at the application level—specifically at the level of individual Yellow Box applications. You can incorporate a very large amount of the Yellow Box functionality into an individual application and run it on Windows—a schematic view is shown in Figure 1-9.

FIGURE 1-9. Yellow Box Application on Windows



The same totally object-oriented application that uses Rhapsody's OpenStep framework can be recompiled to include Yellow Box libraries that will run on Windows. It is simply a matter of recompiling with different options. Obviously, the application is larger than it would have been if compiled directly for Rhapsody since it must include its own library of Yellow Box calls, but the size difference is not enormous. Furthermore, the Yellow Box library is shared (a dynamic linked library—DLL) on Windows, and if more than one Yellow Box application is running they can share the same library.

The application that is written here is the same in both environments: the OpenStep framework and Yellow Box libraries (if needed) provide the adjustments necessary to control whether you are running on Rhapsody with a core OS running on PowerPC or Intel or within a Windows environment.

Since this implementation rests on Windows the look and feel of the environment remain those of Windows—remember that the platform OS is what controls the look and feel, and Windows is both platform OS and core OS here (they are undifferentiated).

Also, many of the features of the Rhapsody core OS are unavailable in this configuration—the modern OS features such as symmetric multiprocessing, memory protection, multitasking, and multithreading may not be present at all or not be as fully or robustly implemented in Windows. For someone who values those features and who does not need to run legacy Windows applications, the Rhapsody configuration with its own core OS and the Yellow Box as a platform OS is the preferred choice.

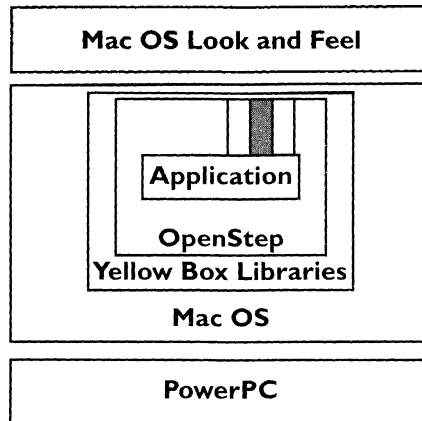
Rhapsody on Mac OS

In addition to running full Rhapsody on PowerPC or on Intel and running Yellow Box applications on Windows, a fourth alternative logically exists and is actually being implemented by Apple. Shown in Figure 1-10, it is the Mac OS version of the Figure 1-9 diagram. You can compile a Yellow Box application using compatibility libraries that bring Yellow Box features to a different platform—the Mac OS.

Just as with the Windows implementation, you can run legacy Mac OS code on the Mac OS—you still are running the same operating system those applications have always run on. Again, the new features of the modern OS and the core OS aren't available—it's not running here. However, your Yellow Box application runs just as it always does, with the Mac OS

compatibility libraries providing the bridge necessary to the Mac OS.

FIGURE 1-10. Yellow Box Application on Mac OS



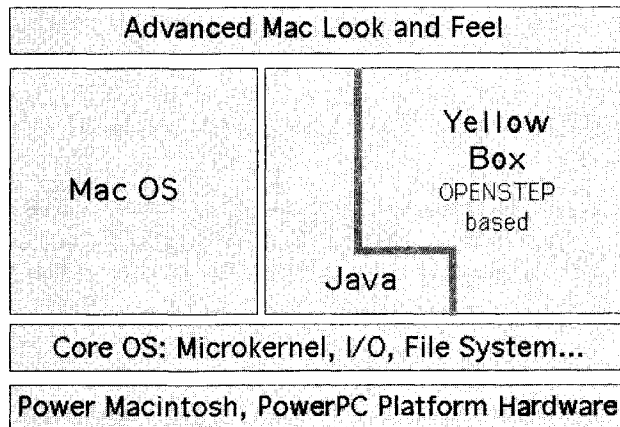
Whether to run Mac OS applications in the Blue Box of Rhapsody or to run Yellow Box applications with compatibility libraries on Mac OS is up to the user. The balance is likely to be tipped in the direction of where most of an individual's applications are. For some people, the increased reliability and performance of full Rhapsody will encourage them to move directly to Rhapsody and to run legacy software in the Blue Box. For other people, who want to try only a few Yellow Box applications without disrupting in any way the Mac OS look and feel, the choice of running Yellow Box applications with compatibility libraries will be preferred.

In any of these four scenarios (Rhapsody/PowerPC, Rhapsody/Intel, Yellow Box application/Windows, Yellow Box application/Mac OS), you as a developer have only to change your compiler options. It really is a matter of just recompiling.

Writing for Rhapsody

The schematic diagrams earlier in this chapter have focused on the evolution of Rhapsody and on how applications run on it. The Official Rhapsody Diagram (reproduced in color on the back cover of this book and in Figure 1-11) presents a slightly different view.

FIGURE 1-11. The Official Rhapsody Diagram



This diagram focuses on the development tools available on Rhapsody. Inside the Yellow Box, note the interlocking program interfaces to the Yellow Box API: the OpenStep framework and Java.

OpenStep is the object-oriented framework that you normally use in the Yellow Box. There are additional frameworks that are a part of Rhapsody—WebObjects and the Enterprise Object Framework. WebObjects allow you to create dynamic HTML using the same object-oriented technology that you use to create Yellow Box applications. Enterprise Objects allow you to move easily through the boundary between the

object-oriented world of the Yellow Box and the (usually) non-object-oriented world of standard databases.

As part of the developer's suite of Rhapsody tools, Apple provides Project Builder and Interface Builder—graphical tools to help you create and manage applications and to develop and maintain their interfaces.

Third-party tools (including the CodeWarrior Integrated Development Environment and Objective-C compiler as well as the Metrowerks Latitude compatibility library) are also available.

Summary

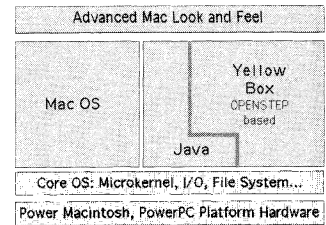
The remaining chapters in this part of the book look at the concepts of Rhapsody—modern operating systems and object-oriented programming—and at the languages of Rhapsody—Objective-C, Java, and C++.

The second part of this book presents the OpenStep API in some detail. The basic classes and their methods are all described here; these chapters supplement Apple's on-line and paper-based documentation. From that part of the book you should get a good sense of the architecture of the framework that you will be using.

Finally, the last part of the book looks at the evolution of Rhapsody—something that is happening very quickly and that you are likely to be a part of. Because Rhapsody builds on existing technologies from Apple and NeXT, there is a body of work that you can learn from and leverage in your applications.

Chapter 2

A Modern OS—The Mach Microkernel



The Museum of Modern Art in New York City suffers from a recurring identity crisis. What is it to hang on its walls, display in its foyers, and suspend from its ceilings? The apparently obvious answer—"modern art"—is of no help at all. "Modern" can mean "contemporary" or "of the moment," but it can also be used to denote a particular style or period—as in the word "mode," which has the same Latin root.

In the case of the museum, the pendulum has swung back and forth. At times, the staff and trustees have preferred the "contemporary" connotation, and they have deaccessioned some important nineteenth century works. At other times, the interpretation has leaned more toward the sense of a given period or style—encompassing art

by “ancients” such as Picasso, Bracque, van Gogh, Klee, Pollock, and Rothko while eschewing (or at least being uncomfortable with) contemporary movements such as Op-Art and the conceptualists.

As used to describe computer operating systems, “modern” is more commonly employed in the sense of a given style than in a specific historical sense. An operating system written last week is not necessarily modern, just as one written 20 years ago may be thoroughly modern. This chapter summarizes the common features of modern operating systems; it then reviews the general design and features of the Mach microkernel which is at the heart of Rhapsody.

There is not total agreement as to the exact components of a modern operating system: you may find some features that you consider decidedly un-modern here; you may also search in vain for characteristics that you consider essential. This discussion is intended to introduce the concepts—particularly those needed to understand Rhapsody. Other books address issues involving operating systems in much more detail than this single chapter can. Some of them can be found in your local library, bookstore, or on the Internet. The purpose of this chapter is to lay a common framework to discuss the Mach microkernel and then to do so. The theory of operating systems is beyond its scope.

The Story So Far...

Operating systems shouldn’t be of any concern to end users and should be of only minimal concern to developers and programmers. They should just go about their business, keeping the hardware running smoothly and trafficking among the needs of software.

Of course, the computer world is far from this happy picture. Operating systems do matter: they not only influence developers in what and how they develop but also impinge on users’ consciousness when they get in the way (often by misbehaving). Precisely because operating systems are not yet highly developed enough to be invisible (or at least trans-

parent) to developers and users, it does matter what is down there gurgling at the bottom of your computer.

By the time of the first personal computers, computer operating systems were highly developed and quite sophisticated. They were able to manage computers with enormous banks of disks, endless rows of tape drives, large amounts of memory (at least for those days), and many simultaneous (or almost simultaneous) tasks running under the control of scores of users—many of whom sat at remote terminals connected via phone lines from distant locations.

Personal Computer Operating Systems

Precious little of this sophistication in operating systems carried over into the early personal computer operating systems. Personal computers, of course, were an inherently different type of computer from mainframe time-sharing computers. Personal computers had no banks of disks—two diskette drives were a luxury; they had small amounts of memory; they ran single tasks under the control of one person—hence the term “personal computer.” Their operating systems, therefore, had different characteristics, and they were implemented to fulfill the needs of the single-processing, unnetworked, desktop world.

As happens with so many things (particularly the computer industry), time brought change and improvement. The desktop operating systems evolved just as the computers they were running on evolved and grew. Today, the desktop personal computer is likely to be more powerful than the mainframe of yore, with more memory, more disk storage, and even faster processors. Like the mainframe of yore, however—and unlike the personal computer of yore—it is likely to have several processors, be called upon to carry out several tasks at once, and communicate over a telecommunications network that is more complex than the mainframe-centric network of two decades ago.

Sharing

Today's personal computing environment may be more familiar to the 1980s corporate mainframe programmer than to a colleague who has grown up totally in the personal computer world. One characteristic of today's environment (as of the time-sharing environment in the past) is that the computer resources are shared among many processes and potentially among many users.

When the computer and its resources are shared, its problems are also shared. There are times when the computer crashes or finds itself wandering along some dark and dangerous by-way. When the computer is not shared, restarting it means that your unsaved spreadsheet may be lost or that your Internet connection may be broken. In a shared mainframe environment, many users' spreadsheets, transactions, databases, and calculations may be lost; if the computer is serving as a telecommunications hub (as many mainframes did and do), hundreds if not thousands of network connections may be broken—and with them scores more spreadsheets, transactions, databases, and calculations may be lost. Restarting a shared computer is a Very Big Deal.

You can appreciate this when you read on the front page of your local paper that an Internet access provider's system has crashed. Programmers from the mainframe era can also recall the consequences of computer crashes: meetings, retributions, memos, and even dismissals. Restarting a mainframe computer was not taken lightly.

On a personal computer where many programs may be running, restarting the computer after it crashes or hangs is equally disturbing. A user may easily not know what was running, and an unsaved spreadsheet may not be noticed for weeks if not months.

More than anything else, modern operating systems address the issues involved in sharing a computer's resources among many users and many programs. Almost everything that makes an operating system "modern"—at least in the formu-

lation given here—can be seen to stem from these needs. As personal computers get bigger and faster, as people demand more and more functionality and responsiveness, and as communications among computers become more intricate, the need for a modern operating system becomes ever greater.

Some programmers remain reluctant to give up their proprietary ownership and management of system resources. You will find many cases where developers have grabbed the event loop to “optimize” part of their code; writing to memory directly, patching traps on the Mac OS, and a myriad other examples show how easy it is to justify the “need” to avoid sharing critical resources at certain times. To the programmer who is reluctant to give up this control, remember that you are giving this control up not to another programmer, but to the user. Sometimes that makes a seemingly bitter pill easier to swallow.

Modern Operating Systems

Modern operating systems don’t start (or end) with UNIX, but UNIX certainly is on the scene of a lot of modern operating systems. Virtually all modern operating systems have their roots in the great time-sharing operating systems developed or perfected in the 1970s: IBM’s System/360, the Burroughs 5000 mainframe series (and successors), Digital’s VAX computers, and UNIX.

The amount of raw computing power devoted to the operating system on those computers was relatively large. A heavily loaded mainframe computer could easily use over half of its processing power in running its operating system.

In those days, sharing computer resources was necessary because they were so expensive and scarce. The advent of personal computers made these resources relatively inexpensive and plentiful, so the early personal computer operating sys-

tems didn't need to worry about the sharing aspects of their predecessors. Before long, however, sharing once again became necessary as users' demands for processing power made even these plentiful resources once again relatively scarce. (Why shouldn't you be able to surf the Net, move files back and forth from a local server, print out a 50-page report, and check your calendar all at the same time?)

You can view the role of an operating system in a shared environment as being that of an agent of transition: almost any programmer of modest skills can develop a simple operating system that runs one program at a time and manages a limited amount of memory (like 640K). The operating system that shares processing power and other resources may look like this at any given moment, but in fact its power and innovation must lie in its management of the transitions between allowing one program access to its resources at one moment and allowing another program access to those resources at another time.

Thus, the modern operating system needs to provide the basics of all operating systems (access to hardware and other system resources) as well as to provide tools for managing transitions that involve the sharing of:

- processing power
- memory
- communications channels
- disks and other storage devices.

The sharing of these resources is done in the simplest way, although it is relatively expensive in terms of computer power. Rather than placing a burden on users, operators, or programmers to determine what programs can fit together, the computer's operating system gives each program total access to all resources for finite periods of time. Each program (and user) can then function as if it were running with no other programs or users, and the resources are shared by being

allocated *in toto* to a constantly changing collection of programs and users.¹

The maid to a touring tragedian playing one-night stands could turn hotel room and dressing room into “homes” in five minutes—and disassemble them equally rapidly leaving nothing but the shell of the room while all the makeup, mementos, clothes, hot plates, and ashtrays found their places in the nooks, crannies, drawers, compartments and shelves of the touring trunk. The star entered hotel rooms and dressing rooms that were always fully furnished and looked as if they had been that way for years. This is basically what an operating system does as it shares its resources. The operating system is responsible for the packing and unpacking—of not one but every process that is running; the programs, like the vaudevillians of the past, never see anything but the complete environment.

In order to do this, a modern operating system needs to implement functionality in the following areas:

- User-level security must be provided so that data from one program or user can be identified and restored to its rightful owner; in addition, the operating system generally plays favorites. Once it can identify users (and classes of users) reliably, it can modify its rotation of resources to favor certain ones. Thus, scheduling is intimately connected to security in a modern operating system.
- Process management is needed to enable a process to be encapsulated in some manner, stored temporarily, and then restored to the appropriate registers and

1. Some operating systems—particularly on well-endowed mainframes—actually support a combination of these techniques. The total pool of resources is subdivided into partitions with programs and users being assigned to specified partitions within which the rotational sharing is carried out.

storage locations when it is next given the opportunity to run.

- Memory management must be able to handle the relatively large amounts of memory that the system and process management may require to be in turn allocated and then temporarily stored on less valuable devices than memory (such as virtual memory disks).
- Some communication mechanism is needed between and among processes that are “running” on the same computer but that may be temporarily inactive and not in memory.
- Exception and fault handling techniques complete the basic list of modern operating system essentials by limiting damage to the smallest possible area so that shared resources are unaffected by a single process’s failure.

As noted previously, there are many other services that any operating system must perform. These are the most critical ones that a modern operating system must perform—usually over and above the services found in other types of operating systems. Experience (often painful!) has shown that these features must be built into an operating system at its most basic level: attempts to retrofit existing operating systems have proven costly failures.

If you read some of the accounts of the development of modern operating systems—or if you have been there as the process unfolded—you will note that these five areas of functionality did not always assume such importance. Their importance today is based on hindsight and on the differences that they strike with traditional personal computer operating systems.

Security

It is hard to overstress the importance of security in making a time-sharing system work. Since all operating systems are responsible for managing input and output to and from files, terminals, and other devices, they in fact can intercept every

single byte or character that is read or written. Moreover, since a time-sharing operating system must periodically pack up each program's entire operating environment—including values stored in memory that are never read or written to disk—it has access to absolutely everything that a program touches.

Thus, a modern operating system has a concept of security that involves individual users with passwords as well as at least a superuser with access to everything. (There may be intermediate levels of privileged users with increasing access.) Implementing security on personal computers has always been a problem since it has been done as an add-on. The fact that you can boot most computers from a diskette means that almost all security mechanisms can be bypassed if you have the appropriate diskette and access to the drive itself.

The ability to know who is using the computer is important for more than just security reasons: many computers are shared in classrooms, offices, and homes. As users are increasingly able to customize their computing environments, it becomes essential to be able to tell who the current user is so that the appropriate interface can be presented. Without the concept of logging into a computer (and its parallel notion of the ability to automatically log out a user after a specified period of inactivity), no practical security exists. A glimpse at the monitors of computers in almost any computer store will demonstrate the problems involved in not providing log in and automatic log out mechanisms.

Why It Matters Security is important not just in order to identify the current user of the computer, but in order to prepare the computer (and its current user) for access to networked resources.

Process Management

Process management is the heart of a modern operating system—it is the core of the packing and unpacking referred to previously. A computer program that is running on a modern

operating system should not be aware that it is doing anything but running alone on a single-user computer.

In Mac OS (as in many operating systems), programs can be suspended whenever they temporarily pause—usually by calling `WaitNextEvent` or `GetNextEvent` in their main event loop. At such a moment, the operating system is able to service other processes. However, if a program does not call `WaitNextEvent` or `GetNextEvent`, the operating system never has a chance to intervene. Thus, lengthy procedures in which the program does not need to poll for mouse, keyboard, or menu events can grab hold of the processor and not let it go—sometimes for very long periods of time. Examples of such programs are disk utilities that may be checking every sector of a hard disk for quite some time, installers and compression/decompression utilities, and major calculation beasts. The damage (that is, the problem of not letting other applications use the processor) can be mitigated by not letting other applications run at the same time, and in fact many installers and some disk utilities enforce this before starting their work.

Furthermore, the Mac OS supports only multithreading rather than full-fledged multitasking. In a multithreaded environment, memory is not moved and the individual threads have access at times to other threads' data. (Threads are sometimes processes but may be smaller units of a single process that legitimately should share data.)

Thus, on traditional operating systems, the sharing of the processor requires that programs be written in certain ways (by calling `WaitNextEvent` or `GetNextEvent` periodically so that the operating system can intervene). In a modern operating system, this strategy—cooperative multitasking—is replaced by symmetrical multitasking in which the operating system is not thwarted in its sharing mission by the way in which a program is written. The operating system is capable of starting and stopping any running process at any moment, moving its entire environment somewhere, and restoring and restarting it at will.

Nomenclature In this chapter, “program,” “application,” and “process” have loosely been used interchangeably. From here on, the specific term “process” is used to mean a single executable software environment with its own code, its registers, its memory, and its other resources (files, communications channels, etc.). A simple program is always at least one process.

Later in this chapter, “process” will be redefined and narrowed to mean a Mach task that contains one or more threads. At this stage, “process” is used in the more general sense that applies to UNIX and most other operating systems.

“Multitasking” and “multiprocessing” are used interchangeably at this point; multitasking is the preferred term.

Why It Matters Obviously, multitasking allows a computer’s resources to be shared among more tasks than one. If all tasks were busy at all times, there would be little if any saving. However, since most tasks periodically wait on external events (keyboard input, network or disk communications, etc.), that time can be given to other tasks.

Much more important, however, is the fact that the mechanism by which a task can be stopped, packed up, restored, and then restarted can be used for more than just sharing a single set of resources. Nothing says that the task has to be restored to the same environment from which it was removed (although the environment must be quite similar). In a computer with multiple processors and significant amounts of memory, the task can be restored to a different processor and to a different area of memory—provided that its internal pointers are undisturbed. This provides additional opportunities for optimization and for using increasingly powerful computers.

Memory Management

The single-process computer (whether an old batch-mode mainframe or a traditional personal computer) allows the process that is running to access memory at will. The advent of virtual memory allowed programs to access more memory than was physically available with the operating system using disk to store the additional memory. However, even under virtual memory, the process has access to all of the computer's memory—both real and virtual.

Some programs took advantage of this. Quite a few early Mac OS programs (as well as those for other operating systems) relied on the fact that certain memory addresses contained certain operating system variables—screen size, for example. Programmers who needed to get this information simply read it from the appropriate memory location despite the fact that that memory location logically “belonged” to the operating system, not to the application program.

Modern memory management involves several features, the two most important of which are virtual memory and memory protection. Virtual memory is the same technique that has been around for some time now (although it is continuously enhanced); memory protection is the ability to divide the computer's memory into areas that belong to certain processes (or to the operating system) and to prevent other processes from accessing those memory locations.

Why It Matters Virtual memory is used to support large memory demands of programs and of the operating system. Memory protection obviously increases the stability of the entire computer since a single process cannot inadvertently or otherwise access other processes' memory locations.

Communication Between and Among Processes

In many traditional personal computer operating systems, applications can read and write from and to memory locations that are owned by other applications. Some programs in fact take advantage of this in order to communicate with one another. Clearly this approach is risky, since changes in memory

layouts for one application program can cause another to behave erratically.

But it is necessary for processes to communicate with one another from time to time. For example, if a single process is divided into multiple subprocesses (threads) to take advantage of a system's multiprocessing capabilities, the threads need to be able to share data, status, etc.—particularly if the process is going to use any mechanism at all to allow semi-concurrent access to data. For threads within a single process, the solution is to access shared data locations which legitimately are visible to all threads. For processes that do not share memory locations—and which are prevented from looking into one another's memory locations by a memory protection mechanism—it is necessary for them to be able to communicate by sending messages to one another.

A modern messaging system allows processes to find one another, identify one another, and to acknowledge, receive, and send messages according to their own internal formats.

Why It Matters In addition to providing solutions to the problems posed in this section, modern interprocess messages lay the groundwork for messaging between and among processes on different computers (e.g., over the Internet).

Failure and Exception Handling

Failure and exception handling returns full-circle to the issue of security which started this section on modern operating system attributes. Like security, robust failure and exception handling makes everything else possible. Having memory protection is all well and good, but if a process attempts to access another process's memory, is rebuffed, and then causes the operating system to crash, you are no better off than if it had been able to access the off-limits memory, proceeded to corrupt it, and caused a crash at some point in the future.

Modern operating systems manage failure and exceptions both within their processes and for themselves. They should not crash (as indeed no operating system should).

Why It Matters In order to deliver an operating system as immune as possible from its own failures, modern operating system architectures take advantage of all of these features in a design called a microkernel.

Microkernel Architecture

When you have an operating system that can run multiple processes at more or less the same time, where the processes can have their own memory spaces, and where they can communicate reliably among one another, you can take many system services and run them as separate processes. Modern operating system architecture splits out functions such as file management, sophisticated message processing, and user access into separate processes which can run either as necessary or as permanently resident processes that respond to messages as needed.

In such an architecture, the operating system is first divided into separate entities that can run in this manner. What is left is the microkernel—that part of the operating system that is needed to enable the others to run. Typically, the microkernel provides process and memory management as well as basic communication and input/output services. Everything else can be run as processes taking advantage of these tools.

Most operating systems themselves are allowed to play by different rules than applications or user processes. (Operating systems, for example, are not restricted from reading or writing memory since they must manage virtual memory and enforce memory protection.) The microkernel is no exception to this rule.

The other parts of the operating system, however, can usually safely run as processes in user space (as opposed to the few-er-holds-barred kernel space). By placing these processes in user

space with its restrictions, they are less likely to cause system-wide problems. The strategy is as simple as that of the bank president's giving the vault combination only to the branch manager rather than to all employees of the branch. In both cases, customers would not know the vault combination; however, by giving only the manager the combination, the bank is reasonably well protected against employee misbehavior and need worry only about the manager.

Mach

The preceding part of this chapter outlined the idea of a modern operating system in general terms. It is safe to say that no actual operating system exactly implements all of those concepts. Mach—the microkernel that is at the heart of Rhapsody—implements most of them.

Mach moves beyond some of the general principles, refining them and enhancing them to provide even more flexible, robust, and efficient performance.

Tasks and Threads

The concept of processes (described on page 49) is refined and extended in Mach. The modern operating system's process is split into a single task and one or more threads within that task. Tasks do not run: their threads run, with each thread containing its own execution state. The task owns the common resources which are available to all threads, including a single address space in virtual memory. Registers, stack pointers, program counters, and other elements that control and record the execution state of a thread belong to the individual threads of a task.

Mach's design builds in multiprocessor support from the start; threads can run on different processors more or less simultaneously: Mach contains the mechanisms to provide synchronism and to prevent threads from running ahead of or

behind others. The design also allows for control (even down to the task level) of which processor specific threads will run on and what area of memory will be used for the task's memory.

As increasingly powerful and numerous processors show up in computers, these capabilities become more and more important.

Memory Management

In addition to Mach's support for sophisticated thread management, it contains a very flexible memory management model based on memory objects. The management of logical memory (as opposed to physical memory) is designed to run in user space—not in the kernel space. By running in user space, the virtual memory module does not have access to system-level calls and is therefore more robust and less likely to have adverse affects on anything else that might be running.

As a developer, you are unlikely to worry about memory management in Mach; its architecture allows for sharing of memory objects among tasks and indeed for sophisticated distribution of memory resources. These capabilities are among those that are difficult to retrofit, and as you find yourself needing them it is reassuring to know that they are there.

Communications Between and Among Tasks

As software evolves away from the single-processor/single-process model, communications between and among tasks on individual computers and across networks become increasingly important. It is hard to overemphasize the value of powerful communication technologies and protocols (the Internet being perhaps the best example to date). In fact, many would argue that the greatest benefits that have been derived from computers in the last 50 years have been due to their networking and communications technologies rather than from their computational technologies.

Communications in Mach are managed by ports and messages. You may be familiar with UNIX pipes, which are byte streams that can link two processes together; the Mach mechanism supports message streams, not pipe streams. As with the task/thread structure and the memory structure, Mach provides a more structured environment than some other modern operating systems.

Ports are created inside the microkernel; they are owned by individual tasks and, like all task resources, are available to all of the threads within that task. A thread can communicate with a thread in another task through the port that they both can see. Just as with process (task/thread) management and memory management, the communications infrastructure is designed to work in an environment of multiple processors and equally well within a single computer or over a network.

Mach and Other Operating Systems

The Mach microkernel is designed to manage multiprocessor and distributed operations. Wherever possible, operating system functions have been moved into tasks that run in user space (rather than in the fewer-holds-barred kernel space). This provides increased stability for the system.

Among user space tasks, Mach has always been planned to be able to support UNIX, MS-DOS, and other operating systems running as user tasks. This ability to be able to run a number of operating systems on top of the kernel was one of Mach's original goals.

You can see the importance of this architecture by referring to the Diagram of Rhapsody which is found in color on the back cover of this book and in gray-scale images at the front of each chapter. Notice that both Mac OS and OpenStep run directly on top of the microkernel. This isn't a kludge or a compromise: this is a use of a major part of Mach's original design.

Summary

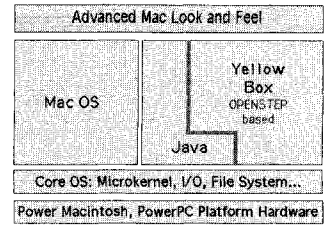
With all the commotion over the importance of a modern operating system, sometimes the jargon gets overpowering and developers (much less users) are to be forgiven if they react to these discussions with a modern attack of the vapors.

At its heart, the notion of a modern operating system isn't some esoteric litmus test as to whether feature A is or is not supported. The ideas behind modern operating systems fall into two major categories: ways to improve performance, stability, and reliability of today's computers; and ways to be positioned to exploit the computers of tomorrow which, with multiple processors and persistent network connections (wired and otherwise), offer opportunities and challenges never seen before.

With OpenStep in the Yellow Box and Mac OS in the Blue Box you need never concern yourself with the details of Mach and Rhapsody's microkernel. This chapter has given an overview of what is there and should reassure you that the groundwork has been laid for building software for quite some time.

Chapter 3

Object-Oriented Programming



A firm foundation on a modern operating system is not enough to make most people confident about putting their development (and production) money into a programming environment. Half a century of software development experience has shown that programs last much longer than anyone ever predicted. (Much of the problem with the year 2000 comes from software written in the 1960s and 1970s that is still in regular use today. When you consider that few automobiles from that time are still on the road, the longevity of software is alternately awe-inspiring and depressing.)

Object-oriented programming is at the heart of Rhapsody. OpenStep itself is an object-oriented framework; MacApp, Apple's workhorse framework, is a comfortable fixture in the Blue Box (Mac OS).

OpenStep offers the developer a choice of three object-oriented languages: Objective-C, Java, and C++. This chapter provides a very brief overview of object-oriented programming from a language-neutral point of view. It highlights some of the implementational issues that need to be addressed and which the three languages address in different ways.

Object-Oriented Programming

Object-oriented programming is one of the technologies that has been proposed—and adopted—in order to make the process of creating software easier, faster, and more efficient. (The other major technologies that have been proposed and adopted to these ends are structured programming and databases.)

In contrast, some technologies have been proposed and *not* adopted widely. Why have CASE (computer aided software engineering) tools not been more successful despite a relentless flood of promotion? Why have code generators had limited acceptance? And perhaps the biggest question of all is whatever happened to the 1980s grand design methodologies that promised to take only a few years to completely model an organization's data needs (during which time, of course, nothing would change).

The difference is quite clear, but you must promise not to tell anyone (because some people might get a little upset). The tools and techniques that have succeeded are those that programmers have perceived as making their lives easier. The tools and techniques that have failed have been those that management embraced and that programmers didn't particularly understand, appreciate, enjoy, or use.

There is a little-noted measure of a technology's viability in the programming world: weekend projects. Most programmers love to program and to design software. Some augment

their working lives with volunteer work for community organizations; others steal away on evenings and weekends to come up with a shareware or freeware product—or even to work on what just might be the next “killer app.”

In these projects, structured programming, databases, and object-oriented programming abound.

To be sure, object-oriented programming and the other major technologies are found in the largest corporations, and the most horrible unstructured spaghetti code messes can also be found in private skunkworks projects. But by and large, these are the tools that programmers use when they’re working for the most demanding (yet understanding) boss of all—theyself.

Why Is It Liked?

Despite a steep learning curve, object-oriented programming is easier than traditional procedural programming. An application is divided into objects which have the characteristics described further in this chapter but which have one critically important feature: they can be totally self-contained.

Totally self-contained entities can be combined into various combinations in ways that can produce larger and complex systems that would be prohibitively expensive to produce on their own. (An example from the 1960s was a combination clothes washer and dryer. Combining the two products into a single cabinet seemed like an efficiency; in practice, the two products constantly managed to interfere with one another and the combination product ended up saving floor space at the expense of being constantly in need of repair.)

From a practical point of view in the world of object-oriented programming, an object can be left untouched once it has been developed. Its functionality and interfaces can be frozen, and it can be used without worrying about its internals. Procedural languages—even with shared libraries and utility routines—rarely can provide this degree of isolation.

Of course, these write-once-use-many objects need to be carefully designed. You can do many things to break down the ramparts of self-containment (you can use globals, refer to other objects about which you make certain assumptions, etc.). Nevertheless, many object-oriented programmers have a bag of objects that they drag along from project to project. Each object can be reused because it is self-contained and adheres to strict programming standards (the programmer's internalized set of standards). The reality of object-oriented programming leans toward programmers saving a day here and a day there by reusing objects that they have written or used before. The enthusiasm for object-oriented programming comes from programmers having experienced this process and being able to testify that it does indeed work.

Where Are the Benefits?

What has remained elusive, however, is the large-scale payoff for using object-oriented programming on large systems. On a daily basis, programmers experience firsthand the benefits of object-oriented design as they reuse objects. There is also an almost tangible feeling that the objects that are written anew will be easier to maintain in place (even if they are never reused).

The benefits of object-oriented programming in large systems depend on the ability to integrate objects written by different people with vastly different programming standards as well as on the ability to combine these objects dynamically.

Using Other People's Objects Despite the fact that an object designed for reuse can (and should) be designed to make as few assumptions as possible about its environment, many do make quite a few assumptions. There is not a significant market in off-the-shelf objects because they often don't fit together well. Even the recent enthusiasm over Java applets has not been able to cover the fact that most Java applets are reusable only in the contexts for which they have been designed. (Of course, for those applets that have been designed as buttons

or other controls on Web pages, that context is so widely available that some reuse is possible.)

Developers and designers pay a great deal of attention to the inside of objects, but the big payoff in reusability comes when attention is paid to the outside of the objects—that is, to their interfaces and to their functional scope. And that is one of the most important elements of OpenStep.

Run-Time Object Combinations When it comes to actually combining objects, the details often scuttle the ship. Unless some binary standard is available, objects in different languages and from different compilers will likely be unable to be used together. Standards such as CORBA (Common Object Request Broker Architecture) are available, but in a classic chicken and egg syndrome, few developers see the need to adhere to these standards since no one is combining objects dynamically across machines—and no one is bothering to try combining such objects because the standards are not widely supported.

Part of the attraction of Java is its promise to simplify this combination of objects at run time across networks, but the crunch is only now coming where it will be seen whether the full power that can be delivered is demanded by consumers.

Thus, object-oriented programming is liked by programmers, is supported anecdotally as saving development and maintenance time, and has at least the potential to provide significant savings in large system development if some standards-related issues can be solved.

The Learning Curve

Many people find object-oriented programming hard to learn; for others, it's not at all complicated. One of the factors that contributes to this discrepancy is the fact that some programmers can be comfortable only if they understand the entire program on which they are working. Others are comfortable

working on a part of a program and relying on others to do the rest.

For programmers who aren't comfortable until they see the big picture, object-oriented programming may be difficult to accept. There is always a faster and more efficient way to do something procedurally—but it is a way that breaks the object-oriented design and that requires that all future programmers also have the big-picture outlook.

Unfortunately, many computer science courses and programs emphasize the big-picture outlook and encourage students to develop large applications on their own. The fact that they are training their students to solve yesterday's problems using yesterday's technologies should not surprise those who themselves were taught to solve day-before-yesterday's problems using day-before-yesterday's technologies.

If you're not interested in learning new technologies and rethinking the way you work, object-oriented programming may not be for you. The demand for programmers is so enormous that there is room for everyone. There's plenty of Cobol code that needs maintenance, and there's a whole bunch of PL/1 applications that need to be converted into something—anything—else.

Objects

What, then, is object-oriented programming, in a language-neutral sense? This section is a brief overview.

Objects are entities that in some ways are like small programs. They have their own data and their own functionality—just as a *solitaire* or *payroll* program does.

Data An object can contain data of any kind—integers, floating point numbers, arrays, strings, etc. The data can even consist of other objects. In the strictest (and best) object-oriented programming, an object never presents its data directly to the world. You always have to call one of its functions or methods to get the data.

In other words, even if the object has a data field called “time started,” you access it by calling a routine like “GetTimeStarted” rather than accessing the variable directly. This is good programming practice even in non-object-oriented programs.

Data fields may be called fields or values; they may also be called member values. Some objects have no data fields of their own.

Functionality Alongside its data, an object has routines—methods or functions—that it can run. You can call these routines just as you would call routines (methods, functions, sub-routines, or procedures) in any programming language. You may also invoke them by sending messages to the object that cause the appropriate routines to be run.

The distinctions between methods and functions as well as those between calling routines and sending messages that invoke them are semantic in the various object-oriented programming languages.

Instances Each specific object that is created is called an instance. For example, you may have 500 instances of the employee object—one instance corresponding to each employee in a company.

Each instance shares the names of the data fields and routines of the object itself, but each instance has its own data values.

Classes The most common usage holds that the class is the abstract object which is instantiated in instances. The relationship between classes and noninstantiated objects is different in the different object-oriented programming languages (but is quite close).

Programmers often talk interchangeably about instances and objects (when they mean an object that has been created and has its own data) as well as interchangeably about classes and objects (when they mean the abstract, noninstantiated thing).

This imprecision is a minor point since the context is almost always clear.

Object-Oriented Design Issues

You define an object with a class definition that specifies its variables and methods; you create a specific instantiation of that object to contain a single set of data.

In a nutshell, that's it. However, there are a few design issues that are important in all object-oriented programming languages and that are particularly relevant to OpenStep and Rhapsody.

Inheritance

An object may be based on another object—may in fact “descend” from that other object. Typically, the descendant inherits all of the data fields and member functions of its ancestor. Thus if an employee object has an employee ID field, a descendant of the employee object (hourly employee) automatically has an employee ID field just as another descendant (salaried employee) will also have an employee ID field.

Any instance of the employee object, the hourly employee object, or the salaried employee object may use the employee ID field (subject to some language and scoping constraints).

Similarly, if the employee object has a method called `CalculatePayCheck`, each of the descendant objects automatically has access to that method and may call it (again, subject to some language and scoping constraints).

Polymorphism

An object may reimplement any method that it inherits from an ancestor. Thus, the salaried employee object may reimplement `CalculatePayCheck` (the method of employee) to return a value equal to the annual salary divided by 52 (in the case of weekly pay).

The hourly employee object may reimplement `CalculatePayCheck` to return a value equal to the product of its own data fields `hourly rate` and `hours worked`.

In fact, the `CalculatePayCheck` method of the employee object may do nothing whatsoever, relying on its descendants to perform the appropriate calculations.

In such a scenario, you can instantiate objects of hourly employee and salaried employee as necessary. Since both are descendants of the employee class, they both have a `CalculatePayCheck` method (although their methods are different). You can therefore call `CalculatePayCheck` and rely on each object's carrying out the correct behavior.

This is polymorphism—the ability of disparate objects to respond appropriately to messages or function calls that call a common ancestor's routine.

Further, this scenario demonstrates the use of an abstract superclass. The employee object may be designed never to be instantiated: only its descendants are actually to be instantiated. Such an abstract superclass contains common routines that are either stubs (where the subclasses provide their own polymorphic implementations) or common routines that need no overrides in the subclasses.

As you can see from this discussion, the terms *ancestor* and *descendant* are used interchangeably with *superclass* and *subclass*.

Data Hiding and Encapsulation

An object can contain data in its instance variables. This data is available subject to certain scoping and language restrictions as previously noted. The most basic restriction is in the class definition: you may declare variables as being *public*, *private*, or *protected*. Public variables are visible to anyone who uses the object instance and may be referred to directly according to the appropriate language syntax. Public vari-

ables are second only to global variables in their inappropriateness for use in good code.

Private variables are visible only to instances of this particular class. Thus a method of class X may refer to a variable of class X, but no other method of any other object may do so. Private variables are as good as public variables are bad.

Also good are protected variables: these may be referenced by methods of their own class as well as by classes which descend from it.

Accessors Typically, variables are referenced by accessors—functions whose duty is simply to return a value, as in the following code:

```
short GetEmployeeID (void)
{
    return employeeID;
}
```

Accessors typically come in pairs; the complement to the code above would set the value:

```
void SetEmployeeID (short theNewID)
{
    employeeID = theNewID;
}
```

Accessors generically describe the types of routines shown here; you may refer to them more specifically as getters and putters, but that is poor English.

Accessors are a good idea in all code—even traditional procedural programming.

What It Means—No If Statements

At a fairly high level, object-oriented programming is a way for programmers to be more productive; they can feel more in control of their code because it is broken up into sturdy chunks; and the opportunities for reuse—not fully realized yet—are very attractive.

At a low level, object-oriented programming means the end of if statements and their switch/case statement cousins. In traditional procedural programming, the employee example used previously would probably have been written in this fashion:

```
fixed CalculatePayCheck (employeeID)
{
    short employeeType;
    employeeType = GetEmployeeTypeFromEmployeeID (employeeID);

    switch employeeType {
        case 1: //salaried employee
            ...
            break;

        case 2: //hourly employee
            ...
            break;

        ...
    }
```

Or, if only two cases can occur, it could be written as follows:

```
if (employeeType == 1)
    ...
else if (employeeType ==2)
    ...
```

Code with conditions (if statements and switches) is very hard to debug. The number of possible execution paths quickly becomes enormous. In object-oriented programming, the conditions are normally moved outside the objects: a specific type of employee object is created and within its processing (salaried or hourly) everything proceeds in a clear and straightforward manner with very few branches.

The benefits of sequential processing are enormous—particularly in debugging and maintaining code.

An Example

Accessors easily allow you to add all types of filters to data. For example, if an accessor returns the coordinate of a point at which to draw an interface element, the accessor can take into

account a scale factor for the view in which it is drawn and adjust the point accordingly.

Here are a few examples of how that could be done, following on from the accessors discussed previously; together they give you an idea of some object-oriented design issues.

These examples should all be considered pseudocode.

```
point GetLocationWithScaleFactor (real theScaleFactor)
{
    return instancePoint * theScaleFactor;
}
```

This is fairly straightforward, but it is quite specific. What if your interface allows you not only to scale the view but also to relocate it within a larger context? You could create a `GetLocationWithScaleFactorAndOffset` method and allow either the scale factor or the offset to be zero, but the need to do this would surely alert you to the possibility that other changes may come down the road.

A more general way is to construct a new object which can encapsulate all possible adjustments. The new object could be called something like `CoordinateAdjuster`; its instance variables would include a scale factor, an offset, and any other possible modifiers. The accessor then becomes:

```
point GetLocation (CoordinateAdjuster* the Adjuster)
{
    point localPoint = instancePoint;

    localPoint = localPoint * theAdjuster.GetScaleFactor ();
    localPoint = localPoint + theAdjuster.GetOffset ();

    return localPoint;
}
```

A change to the imaging algorithms would entail only a modification to the `CoordinateAdjuster` object (addition or deletion of an instance variable and accessors) and a possible

modification to the `GetLocation` code. No calls to `GetLocation` would have to change and anyone who uses it would likely be shielded from the changes.

A final structure provides an even more sophisticated methodology:

```
point GetLocation (View* theView)
{
    point localPoint = instancePoint;

    localPoint = theView.ConvertPointToDisplay (localPoint);

    return localPoint;
}
```

The `ConvertPointToDisplay` method of the view object might look something like this:

```
point View::ConvertPointToDisplay (point thePoint)
{
    CoordinateAdjuster*the Adjuster =
        GetCoordinateAdjuster ();
    return theAdjuster.AdjustPoint (thePoint);
}
```

In this final version, you can see a number of object-oriented programming techniques and issues coming into play. The most interesting one is the decision as to where to place variables and functionality. In the first case, the scale factor and offset were simply passed in as parameters. With the creation of the `CoordinateAdjuster` object, that—and other—data could be encapsulated in an object which itself could be shared. This object can be passed around by other objects that have no knowledge or interest in its internals.

Finally, you see the `CoordinateAdjuster` object being used as a data member of the `View` object. Someone who is interested in finding a point at which to draw something need not even know if there is a `CoordinateAdjuster` involved: it is totally the `View`'s business.

The ultimate decisions on where to locate data and functionality often take much trial and error. Fortunately, with object-oriented programming the revisions are usually fairly easy to make.

Performance

When object-oriented programming began to become widely used in the mid-1980s, a number of people went ballistic over this hiding of data. They passed around analyses of the “wasted” processor cycles spent in getting to data through method calls.

Suffice it to say that 50 years of experience with computers shows that they get faster and cheaper—wasted cycles are of less and less importance. On the other hand, programmers get scarcer and more expensive. A technology that optimizes a programmer’s time at the expense of a computer’s time is right on the mark.

Run-Time Issues

You can design an object-oriented programming language whose “object-ness” disappears at run time. For example, the initial C++ used a preprocessor that turned everything into C for compilation; strictly speaking, no object-oriented language constructs remained, although some object-oriented structures were created in the process.

Two major run-time issues affect all object-oriented programming projects to a greater or lesser extent.

Dynamism

Dynamism (which is discussed more fully in the chapters on Objective-C and Java) refers to the general notion of allowing objects to be assembled and connected at run time rather than at compile time. The general concern is larger than an arcane compiler issue: it involves the ad hoc combination of objects

over a network that may have been written in various languages.

Dynamism is a very important concept; with the advent of the Internet as a pervasive connectivity tool, the ability to combine objects over networks is becoming increasingly important. Rhapsody and OpenStep specifically address these issues in many ways, and much of Part III is devoted to these concerns.

Managing Objects in Memory

Instances of objects are created from some big glob of memory (not to be too technical about it). In this sense, they are very different from the variables that are created on the stack in traditional methods. If you declare an integer *i* inside a block, its memory is deallocated when you leave that block and the compiler will not recognize references to it beyond that scope.

By their very nature, objects exist in another dimension than the scopes of blocks and functions. While some small objects are created on the stack in C++ (and automatically are deleted when control passes out of their block), most objects need to be kept around.

Two issues need to be addressed. The first is how to create these objects. Most commonly, chunks of memory are allocated for object creation; as objects are created and therefore use up memory, other chunks are created. Experience with much object-oriented software (particularly including MacApp) showed that memory usage for objects is very different from memory usage for other purposes. Object-specific memory managers have been developed and will continue to be developed.

The flip side of the memory allocation issue is the problem of knowing when to delete objects. OpenStep relies on a reference-counting mechanism to automatically delete objects when it is safe to do so. In all cases, for automatic deletion

(garbage collection) to work, the programmer must take a few modest steps to enable the functionality.

Frameworks

Frameworks like MacApp, ODF, and OpenStep are collections of objects that are designed to work together. You use many of the objects directly, and subclass others as needed. Together, the framework objects, your subclasses of them, and your totally customized objects allow you to easily create an object-oriented application.

OpenStep is one such framework. Part II of this book is devoted to its design and structure.

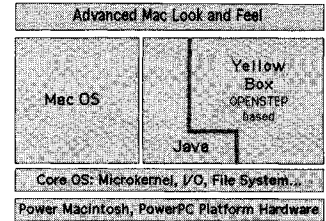
Summary

This chapter has presented some of the high points of object-oriented programming and pointed up some of the issues that come into play as you start to use object-oriented tools. Rhapsody is thoroughly object oriented, and it allows you to use its framework with several languages.

The languages of Rhapsody—Objective-C, C++, and Java—are described in the following chapters.

Chapter 4

The Languages of Rhapsody— Objective-C



OpenStep is the principal programming tool for Rhapsody. It was developed in Objective-C; you can use Objective-C (in two syntax versions) or Java to reference its application programming interface (API—the routines that do the work). Additionally, you can use C++ in your code to do application-specific processing. Integrating legacy C++ code with OpenStep is not a problem.

Just as with natural languages, choices of programming languages quickly cause passionate battles. In fact, while there are substantive differences among the three languages of Rhapsody, they have much more in common with one another than they have with traditional procedural languages. You would be quite justified in worrying what all the fuss is about, except for the likelihood that even express-

ing such a thought might subject you to suspicion on the part of the language purists.

This chapter and the chapters that follow summarize the high points of the three major OpenStep languages—not from each language’s point of view, but from the broad perspective of object-oriented programming in general and OpenStep in particular. For details of each language, consult specific language references such as the documentation that comes with your development environment of choice (which may slightly modify the language standards for its own reasons).

This chapter starts with a brief summary of how the object-oriented programming languages developed and diverged; it continues with some highlights of Objective-C:

- *dynamism*
- *class and instance objects*
- *protocols*

Note that these language chapters are highly interrelated, as they contrast the differences among the languages. For example, this chapter talks almost equally about Objective-C and C++ as it demonstrates the differences in the languages from the Objective-C side. The corresponding C++ chapter talks at length about Objective-C and Java as it draws its own distinctions and comparisons.

Object-Oriented Programming Languages: Two Directions

The origins of object-oriented programming in the 1970s lie with Smalltalk and Lisp: development environments that were used for experimentation, development, and later for production. The distinction between the language and the development environment was not clear (nor was it intended to be). In recent years, the Dylan language rethought and extended these concepts: its popularity among developers (and lack of popularity among managers and end-users) in many ways repeated the experiences of Smalltalk and Lisp.

Objects and Messages

The initial model of object-oriented programming was of objects (often based on real-world models) sending messages to one another. Thus, documents sent draw messages to rectangles, employees sent timecard messages to payroll objects, etc.

When an object receives a message, it acts on it—usually by invoking a method or function of its own. C++ programmers usually emphasize the method or function; Smalltalk and Lisp programmers usually emphasize the message. Since the message usually invokes a method of the same name, there is not a great deal of practical difference between talking about the draw message and the draw method. (There is a difference, but it is subtle.)

The Look of the Languages

The evolution of object-oriented programming has moved in two directions:

1. The objects of modern object-oriented programming are often not analogues of physical objects; they may be very esoteric and abstract objects.
2. The combined language/development environment of Smalltalk, Lisp, and Dylan has evolved into more or less separate worlds of languages (Objective-C, C++, and Java) and development/run-time environments (OpenStep, Java Virtual Machine, etc.).

The evolution of object-oriented languages has followed two main paths, both based on the C programming language. Objective-C added Smalltalk-like object and messaging syntax onto C; the result was intended to be as C-like as possible with the object-orientedness retaining its distinctive object/messaging roots.

The other path—which led to C++ and then to Java—approached the problem as one of enhancing and improving C and incorporating object-orientedness into the language as if it had always been there.

As a result, Objective-C on the page looks very different from C and from C++: it is designed to emphasize and separate the distinctions. On the other hand, C++ looks very much like C since its object-orientedness has been deliberately designed to look C-ish.

There are advantages and disadvantages to each approach, but as usual in the cases of both natural and computer languages, saying something so eminently reasonable and true will likely get your head handed to you on a platter.

Modern Syntax

Objective-C in Rhapsody offers two styles of orthography: classic and modern. The classic style maintains the original look of Objective-C—that is, a language in which the additions to C are quite visible. The modern syntax looks very much like C and C++. Both classic and modern syntaxes do the same thing in Objective-C; Objective-C with modern syntax is not C++. Snippets are given in both syntaxes in this chapter; a summary is provided at the end.

Dynamism

Perhaps the most important feature of Objective-C as compared to C++ is its dynamism: dynamic typing, binding, and loading/linking replace or expand features of a C++ compiler and are performed at run time.

Dynamic Typing

In C++, every object you refer to in your program is typed. You can use any of its methods—either methods declared in the object itself or methods in any of its ancestors. For example, the following syntax is legal in C++ and would allow you to ask a view object to return the shape at a given mouse location:

```
TShape* theShape = theView->GetShapeUnderMouse (mousePoint)
```

If the class of theView contains the following method:

```
TShape* GetShapeUnderMouse (point mousePoint);
```

that code will work.

In MacApp, for many years most objects were descendants of TObject (they aren't now). Thus, the following method declaration would not allow you to write the line of code above:

```
TObject* GetShapeUnderMouse (point mousepoint);
```

GetShapeUnderMouse returns a TObject, and presumably TShape is a descendant of TObject; therefore, you cannot use a TObject as a TShape (although you can do the reverse). To avoid a compile-time error, you would need to rewrite the line of code as

```
TShape* theShape =  
    (TShape*)(theView-> GetShapeUnderMouse (mousePoint));
```

Even if GetShapeUnderMouse actually did return a TShape at run time, you would never get past the compiler: it has no way of knowing that the TObject return value of GetShapeUnderMouse could in fact be a TShape and that the code would therefore succeed.

This is not a matter of idle curiosity, because if you have dynamic typing, you can then have dynamic binding which is the really big deal.

Dynamic Binding

In the previous example, you could continue and write code such as the following to either highlight the shape (if it had previously been unhighlighted) or vice versa:

```
if (theShape)  
    theShape->SetHighlight(!theShape->GetHighlight());
```

Presumably, theShape->GetHighlight returns a boolean value as to whether or not the shape is highlighted; the method theShape->SetHighlight sets the highlight value. This code

snippet sets the shape highlight to the opposite of its current value.

This code will work for objects of type TShape or objects that are descendants of TShape (as TCircle, TRectangle, TTriangle, etc. might be). It will compile as long as TShape contains the methods

```
virtual boolean GetHighlight (void);  
virtual void SetHighlight (boolean newHighlight);
```

If all of the objects in your application (or framework) are descended from a common ancestor such as TObject was for much of MacApp's life, you could not write:

```
TObject* theShape = theView->GetShapeUnderMouse (mousePoint)  
if (theShape)  
    theShape->SetHighlight(!theShape->GetHighlight());
```

To avoid a compile error, theShape must be an instance of type TShape or one of its descendants. In the class hierarchy assumed here, GetHighlight and SetHighlight are methods of TShape, not of the ancestor TObject.

Dynamic binding moves the checking of whether GetHighlight and SetHighlight are methods of theShape from the compiler to the run-time environment. Thus, in Objective-C, the code snippet immediately above (in which theShape is given no further definition than being of type TObject) would compile. It would get a run-time (not compile-time) error if the actual object theShape as returned *at run time* from GetShapeUnderMouse did not support GetHighlight and SetHighlight.

Note that in Objective-C the common type for generic objects is **id**. Thus, the code snippet would read (in modern syntax):

```
id theShape = theView.GetShapeUnderMouse (mousePoint);
```

or in classic syntax:

```
id theShape = [theView GetShapeUnderMouse : mousePoint];
```

(The syntactical differences will be discussed at the end of the chapter.)

Dynamic typing has many design consequences for the development of frameworks and applications. In statically typed languages like C++, there is a constant struggle to find the appropriate places for methods. Since a common ancestral class must contain the methods that all descendants need to use (like `SetHighlight`), the temptations are either to load up an ancestral class with many methods—some of which will not apply to descendants—or to use multiple inheritance so as to provide several ancestors for each object, with each ancestor providing the set of methods appropriate.

In the latter case, you might have an ancestor object called `HighlightableObject` which you could mix into appropriate shape objects; other possible mix-in objects would be `DraggableObject`, `EditableObject`, etc. However you do it, though, at compile time your `theShape` object must have a `GetHighlight` method.

To see some of the intricacies involved here, look at the `MacApp` methods that read and write views. You will see the ping-ponging that goes on as views and subclasses of views are instantiated and generic `TView` methods interact with subview methods.

The C++ fragile base class problem just exacerbates this. The fragile base class problem can be illustrated in the following scenario. If in the course of developing an application you discover the need to add a method to an ancestor of a class, all classes that descend from that class need to be recompiled. Thus, you can add a `DrawResizeHandles` to a `TCircle` object with little difficulty; if your development process leads you to believe that you should really have a `DrawResizeHandles` at the `TShape` level, `TCircle` and `TRectangle` (both of which may need it) will need to be recompiled—as well as `TTriangle` (which, for some reason, might not strictly need `DrawResize-`

Handles but nevertheless will need to be recompiled due to the changes in TShape).

With dynamic binding, you can avoid bloated superclasses and minimize recompilation due to the fragile base class problem.

Dynamic Linking The third aspect of dynamism is the ability to load and link code dynamically as needed. This capability is available to greater or lesser degrees even with languages like C++; when combined with dynamic binding, however, the ability to assemble code on the fly presents many opportunities.

Class and Instance Objects

In languages like Objective-C and SOM (the System Object Model), classes themselves can be run-time objects. As with the other issues discussed in this chapter, this can be a matter for esoteric speculation and argument; it also has practical implications.

Creating New Instances In C++, you create a new instance with the new function:

```
Triangle* myTriangle = new Triangle;
```

The instance's constructor method is automatically called.

In Objective-C, if you have declared a Triangle object, you create a new instance by calling a method of the class object which creates a new instance.

```
Triangle* myTriangle = Triangle.alloc; {modern}
```

```
Traingle* myTriangle = [Triangle alloc]; {classic}
```

By convention, the alloc method creates a new object. Note the difference: the C++ example uses a function (new) which is not an object-oriented construct; the Objective-C example

uses an object-oriented construct in the form of the class object (Triangle).

Each declared class has a single class object which is referenced by the class name (Triangle, here). You define methods for each class as being available either to the class object (such as `alloc`) or to the instance objects. The code

```
myTriangle.alloc
```

would fail, since `alloc` is a class method (available only to Triangle and not to any specific instances such as `myTriangle`).

Rather than relying on a constructor, you call an initialization routine (which by convention starts with `init`).

Thus, the code to allocate and initialize a triangle could be written:

```
TTriangle * myTriangle = (Triangle.alloc).init; [modern]

TTriangle* myTriangle = [[Triangle alloc] init]; [classic]
```

As you can surmise, the result of the `alloc` method is an instance of Triangle on which the `init` method is called; the result is stored in `myTriangle`.

Class objects are of class `Class`; instances are objects of their given class.

Using Class Objects

Class objects can also be used for purposes other than simply as factories to create instances of themselves. If you have written C++ code, you may have experienced the joys of getting yourself wrapped up in Runtime Type Identification (RTTI) in which you need to find out if an object that you happen to have is an instance (or descendant of) a certain class. This often happens when you are maneuvering around the limitations of static typing and binding—before coercing a descendant of `TShape` to a `TCircle`, for instance, it's a good idea to know that it is a `TCircle`.

A class object can return its class—not its class name (which you can derive from the class)—but a return value of type `Class`.

Protocols

C++ relies on class hierarchies to create objects—a class’s methods are the ones it declares as well as those of its ancestors. With multiple inheritance, you can mix in a number of classes, each of which is targeted to specific functionality.

Objective-C does not support multiple inheritance, but it does support a multidimensional structure to create objects. As in C++, a class’s methods are its own as well as those of its ancestors. In addition, you can declare protocols—which are collections of methods—that can mix into classes.

A protocol could consist of mouse management methods (you could have a protocol to implement the hypothetical `DraggableObject` mix-in class referred to previously). Classes are said to adopt protocols; they conform to protocols if they either adopt them or are descendants of classes that adopt them.

Unlike mix-in classes in C++, protocols consist only of methods (not instance variables); furthermore, protocols merely define the methods that the class adopting the protocol must implement. The protocol itself contains no implementations but is applicable only to the interface.

In declarations, a generic id can be qualified with a protocol—thus `id<NSCoder>` refers to any object that adopts the `NSCoder` protocol.

Keeping It Running

The strict compile-time type checking that C++ provides helps to make certain that you do not wind up calling a non-existent method. In Objective-C, you often have to do that work yourself—at run time. (You can use static typing to have the compiler do the work for you, but you often use dynamic typing and may be prone to problems.)

As noted previously, an Objective-C object responds to a message which selects a method or function to invoke. Accordingly, the message may be called a selector—as it is in this code sample.

All objects that descend from NSObject (hence most of the objects you will be working with) have a `respondsToSelector` method to which you pass a selector which is the method you want to check:

```
if ( [anObject respondsToSelector:@selector (GetPoint::)] )
... [classic]
if ( [anObject respondsToSelector:@selector (GetPoint)] )
... [modern]
```

You can also use variables of type `SEL` to store references to selectors that you may use frequently:

```
SEL pointMethod;
pointMethod = @selector (GetPoint::); [classic]
pointMethod = @selector (GetPoint); [modern]
```

Checking `respondsToSelector` is as natural to Objective-C programmers as checking for NULL objects is to C++ programmers.

OpenStep also provides a method of NSObject which you can override called `forwardInvocation`. This method is called (a `forwardInvocation` message is sent) in response to an unimplemented method being called in an object (that is, in response to an unidentified message). The unprocessed message is returned as a parameter to `forwardInvocation`; you

can then use `respondsToSelector` on other objects to find an object to which to forward the message and to avoid a run-time error.

Syntaxes

Objective-C in Rhapsody supports both classic and modern syntaxes. You can use whichever is more comfortable for you, as long as your development environment supports it.

Classic Syntax

In classic Objective-C, messages are sent to receivers, sometimes with parameters. For example, if you have a view object that responds to the `GetShapeUnderMouse` message with a point parameter that describes the mouse position, you can write it as

```
[aView GetShapeUnderMouse : thePoint]
```

All Objective-C messages are embedded in square brackets.

If there is more than one parameter, you may name the subsequent ones. For example, if `GetShapeUnderMouse` had a second parameter, you could rewrite the above code as:

```
[aView GetShapeUnderMouse : thePoint, saveShape:YES]
```

The first parameter is never named.

The result is returned and can be assigned to a variable. In this case, it is logical that an object be returned. All objects can be referred to as type `id`, so you could write:

```
id theReturnedObject = [aView GetShapeUnderMouse : thePoint];
```

Modern Syntax

In modern syntax, methods of objects are called. *Nothing changes semantically—it's just the notation.* Thus, you could write

```
id theReturnedObject = aView.GetShapeunderMouse (thePoint)
```

to replace the code sample shown at the end of the last section.

Summary

This chapter summarizes the high points of Objective-C for object-oriented programmers who are familiar with other languages. The rest of the syntax is quite comparable—interface and implementation files are as they are in other languages. Directives to the Objective-C compiler start with @, which is a change from other languages, but that shouldn't stump you.

What matters most about Objective-C is its dynamic nature which makes much of the OpenStep frameworks possible; it also eliminates the need for some of the more complex C++ structures (in particular multiple inheritance).

Protocols, for example, allow the creation of complex objects along two dimensions—one dimension of inheritance and another dimension of common functionality.

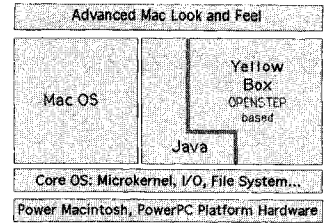
Objective-C—particularly with classic syntax—is very much a product of the object-oriented world of the 1980s. It takes the then most popular programming language (C) and adds object-orientedness to it without touching the basic C syntax. It incorporates the very concrete notions of objects as message processors that were prevalent in Smalltalk, Lisp, and other early object-oriented systems.

Incidentally, do not make the mistake of thinking that Objective-C is dated or old-fashioned. Its roots are different from those of Java and C++ but all in fact evolved over time; the fact that they evolved in different environments accounts for some of their differences. If there were a clear developmental structure of A being supplanted by B which in turn was supplanted by C, there would be no question that C is the preferred language. That is not the case, and there are reasons to use each of these languages today.

As people have used object-oriented programming techniques, their approach to architecture has changed. Objects have gotten smaller and smaller. In Java, the most recent object-oriented programming language, everything is an object—as you will see in the next chapter.

Chapter 5

The Languages of Rhapsody—Java



Java caught the imagination of the programming world in the mid-1990s. It seemed to promise one of the long-sought objectives of object-oriented programming—the ability to write objects that could be used and reused in different contexts without having to recompile and relink them. In fact, Java does do this to a large extent.

Java was designed originally as a language to use for embedded microprocessors in consumer appliances. Issues such as stability, reliability, and compatibility with network communications were paramount. On the World Wide Web, these issues turned out to be very important, and the ability to write and compile Java applets once and to deploy them via Web pages is very important.

Java can be used to create applets—small programs that can be distributed over the Web and embedded in HTML. It can also be used as a traditional programming language. In Rhapsody, a Java Virtual Machine is part of the overall architecture so that applets can run. Furthermore, you can use Java as a programming language to access the OpenStep APIs.

This chapter provides a high-level look at Java, emphasizing some of its similarities to and differences from other object-oriented programming languages. As with the other chapters in this section, it is no substitute for a complete language reference; however, if you are an experienced object-oriented programmer, you may well find that the sample code from Apple is sufficient to get you going with Java (or Objective-C or C++) as you start to experiment with Rhapsody.

The Look of Java

If you look at Java code, two things may strike you immediately:

1. Java allows Unicode characters.
2. Java looks a lot like C and C++.

Unicode

Most programming languages limit the characters their compilers process to standard ASCII characters. This eliminates such useful symbols as π and Σ as well as all non-Roman characters and a number of non-English characters as well. Java accepts any Unicode character which makes it possible to write readable code in any language. (Unicode characters are 16-bit characters; you can represent not only the standard ASCII upper- and lowercase letters in Unicode, but also the double-byte characters from non-Roman alphabets.)

The choice of a character set for a language has long been a matter of some debate. One of the original criticisms of Algol by the Cobol language team was that the Algol replacement operator could not be represented by a single keystroke on the IBM 026 keypunch machine. While Algol syntax required $x \leftarrow y$ to indicate that x was to receive the value of y , keypunch operators had to use the two-character symbol $:=$ to achieve that result— $x:=y$. You may note the great pride that the Cobol team expressed in proclaiming that their language didn't require any such rigamarole. (And who would have thought that the IBM 026 keypunch would not remain the programmer's primary input device forever?)

Java Isn't C

Java looks like C, C++, or modern syntax Objective-C. However, there is one critically important difference: Java is not built on top of C. Both C++ and Objective-C were designed to expand, enhance, and (in the case of C++) improve C. They started with the assumption that they would do not violence to the original language. C code should compile without error in a C++ or Objective-C compiler.

Java, on the other hand, is not built on top of C and is free to reimplement C constructs. For example, C defines a basic type called `float` to be a floating point number. Both Objective-C and C++ therefore recognize the `float` type.

In Java, `float` is not a floating point type. Although it serves the same purpose, `float` in Java is in fact a floating point class (as opposed to a type). In fact, everything in Java is an object, and that is one of the most significant differences between it and the other object-oriented languages.

This line of code has identical meanings in C, Objective-C, and C++:

```
float taxRate;
```

Its meaning in Java is similar, and you can write it and use `taxRate` more or less as you would in the other languages, but in fact `taxRate` will be an object in Java.

Everything Is an Object

As you saw in the chapter on Objective-C, classes in that language are themselves objects, and you can use class methods and class variables as well as instance methods and variables. Class objects are commonly capitalized while their instances are not. Thus

```
Triangle.alloc
```

calls the Triangle class's alloc method, while

```
aTriangle.draw
```

calls a specific triangle instance's draw method.

Java uses a C++-like syntax in which classes can have static fields and methods; those fields and methods belong to the class, not to particular instances.

In Objective-C, classes are members of the Class class, and they inherit some utility methods from it. Java approaches the issue somewhat differently.

Object Class

In Java, all objects are descendants of the Object class, and they therefore inherit the utility methods from it. Object has 12 methods; therefore all Java objects have at least these 12 methods—either by inheritance or through overrides. Since everything is a descendant of Object, every element of the language—every boolean, every array, every string, and every floating point number—can do any of the things listed in Table 5-1.

TABLE 5-1. Java Object Methods

Method	Use
<code>Object</code>	constructor
<code>clone</code>	provides a copy of itself with all fields set to the same values as in the original
<code>equals</code>	returns true if another object is the same as this one (i.e., refers to the same object in memory)
<code>finalize</code>	does nothing; similar to a C++ destructor—designed to be overridden in subclasses
<code>getClass</code>	returns the run-time class of an object (run-time classes are described later in this chapter)
<code>hashCode</code>	returns an integer value that uniquely identifies this object at run time; different values may be returned during different executions
<code>notify</code>	wakes up a thread waiting on this object's monitor
<code>notifyAll</code>	wakes up all threads waiting on this object's monitor
<code>toString</code>	returns a textual representation of this object; it is meaningful (to people) and concise but usually is not the entire object's data
<code>wait</code>	waits for a thread to notify it of a change
<code>wait (long timeout)</code>	waits with a timeout in milliseconds
<code>wait (long timeout, int nanos)</code>	waits with a timeout in milliseconds plus nanoseconds

A glance at the methods will show several important aspects of Java:

- Java is designed for a multithreaded environment (notify, notifyall, and wait methods).
- Java defines a run-time object (Class) that interacts with the descendants of Object that you describe in your code.
- The hashCode method demonstrates Java's implementation of a mechanism for locating and identifying objects on the fly at run time.
- The finalize method is called as an object is being automatically destroyed by Java's garbage collection mechanism. You don't have to do anything to have automatic memory management with Java.

Some Java Syntax

Furthermore, the actual Object API shows several language features that distinguish Java from other languages. Here is the API:

```
public class java.lang.Object
{
    //Constructors
    public Object ();

    //Methods
    protected Object clone ();
    public boolean equals (Object obj);
    protected void finalize ();
    public final Class getClass ();
    public int hashCode ();
    public final void notify ();
    public final void notifyAll ();
    public String toString ();
    public final void wait ();
    public final void wait (long timeout);
    public final void wait (long timeout, int nanos);
}
```

These are some of the syntactical elements that distinguish Java from other languages:

- Names in Java can be qualified to make them unique, as in `java.lang.Object`. You can create your own name space; the convention is to reverse your Internet domain to provide a unique identifier. Thus, Apple's Internet address is `apple.com`; a unique Java identifier for Apple Java objects is `com.apple`. (Since the Internet domains are unique, the reversed-order identifiers are also unique.)
- The `clone` and `getClass` methods return objects. In C++, they would return pointers to objects—`Object*` and `Class*`. Since everything in Java is an object, it is unnecessary to distinguish between references and objects. And in fact since there are no pointers in Java, that distinction would be irrelevant.
- Methods can be marked as `final`, which means that subclasses cannot override them. This is in contrast to the approach in C++, where methods are marked `virtual` to indicate that where a like-named method exists for a subclass, that method should be used instead. C++ asks you to specify when a method will be overridden; Java asks you to specify when it cannot be overridden. Classes and fields can also be marked as `final`.

In the case of `Object`, only `clone`, `equals`, `finalize`, `hashCode`, and `toString` can be overridden. Since these all obviously need to use the fields of a specific descendant of `Object` this makes sense. You cannot override the thread-management and run-time class management methods.

Inheritance and Organization in Java

Organizing the parts of an object-oriented system is always a challenge. C++ provides for organization primarily through

the use of inheritance (including multiple inheritance). Objective-C supports not only inheritance but also the concept of protocols—sets of methods that as a unit can be adopted by an object.

Java addresses the issue in a somewhat similar manner to Objective-C. Any class can be a descendant of any other class—in Java terms, a subclass extends its superclass, as in:

```
public final class java.lang.Boolean extends java.lang.Object
```

This is the Boolean class which wraps the primitive Boolean value that corresponds to the boolean type in C. It uses fully qualified names for itself and its superclass (java.lang.Object); by being declared as final, the class cannot be overridden.

You can also declare an interface which looks like a class declaration, complete with methods and fields. A specific class can extend a given superclass and may implement any number of interfaces in a manner somewhat similar to Objective-C's adoption of protocols.

You can query the Class object at run time to find what (if any) interfaces it implements.

Packages

You can assemble classes into a package in a single source file and import the package as a whole. You identify the package with a fully qualified name using the convention cited previously of starting with your reversed Internet domain name (as in com.apple).

You can then import a package into any other Java file. Names are unique within packages and—if you are using the reverse Internet naming convention—package identifiers will be unique. You can refer to items within a package either by fully qualified names or by having imported the package.

Frameworks and Java

From the start of object-oriented programming, there has been uncertainty over what features should be part of the lan-

guages and what should be part of a framework. C++ is very pure in this sense—it has no framework-type constructs (such as a common object from which all others descend).

Objective-C has some framework-type syntax; when used with OpenStep it of course is tightly bound to that framework. For example, `NSObject` is the common ancestor of most OpenStep objects and has many common utility functions that OpenStep objects can use.

In Java, the use of a common ancestor (`Object`) in the language itself means that some of the framework-type functionality moves into the language. This includes object inspection at run time (what type of object is this, does this object support `X`, etc.) as well as extensive exception handling.

Exception Handling

Java uses the try/catch/finally mechanism that is familiar to most programmers today. Method declarations explicitly specify the exceptions that may be thrown with a throws clause:

```
public int read (byte b[], int off, int len) throws IOException
```

Exceptions themselves are objects; using the principle of polymorphism, you may actually throw a descendant of the exception type specified in your declaration.

Summary

This chapter has summarized the high points of Java—particularly as they differ from C++ or Objective-C. The principles of Java are those of object-oriented programming: inheritance, polymorphism, and data hiding/encapsulation.

Java specializes in providing a particularly robust run-time environment and in providing a well-structured way of combining objects. Java's packages, interface/implementation

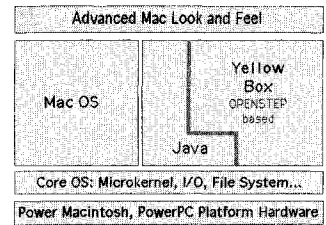
structure, and the ability to specify that methods, fields, or classes are not overridable (with the `final` keyword) provide a sturdy foundation.

Furthermore, Java's choice of not implementing multiple inheritance (although providing some of its features through interfaces) increases the manageability and maintainability of Java code. While multiple inheritance is one of the topics that can set off religious wars in the programming community, it is not one of the features that makes for easily maintained code.

Java is very close to Objective-C in its structure and in the degree to which it implements some important run-time features. By contrast, C++ is very sparse in making any assumptions about what happens at run time. Its power is focused on the compile-time language and its constructs.

Chapter 6

The Languages of Rhapsody—C++



C++ is the other major language that you can use with Rhapsody. Perhaps the most widely used object-oriented programming language today, C++ has many partisans and experts; in addition, a great deal of existing code (tomorrow's legacy code) is written or is being written in C++.

You can use C++ with Rhapsody, and you are encouraged to do so. The mingling of C++ with dynamic object-oriented languages (Objective-C and Java) is no problem within the basic functional sections of your code; however, for the interface, you should be using Interface Builder, which provides a consistent interface and a modern graphically oriented interface building environment. C++ has a limited role to play in that area.

This may worry you: what about all that existing code? What about your expertise? Have no fear: most of that code is reusable or easily modified, and your expertise is primarily at a higher level than language syntax. Your favorite C++ language tools are available with Rhapsody's dynamic languages; they may go by different names, and fierce partisans of one language or another may fight to the bitter end saying that the language structures aren't parallel, but from the practical point of view of the professional programmer, the differences are not great. What true differences there are all revolve around the issue of dynamism—dynamic binding, loading, and typing.

This chapter covers a few of the hot issues regarding C++ and Rhapsody.

The C++ Objective

"C with Classes" began its life at Bell Labs in 1979. It was designed from the beginning to be a production language—one that would be usable in real-world everyday projects. In this it differed from Smalltalk and Lisp, which were experimental languages. Smalltalk and Lisp (as well as Simula) convinced a number of computer scientists that there was indeed a practical benefit to the use of object-oriented programming. What was necessary was a way to bring it into the world where production programming was being done.

When C with Classes became C++ in 1983, the computing world had changed dramatically from what it had been only a few years earlier when C with Classes was born. The personal computer was well on its way to becoming the widespread phenomenon it is today. Programmers and users alike were turning from mainframes to personal computers because of the ability to do their computer work themselves without the mediation of large professional staffs.

In the early 1980s, personal computers were far from the powerful devices that exist today. The first Macintosh (1984) boasted 64K of memory. Fifteen years later, a thousandfold increase in memory to 64MB is not uncommon; many computers ship with 32MB as standard—a mere 500-fold increase.

It was an era in which computer resources—memory, processor, and disk storage—were husbanded carefully. The first compilers on personal computers were designed to run in very small memory partitions at the expense of constant diskette-swapping as memory structures were swapped to and from disk.

It is therefore not surprising that when a new language was developed it was designed to respond to the needs of the programming community. The single most significant design decision of C++ was the decision to move all possible decisions from run time into the compiler: it might take longer for the program to compile, but the finished code would run as quickly as possible.

This was a change from the designs of the predecessors of C++. Lisp and Smalltalk were dynamic languages—a lot happened at run time. They had reputations as being very slow (which they were), and object-oriented programming had a big hurdle to overcome because many people believed that object-oriented programming automatically meant at best sluggish performance of the final product.

The decision to make typing static (rather than dynamic) and not to support run-time binding was far from a mistake: in fact, you can argue quite convincingly that by making object-oriented programming feasible in the real world, C++ was largely responsible for moving it from computer labs to the workplace.

Today, processors are much faster, memory is plentiful, and disk space is cheap. Dynamic languages like Objective-C and Java are not disadvantaged by their dynamism, although you

can still hear legitimate concerns about the performance of Java (this is improving by the day; it is unlikely to remain a serious issue after another few years). If you go back and read some of the books about the evolution of C++, you will find emphasis on the need to keep the run-time performance of C++ programs as high as possible as well as on the need to make the code as portable as possible. The choices with regard to dynamism (specifically, the decision not to use it), are related to these concerns far more than to abstract designs.

C++ is not a dynamic language, and that's that. It was the right language at the right time, and every object-oriented programmer today owes a tremendous debt of gratitude to the C++ designers who made the choices that made the language and the technology usable. C++ remains usable on Rhapsody; the dynamic languages—Objective-C and Java—are just a little bit more usable.

Consequences of Compile-Time Typing

By forcing the compiler to do as much work as possible, the run-time code is kept very efficient. The flip side is that in C++ design decisions and implementations must be made much earlier in the process than is possible with dynamic languages.

Two consequences of compile-time typing are discussed in this section: the use of small, stack-based objects, and the fragile base class problem.

Small Objects Are Not Expensive

No matter how much the compiler and operating system try to minimize the impact, object-oriented programming and its attendant data structures do exert a cost at run time. Programmers have known this from the start; particularly in early days the use of objects was quite limited. Because of the over-

head involved in constructing and accessing objects, the objects tended to be fairly large structures.

Over time, it became feasible to use small objects—sometimes very small objects—in programs. Because much of the object management is done by the compiler, there is little run-time cost associated with these small objects. Furthermore, some frameworks have their own memory allocation routines which optimize the allocation and deallocation of these small structures; the OS memory allocation routines were often optimized for relatively large objects and data structures.

Small objects often have very brief lives; in MacApp, for example, you will find objects that wrap platform-specific structures like points and rectangles. These objects are used precisely as the flat structures would have been used: they are declared on the stack at the beginning of a function and are implicitly destroyed when control passes out of the function, as shown in this code snippet:

```
void TMyObject::DoSomething (void)
{
    int i = 0;
    CRect aRect;

    ...
}
```

On exit from the function, neither `i` nor `aRect` is available for use. The C++ compiler has taken care of preparing the stack to be cut back; it has also taken care to make certain that any constructors and destructors for `aRect` are automatically called.

In fact, many programmers use this feature to release memory structures automatically. If you create a small stack-based object that wraps a standard toolbox call, you can implement a single method—its destructor—and be guaranteed that it will automatically be called as necessary to deallocate auxiliary memory structures.

The referencing of methods within an object is done by the use of a virtual dispatch table for each class—a vtable—which is built at compile time by the compiler. These vtables make run-time dispatching extraordinarily efficient.

In Objective-C, there are no vtables. The dispatching is done at run time by highly optimized code that checks at that specific moment to find the method to call. With today's processors, the performance hit is rarely noticeable. However, because it is there and can be noticeable with a multitude of small objects, OpenStep rarely uses small objects such as the CRect object in the sample above. (OpenStep declares NSPoint, for example, as a struct; MacApp's VPoint is a stack-based class.)

The Fragile Base Class Problem

The fragile base class problem is perhaps the most serious issue plaguing C++. Since all of the mapping of objects is done at compile time, every class's header structure must be known then. A change to any class's header requires recompilation of all of the classes that descend from it. This has caused severe problems as people have tried to reuse objects.

A number of attempts have been made to work around the fragile base class problem (including the Standard Object Model—SOM), but it remains a fundamental issue in dealing with C++.

Run-Time Type Identification and Exception Handling

Recent additions to C++ provide the ability for objects to query themselves and others about the classes to which they belong. This feature (Run Time Type Identification—RTTI) is standard in Objective-C but was a long time coming to C++. The reasons it took so long were that not everyone was convinced it was a good idea, and more important, it required building and maintaining a run-time set of information about objects that C++ had never had to keep before.

Related to RTTI is native exception handling. The standard model for handling exceptions is shared by most contempo-

rary programming languages—declare a try block in which you do the work, declare a failure block to which control is transferred if something fails, and finally declare a completion block that is executed if things work properly. This is something that is not always easy for a compiler that expects to be able to set everything up in advance.

Multiple Inheritance

C++ supports multiple inheritance; Objective-C and Java do not. If you want to start a barroom brawl or language war, bring up this point. People have very strong feelings about it. In fact, to hear some people, multiple inheritance is the very essence of C++.

The History of Multiple Inheritance in C++

Multiple inheritance appeared in C++ in Release 2 (June 1989). It was not part of the initial design for C with Classes. So much for the very essence of C++.

Multiple inheritance is very useful and valuable; it is widely used in frameworks like MacApp, and it clarifies the program structure. In its simplest form, multiple inheritance means that an object can have two superclasses. In his book on the development of C++¹ Bjarne Stroustrup, the designer of C++, presents three cases in which multiple inheritance can be useful:

1. You can combine two completely separate objects into a third.
2. You can combine related objects: for example, he presents an abstract stream class, abstract input- and output-stream descendants, and finally a file stream class

1. *The Design and Evolution of C++*, Bjarne Stroustrup. Addison-Wesley, 1994, p. 271.

that merges the input- and output-stream interfaces.

3. You can combine an interface and implementation class to produce a fully configured object.

All of these are valid cases. The first can save recoding and re-designing a new hybrid class; the second makes for more easily maintained code; the third may help you reuse utility objects in a customized solution.

In practice, another aspect of multiple inheritance has become obvious: rarely do two objects of equal size and importance become joined in the bonds of multiple inheritance (an exception is the structure of the symmetric input- and output-stream objects in the second case cited). If you look at code that uses multiple inheritance, there is usually a primary object that mixes in some additional classes of relatively small, focused functionality; these classes are called mix-in classes.

It is not surprising that the mix-in class structure should be so popular: it provides the advantages of multiple inheritance, while avoiding the programming headache of managing an object that truly has two equal parent objects. It is much easier to live in the mix-in world with a primary object and the subsidiary, targeted mix-ins.

This usage of multiple inheritance is very important when comparing C++ and its multiple inheritance to the single inheritance structures of Java and Objective-C. The functionality of the mix-in flavor of multiple inheritance is provided through delegation and to a lesser extent through the use of protocols. These are no substitute for the Grand Scheme of Multiple Inheritance, but they are a very satisfactory substitute for the reality of the use of mix-in multiple inheritance.

C++ Delegation

In the nitty-gritty world of programming, multiple inheritance means that you can call three kinds of methods from a class that you create:

1. You can call methods that you declare and implement in that class.
2. You can call methods from the first superclass of your class.
3. You can call methods from the second (or nth) superclass of your class.

The implementation of multiple inheritance must make this possible. Object-oriented programming languages all provide for the ability to call methods from your class and from a superclass. It's calling a method from the second (or nth) superclass that makes for multiple inheritance—and the problems.

Language-Level Delegation in C++ An early attempt at implementing multiple inheritance used the concept of delegation: in your class's declaration, you could declare a delegate class; if you called a method that was not in your class and not in your superclass, the delegate class was given an opportunity to perform the command.

This language-level implementation of delegation caused problems. They were chiefly implementation problems for programmers: the relationship between delegate and primary class was close but not close enough. For example, in the multiple inheritance design as implemented today, you can override methods from any of your superclasses. In this delegation structure, you could override methods in your single superclass, but not methods from either the superclass or the delegate.

Another aspect of this problem was that while your class knew about the delegate, the delegate had no reference back to your class (unless you implemented one specifically). And so, language-level delegation in C++ was dropped in favor of multiple inheritance.

Implementation-Level Delegation in Rhapsody Delegation is used throughout Rhapsody, and as implemented there these problems do not occur (at least not nearly to the extent they did in the early C++ experiments). Two types of delegation are used:

1. You can declare a specific delegate variable for a class. Your class will send certain messages to that delegate for processing.
2. You can use the forwarding mechanism that is based on overriding the NSObject `forwardInvocation` method. This method is called at the moment when neither the class nor its superclass can handle a method call (the moment when the second superclass in the C++ world would get a chance). Your `forwardInvocation` method receives all of the information about the method being called and can attempt to forward the message to any other object that can process it.

This structure in Rhapsody provides much of the functionality of multiple inheritance in C++; it also makes several points even clearer than they are in C++:

- There are clearly a primary object and subsidiary objects in the single superclass and possibly many delegates. This conforms to the way in which most programmers think.
- You can control the calling of methods beyond your own class and your superclass. Whether you use delegation or forwarding, it is up to you to pass on the message.
- You can control which other object gets the message; this eliminates an area of ambiguity in C++ multiple inheritance, where there are many superclasses with possibly conflicting names. The way around this in C++ is to call a specific superclass's method, qualifying it with that superclass's name. You don't have to

do this, however. Thus, you can get yourself into a situation where your code works today, but tomorrow when you add another superclass (with a conflicting name) your code may no longer work properly.

- Since you manually establish the link to other classes (either dynamically with `forwardInvocation` or by establishing a delegate), you can create whatever context-specific links forward and back that you need.

Multiple inheritance is something people feel very strongly about; in practice, you can accomplish what you need to do either with C++ or with the Java and Objective-C structures. There are pros and cons of both approaches; as you may imagine, the Java and Objective-C architecture lends itself to dynamic processing much more so than the static structure of C++.

Rhapsody and C++

Because the world of Rhapsody is so bound up with dynamism, the static structure of C++ poses certain problems with integrating the dynamic objects of the OpenStep frameworks. Specifically, you can't override the framework classes in C++, nor can you (easily) call their methods.

Before you decide that this is a disaster, consider the implications of this. Most important, your C++ (or C) code can be used within Rhapsody applications. Any existing code that performs application-specific functions can run quite happily in this world. The code that will have problems is existing C++ code that deals with the program interface.

If you are converting code to Rhapsody, that is far less of a problem than it might appear. Throwing out pages and pages of legacy C++ code in favor of using the standard interface is a benefit for everyone. You have less code to maintain and

test, and you are guaranteed a consistent and conforming Rhapsody interface. *Throwing out legacy C++ interface code doesn't mean rewriting it: it means that you don't need it anymore when you're using OpenStep.*

The code that you have that performs the work of an application can remain as C++ code and can be called as needed in your application.

The only C++ code that does need rewriting is any interface-related code that does things that Interface Builder and OpenStep don't do for you. These fall into two general categories:

1. Things that they don't do and that you want to do because you're bringing a preconceived other interface along with you. In this case, forget it. Use the proper interface.
2. Things that they don't do that are specific to your needs and that you must add. Although this code may live in your interface classes, it is truly application specific; you can call C++ code as needed. If it really is interface-related (involving drawing, for example) you may need to rewrite it in Objective-C or Java. This is actually a very small amount of code for almost everyone. (For most people, it is no code at all.)

Should You Write in C++? If you are converting an existing application to Rhapsody, you may choose to leave its functionality in C++ while using Interface Builder and OpenStep (together with Objective-C or Java). Depending on who has what skills (as well as the magnitude of the code base) it may be desirable to do a full-fledged port. This is an individual decision that must take into account deadlines, individual expertise, etc.

For new projects, Objective-C and Java are the preferred languages. However, if your resources are all C++, there is no reason not to continue to write your application's functional-

ity in C++ and to use Interface Builder, OpenStep, and Java or Objective-C for the user interface.

Latitude An additional option is available for legacy code on the Mac OS. Metrowerks has a product called Latitude that you can use to recompile Mac OS applications for the Yellow Box. This allows you to use C++ directly. Several caveats need to be raised:

- Your legacy code will run, but not all Yellow Box features may be available.
- Recompiling under Latitude is very easy for your well-designed and well-behaved applications. On the other hand, if they're well-designed and well-behaved, porting them directly to Rhapsody isn't a big deal, either. The problem legacy systems are the monsters that have been written and rewritten over years and years where the original system design (if there was one) is not discernible.
- You postpone the learning experience and programmer productivity benefits of Rhapsody.
- Finally, you add another piece of critical software to your mix of tools.

The benefits of a quick and easy port using Latitude are clear when your objective is to get something converted and running under Rhapsody. For ongoing system development, you must evaluate all of your choices.

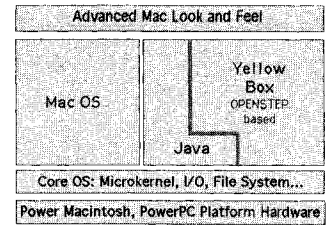
Summary

C++ is one of the most popular programming languages today. Existing (and new) code in C++ can be used within Rhapsody to implement application functionality as needed. The use of C++ in conjunction with OpenStep and Interface Builder is circumscribed; however, it is not so much a question of

rewriting C++ interfaces as of jettisoning existing C++ code in favor of the frameworks and no (or little) custom interface code.

Because of the widespread use and popularity of C++, it is likely that C++ tools will be available for Rhapsody programmers. Metrowerks's Latitude porting library is the first of these.

Part II: The Rhapsody Frameworks



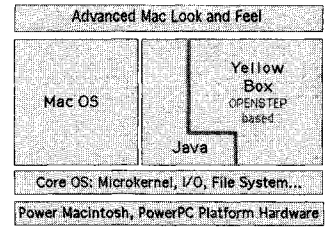
There are many ways to approach a framework—the goal here is not to list the classes in alphabetical order or according to the kits to which they belong or the files in which they are delivered. The classes are grouped according to functionality—building blocks, visualization, sharing, printing, etc. Within each chapter, the classes are arranged by subject area—color, commands, text, programming tools, etc.

The bulk of this part is devoted to the primary framework—OpenStep. OpenStep is undergoing expansion as part of the Rhapsody development process. All of the objects described here exist in Rhapsody, but they are joined by new objects implementing the QuickTime Media Layer (QTML), scripting, enhanced interface developments for Advanced Mac Look and Feel (many in the area of menus), and document-centered computing.

The architecture of these frameworks is similar: as you become familiar with it, learning about new classes (and even new frameworks) will be easy. The complete Apple documentation—on paper and online—contains the latest on these frameworks. New classes are documented there, but if you have a general understanding of the architecture, you will find that their structures are not new to you. Also, your development environment will have environment- and language-specific additions to this section.

Chapter 7

Rhapsody Frameworks Overview



The primary framework for Rhapsody is OpenStep. It consists of two kits—the Foundation Kit and the Application Kit. Other major frameworks are the Enterprise Objects framework and the WebObjects framework; both rely on the Foundation Kit.

The documentation that comes with your Rhapsody tools (whether from Apple or a third party such as Metrowerks) describes the process of creating applications. The step-by-step process is outlined in their tutorials: boring though it may be to follow a tutorial to produce an application that does something as mundane as putting up a basic text processing window, the process is valuable. For one thing, actually going through the tutorial and building the application will guarantee that all of your files have been installed correctly

on your system. After all, if the tutorial doesn't work, it's not your fault—and the technical support people at the vendor should be able to help you out.

Development Tools

One of the most important aspects of Rhapsody is that it brings to programmers the advances in user interface design that users have enjoyed for years. The two basic tools you use within Rhapsody are Project Builder and Interface Builder. (Consult the documentation that comes with your development environment for further information.)

Project Builder

Project Builder is the graphically oriented tool that lets you build and maintain applications. In the Rhapsody environment, applications consist of a related series of files (see “Bundles” on page 116); Project Builder manages these files for you.

Project Builder also provides the interface to the lower level tools that actually construct your application. Like many development environments, it calls compilers and linkers to construct the code. It constructs three makefiles (which you can look at but shouldn't touch) to control this process: `makefile.preamble`, `makefile`, and `makefile.postamble`; it runs them in that order.

Interface Builder

Interface Builder should be used with great caution. It is the tool that lets you construct an interface graphically and dynamically. You can link objects (including your custom objects) to interface elements such as buttons and windows; you can specify the precise method of an object that should be called when the user clicks on an interface element. You can wire objects that receive events and messages to predefined outlets in your own and others' classes.

Unlike some other graphical development environments, Interface Builder is not really a front end to text-based systems; Interface Builder builds graphically oriented code. In fact, developers who have used Interface Builder make a distinction between the code that Interface Builder produces and “ASCII code”—the text-based code that you are used to.

The reason for the caution in using Interface Builder is simple. If you are going to be using Interface Builder and Rhapsody exclusively for your development, you don’t have to worry. If you are going to have to return to traditional tools, you are certain to experience exasperation and frustration. Like telling a mystery fan the denouement, the experience of using Interface Builder will ruin other development environments for you.

The Development Components

Over and above the classes of the Rhapsody frameworks and the tools that you use, four parts of Rhapsody’s application structure are important to know about. Fortunately, they are very simple.

Nibs

Nib files are built by Interface Builder (they were formerly known as Next Interface Builder files). Nib files contain the code that you have built with Interface Builder. The connections that you wire between objects are stored there, as are the menus for your application. Images related to the interface (such as a picture for a button) are stored in the nib files.

The bulk of the nib file storage is devoted to objects and to references to objects. As you run Interface Builder, you actually create standard interface objects such as buttons and text views. When you save the nib file, these objects are written to the nib file using the standard archiving methods of Open-

Step. The object that you create in Interface Builder is the object that is used in the application.

In the case of custom objects that you reference in Interface Builder (that is, objects that you refer to by name rather than by dragging them from the palette), only a reference is stored. Since Interface Builder doesn't know about your custom objects (except for their names), it cannot instantiate and then archive them; instead it stores references to them.

An application usually has a primary nib file; it may have other nib files. You can group interface elements that are used together into separate nib files that are loaded as needed. This is an efficiency strategy that works when the functionality of your application can be divided. If all of the interface needs to be loaded at all times (probably a poor design, anyway), there is no reason to split it up into separate nib files (unless different people are working on different parts of it and need to be able to work simultaneously on separate files).

Bundles

Bundles are the collections of files that make up an application. They can include nib files, but need not (in the case of applications with no interface or of applications that for some peculiar reason are not built with Interface Builder).

Whereas nib files consist primarily of objects and references to objects, bundles consist primarily of code. This is the object code for your application.

As with nib files, you can split the application into several bundles; the reasons for doing so are the same—the ability to load only those parts of the application that are needed and the ability to have several people work on the same files simultaneously during development. (In some ways, bundles are similar to shared libraries which you may already be familiar with.)

NSApplication

When you run an application, an `NSApplication` Class object is created; its `sharedApplication` method is called, which returns an instance of `NSApplication`. A global reference to this shared application is stored in `NSApp` and is available from any of your program files and objects. (You usually do not subclass `NSApplication` as you would in other frameworks—see Chapter 9 for more details on `NSApplication`.)

As part of its initialization, `NSApplication` establishes links to the Window Server and the Display PostScript environment; it then loads the primary nib file for the application and you're off and running.

Resources and Dictionaries

If you are coming from the Mac OS environment, you may think that nib files are resource files. That is partly true, but you should be aware of a significant distinction. Within an application bundle, there is a directory known as `lproj` which contains language-specific files for your application. At run time, `NSApplication` picks the appropriate language file and uses its contents as a dictionary for localization.

Dictionaries are an OpenStep structure that let you associate a keyword (such as an internal programmatic name) with a language-specific string. They are used to localize strings within an application. Since the dictionaries are loaded at run time, you get the advantage of localization using resources without having to recompile your application. (In other environments, the resources remain as external files, but they must be added into the application before it is used.)

About This Part of the Book

As noted previously, the OpenStep classes are arranged here by functionality and by subject area. These represent two of the dimensions along which you can locate each OpenStep class (other dimensions include file structure and kit).

Functionalities

The remaining chapters in this part of the book are devoted to specific functionalities—things that you do with OpenStep:

- **Terminology and Basic Concepts.** This chapter covers the OpenStep basics—objects, actions, events, etc. It then focuses on the main OpenStep object—`NSObject`—both as a class and as a protocol.
- **Building Blocks and the Application Object.** This chapter covers the functionality that OpenStep provides to support common programming needs—generic data types, error handling, and text. Its focus then turns to the most fundamental building block of all OpenStep applications—`NSApplication`.
- **Visualization.** Here you will find the basic information on images and text visualization. The most basic visualization classes—`NSWindow` and `NSView`—are detailed here.
- **Interface Design and Controls.** Menus, commands, and controls are the focus of attention in this chapter—along with OpenStep’s help and assistance classes.
- **Living in a Shared Environment.** One of the most important aspects of Rhapsody is its designed-in support for multitasking, multithreading, and multiprocessing. In a world where so much is going on, you cannot program as if you owned the computer. It’s no longer a matter of good manners—you must plan ahead to share your information and resources. This chapter covers the basic sharing functions—including synchronism and locking, archiving and distribution of objects, and garbage collection.
- **Files and Documents.** OpenStep supports the file system calls you would expect; in addition, this chapter deals with bundles—the file structures that support OpenStep applications (supplanting the resource fork structure the Mac OS programmers have used for years).

- **Interaction, Delegation, and Notification.** In this shared world where processes and threads need to communicate, you need to manage the communication. This chapter covers events, responders, and services which are the tools that OpenStep provides to support these features.
- **Printing.** Most printing is handled using the same routines as are used for visualization; the additional routines are described here.

Subject Areas

Within each chapter, you will find sections relating to different subject areas—for example, text is dealt with in several chapters. Not every chapter has sections for all subject areas.

These are the subject areas that are used in this part of the book:

- **Archiving and Distribution.** This is the mechanism that OpenStep uses for persistent storage of objects—the ability to take an object in memory and write it out to some storage device so that you or someone else can reconstitute it in memory at some time in the future (and possibly on some other machine).
- **Color.**
- **Commands.** Traditional mouse and menu commands are covered within this subject.
- **Controls.** Here you will find the classes that support buttons, rulers, cells, popups, and boxes.
- **Data Types.** Generic and some specific (date and time) data types are discussed in this subject. Note that text and images—which of course are data types—are covered in their own subject areas.
- **Display.** These are the classes that you use to show tables, browsers, cursors, etc. on various devices.
- **Error handling.**

- Images.
- Programming tools. These are the arrays, enumerators, events, views, windows, and objects that form the basics of your environment as a programmer.
- Synchronism. In order to share information and to manage events and mechanisms properly, you need classes to synchronize things—these are covered in this subject.
- Text.

Semantics Tables

The OpenStep API is presented using semantics tables in the following chapters of this section. The documentation from Apple (both on paper and electronic) shows the precise syntax for each class and method call in the supported languages. That documentation also reflects the structure of OpenStep—that is, the kits and files in which the classes are provided.

The semantics tables provide a language-neutral overview of the framework. Methods are shown on the left side of each table, with a description of each method to its right. The arguments of each method are listed in the description; the name of each `argument` is shown in a distinctive typeface. If an argument's type is not `id`, its type is given in parentheses.

Furthermore, information about the use of each method is also provided:

- **Methods** that are commonly used are shown like this.
- *Methods* that you often override are shown like this.
- **Methods** that are both commonly used and are often overridden are shown like this.

Results of methods are listed only if they are not `nil`.

Fabrication

Objects are fabricated in several ways in OpenStep. You will find class methods (factory methods) that allocate uninitialized objects and other factory methods that create and initialize instances of the class (or of its private subclasses). Methods involved in this area are grouped as fabrication methods; while it is not desirable to introduce still more jargon into the mix, the fact is that OpenStep's support of several programming languages means that a number of more mundane terms (creation, construction, etc.) have specific meanings in various of the languages of Rhapsody.

The routines described in the semantics table handle object fabrication in several ways:

- Routines that start with “alloc” (such as alloc or allocZone) are class methods that create an instance object. You are responsible for initializing the object and for releasing it when you are done.
- Routines that start with “init” assume that you have obtained an instance object and that you need to initialize it. These methods are methods of the instance, not of the class.
- Other creation routines are methods of the class and perform allocation and initialization for you; they also mark the objects created as being ready for autorelease. These routines typically have simple English names. For example, the NSDate method `date` returns an allocated and initialized date object. NSDictionary's method `dictionary` returns an allocated and initialized dictionary object. These methods are called factory methods; some of them take various arguments and create different subclasses of the primary object when they return them. You don't have to worry about this.

In order to save space, the rules for alloc and init methods are not repeated for each class.

Accessors Accessors are usually grouped together in the semantics tables. Accessors consist of a pair of methods—one that starts with “set” and another that may adhere to one of several naming conventions.

TABLE 7-1. Sample Semantics Table

<code>setAlternateTitle</code> <code>alternateTitle</code>	The string displayed when the button is in the alternate state (see <code>setTitle</code>). Set: <code>aString</code> (<code>NSString*</code>). Get: returns <code>NSString*</code> .
<code>setPeriodicDelay</code> <code>getPeriodicDelay</code>	After <code>delay</code> (<code>float*</code>) seconds, send messages to the target at <code>interval</code> (<code>float*</code>) periods (also seconds).

As shown in Table 7-1, two general formats are used. In the first, the data that is being accessed is described. If the accessors use different sets of arguments, the Set and Get methods are described separately. Thus, in this case, `setAlternateTitle` (the setter) takes a single parameter—`aString`, of type `NSString*`; `alternateTitle` (the getter) takes no parameter and returns an `NSString*`.

The other format, which is shown for `setPeriodicDelay` and `getPeriodicDelay`, is used when the two accessors use the same arguments. The arguments are listed in the description in the order in which they appear in the method. Thus, you could write either of the following lines of code:

```
float* aDelay;  
float* anInterval;  
  
[setPeriodicDelay:aDelay:anInterval] [classic]  
setPeriodicDelay(aDelay, anInterval) [modern]  
  
[getPeriodicDelay:aDelay:anInterval] [classic]
```

```
getPeriodicDelay(aDelay, anInterval) [modern]
```

**Grouped
Methods**

In addition to accessors, other related methods are sometimes grouped together where they are related and their meanings are clear. If not indicated otherwise, all of the methods grouped together share the same arguments.

**Optional
Arguments**

Some methods are shown as having optional arguments. Two points need to be stressed:

1. In reality, there is no such thing in the OpenStep API. The API itself actually lists each of the alternative configurations (i.e., separate method calls with and without the “optional” arguments). In practice, although you are actually invoking separate messages depending on whether or not you use optional arguments, you can consider this to be a single message with optional arguments.
2. In the rare cases where more than one argument is optional, their ordinal positions matter. Thus, if a method is presented as having two optional arguments, you must supply a value for the first argument if you are going to use the second. Otherwise, the system will not be able to determine that your single argument is meant as the second argument.

Class Methods

Finally, class methods are marked with a †. Class methods (C++ static methods) often are used to create instances of a class. You can call a class method as you would any method of an instantiated object, but you can also call a class method without having an instance around. The `alloc` method, for example, is almost always a class method; it returns an instance whose methods you then call for further processing.

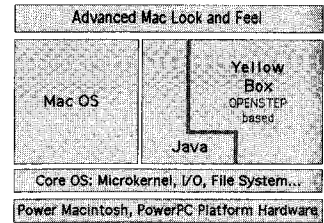
Summary

The balance of this part of the book is devoted to the Rhapsody frameworks, with the bulk of the material being on OpenStep. While the frameworks presented here will be enhanced over time (some of them with significant additions in the early releases of Rhapsody), the architecture is unlikely to change significantly. This is a structure that has stood the test of time; it is several generations beyond the first object-oriented frameworks (Smalltalk, MacApp, and the original NextStep framework).

If you have used other frameworks, you will be struck with how similar OpenStep is to them: after all, graphical user interfaces rely on menus and windows, and there's only so much you can do with them! The differences in these frameworks relate largely to dynamism and to the ubiquity of object-orientedness in the system.

Chapter 8

Terminology and Basic Concepts



Rhapsody and OpenStep use the basic object-oriented terminology and concepts described in Part I and which you may already know. Some terms are used in slightly more specific or nuanced ways than you may know them—this chapter covers the OpenStep/Rhapsody flavors of those terms and provides references to their detailed discussions later in the book.

Programming Design Terminology deals with objects, events, actions, responders, the responder chain, outlets, and connections—the architectural foundations of OpenStep.

Interface Terminology covers specific concepts of OpenStep (cells, panels, browsers, and boxes) with which you may be familiar under other terms.

Central to OpenStep is the NSObject class—the class from which most other classes (including most of your classes) are derived. NSObject is not only an OpenStep class but also an OpenStep protocol. This chapter covers the NSObject class and protocol as part of providing a basis for the chapters that follow.

Programming Design Terminology

Two general concerns apply to the OpenStep terminology in addition to the general principles of object-oriented programming:

1. Certain terms and conventions (services, connections and outlets, for example) make it easier for you, your users, and consultants to use Rhapsody to construct customized applications and solutions.
2. Dynamism in the forms of dynamic typing and binding further expands the flexibility of OpenStep and makes much of Rhapsody's customizable interface possible.

As is often the case, nothing forces you to make your objects usable in this way—you can decree what connections and interfaces exist and freeze them for all time. Likewise, you can ignore dynamic binding and typing, keeping control over your application and how it is used.

When it comes to delivering value to users and customers, it is far better to provide flexibility that allows them to do what they want than to keep things “off limits” so that they don't cause damage. Use the OpenStep error-handling routines and run-time facilities to prevent, detect, and correct errors rather than deluding yourself that by preventing user access to your objects' functionality you will have a more stable system.

Objects

Everything is now an object, and sometimes the meanings blur. At run time, an object is an instance of a class—a structure which has access to that class's methods and which has data values of its own. Other instantiated objects of that class have access to the same methods and have their own data values.

The class itself is the definition or template from which each object is instantiated. In OpenStep, the class is itself a run-time object of type `Class`; that object can instantiate an instance of its class. The instance is of the class. Thus, you might have a class defining an object of type `Barn`. At run-time, an instance object of type `Class` exists for `Barn`. The `Barn Class` object can instantiate instances of `barn`—`myBarn`, `yourBarn`, etc.

Unless otherwise noted, all references to objects are to run-time instantiations of classes—that is, to objects that exist in memory and can send and receive messages. “Class” is used to refer to the programming structures and usually is used in discussing code-writing or compile-time issues. This distinction is one of usage, not one of definition.

Events

If you have programmed for Mac OS, events and the event loop are familiar to you. Events may be generated by user actions (mouse clicks, mouse movements, etc.); they equally can be generated by the system either directly or indirectly in response to user actions or for other reasons. (For example, OpenStep supports the concept of **periodic events**—events that are sent at specific intervals to an application in response to a single request from it. Periodic events may be used by a window during mouse tracking.)

Events in Mac OS are data structures; in OpenStep, they are instances of `NSObject` (for more information, see “NSEvent” on page 392).

If you have used `MacApp`, this may or may not be news to you. Deep inside `MacApp` is a standard event loop in which Mac OS events (data structures)

are retrieved and immediately wrapped in MacApp event objects. You may have been dealing with Mac OS events as objects for years without realizing that at their core they are nonobject data structures.

Actions

Actions have more semantic meaning than events and often correspond to user commands. Cut, copy, paste, capitalize, indent, and page down are typical actions in OpenStep. Events are generated by the operating system in response to a user's interaction with the interface; actions are generated by objects such as windows, views, menus, and controls in response to lower level events.

User actions (moving the mouse, clicking, etc.) often generate events—which may in turn generate actions in the OpenStep sense. Do not confuse user actions (a general interface concept) with the specifically OpenStep concept of actions that are certain types of messages.

Responders

An OpenStep object that can handle events and actions is a responder—a descendant of `NSResponder`. Responders look and behave a lot like MacApp `TEventHandlers` (as well they might). Applications, views, windows, and controls are all descendants of `NSResponder` and therefore can respond to events and actions.

The dispatching of actions differs somewhat from what you may be used to. In frameworks based on static languages like C++ (MacApp, PowerPlant, and ODF), each object that can handle an event (applications, views, windows, and controls) is given a chance to handle every incoming event. Usually this is accomplished in a method such as `DoMenuCommand` in which a gigantic switch statement has a case for each command that the object could handle. Commands that fall through the switch statement encounter a default block, which routes the command on to the next object.

In OpenStep, the process is more direct. Given an action that needs to be responded to, `NSApplication` finds a responder that implements that action by using the `respondToSelector` method that all descendants of `NSObject` have. No longer does the object itself decide if and how it will respond to a command; instead, it doesn't receive the command unless it can respond to it—and it receives it not through a generalized command dispatching routine but as a direct call on its method that will process the command.

For more information on `NSResponder`, see “`NSResponder`” on page 398, and for an overview of `respondToSelector`, see “`NSObject`” on page 135 later in this chapter.

Responder Chain

Just as in other frameworks, events and actions are passed along from one element of the framework to another until one of them handles the request. This is done through a linked list of responder objects—each one points to its next responder or, if there is none, to `nil`.

The framework takes care of following the responder chain through all of its elements. Typically, part of a window (a view or a control) is the **first responder**; if it cannot handle the event or action, it is passed on to its next responder—its superview (the view in which it is contained)—until its window is reached.

At that point, if the event or action has not been handled, the application (`NSApplication` normally manages the responder chain) moves on. There are usually three parts of the responder chain (each of which consists of its own linked list):

1. The OpenStep key window is the window to which all events and actions are directed.
2. The main window is the content window of data currently being worked on. The key window often is the

content window; however, when a floating panel is interposed between the main window and the user, that panel becomes the key window. (This is one more step in the continuing saga of how to manage floating palettes.) The application sends unprocessed events and actions from the key window to the main window.

3. Finally, the application uses its own responder chain to attempt to process the event or action.

As in most other frameworks you may have used, this is a critical part of processing—and one which you can safely ignore most of the time. It is necessary for you to set up the first responder in a view hierarchy; thereafter, the framework will manage the responder chain in most cases.

The choice of “responder chain” rather than “target chain” can be seen as more than merely semantic. In frameworks built on static languages, events and actions are sent to the elements of the chain. With its dynamic structure and the possibility of run-time introspection of objects, OpenStep can instead query the elements of the chain and send the event or action only to the object that will respond to it. The difference is subtle and general (there are cases where optimizations and structural concerns blur the architecture), but it is part of the flavor of OpenStep. Furthermore, this type of design is what makes the dynamic interface possible.

Delegates

As the penultimate step in each part of the responder chain (key window, main window, application), a delegate of the window or application is given a chance to handle the event or action. The delegate—which must be an `NSObject`—need not be a descendant of `NSResponder`. The window or application has a chance to forward the action or event to any object that it chooses (that presumably can respond to the event or action). If that, too, fails, then the ultimate step in each part of the responder chain is to pass on to the next part—from the key window to the main window to the application.

The use of a delegate is a very important part of the OpenStep architecture. For your application object (or a window or a view) to implement some functionality (responding to a message or a method call), it need only have a delegate that can do so. In other architectures, the application object (or window or view) would need to be subclassed to add the appropriate message or method call.

Outlets

An object often needs a reference to other objects with which it has to deal: a view may need to be able to communicate with its window, two views may need to mutually update data fields, etc. An outlet is just such a reference. It is of type id (not a specific class type) and lives in an instance variable. Interface Builder can find such instance variables in your class definition and allow them to be manipulated by you or your users.

You can always hard-wire your outlets and hide them from view by declaring them as a type other than id so that Interface Builder won't see them. This may make your code more secure, but it will make it less flexible and easy to use.

With your outlets visible to Interface Builder, you can then connect them to graphical elements in your interface using connections.

Connections

You connect outlets to interface objects—a data entry field called “city” might be an outlet that is connected to a specific text field in your interface. Since Interface Builder can recognize your outlets, it can display the list of outlets for your object, allowing you to connect them to interface elements. Again, this differs from static language-based frameworks in which there is always a little snippet of code that you have to execute at run time to link the text field to the instance variable “city.”

Note in this example that the outlet—a text field that is referred to by the outlet variable “city”—is in fact a data entry field. “Outlet” is a messaging concept and doesn’t indicate the direction in which data flows in your application.

Connections are also made for actions—and here direction of data flow is critical. Action connections are made from control objects (such as buttons) to the objects that you create. Actions not only have a source and destination but also have a message that corresponds to a method of the receiving object. Thus, you can create an action connection from a button to an object you create and have it process the appropriate message of your object.

Just as OpenStep relies on instance variables of type `id` to be outlets, it relies on methods of the form

```
(void*) actionName: (id) sender [classic]
(void*) actionName (id sender) [modern]
```

to find your object’s actions. It generates a list of possible actions that your object can perform by compiling all of the action names into a list.

Interface Terminology

Certain elements of the OpenStep interface have names that may be new to you—or which you may know in other concepts. Cells, panels, browsers, and boxes are described later in the book; their basic concepts are covered here.

Cells

Cells are relatively lightweight objects that can display text or images. They can also manage mouse tracking, selection, menus, actions, keyboard events, and the cursor. In this they are like responders in general and views in particular.

However, cells are not responders or views—they descend directly from `NSObject`. Controls (descendants of `NSControl` such as `NSButton`) often have a cell associated with them to handle these generic activities. `NSControl` does descend from `NSView` and thus from `NSResponder`; it participates in the responder chain as appropriate and often passes off the processing of a specific message to an associated `NSCell`.

The problem of providing lightweight view-like objects to display text and images has evolved over time in all object-oriented frameworks. Clearly the text field that displays the label for a data entry field requires less functionality than the text field into which you can enter data.

For more information on cells, see “`NSCell` Fabrication Methods” on page 218.

Box

Boxes are special-purpose views that can draw a border around themselves and can display a title. They are typically used to group other views (such as a cluster of buttons).

Boxes are discussed in “`NSBox`” on page 215.

Browsers

Forget the Internet for a moment (if you can). OpenStep browsers are scrolling hierarchical windows which let you quickly move through a hierarchy (usually of files). At the top, a separate panel (often called a shelf) lets you store frequently used icons. This type of browser is an integral part of Smalltalk. It was a standard part of the OpenStep environment and is evolving for Rhapsody.

When you incorporate a browser into your application, you leave all of the display (scrolling, collapsing, etc.) to it: you provide a companion object which provides the contents of each cell that the browser asks you to display, and that’s it. (This architecture is used in a number of places within Rhapsody, including menus. The general design and functionality of the interface element is not your concern; instead you con-

centrate on simply providing the contents of the cell or menu item.)

Panels

Panels are a special case of windows. They often float in front of the application's windows and are used as palettes, alerts (attention panels), and other subsidiary windows. They can be key windows, but they can never be main windows.

Panels are described in "Panels" on page 301.

Types

Several types are used repeatedly in the OpenStep framework; others are specific to certain classes. These are the types that will be referred to frequently in the following chapters.

NSRange

An NSRange is a simple structure of two unsigned ints. The first (`location`) identifies the starting location in an array or string; the second (`length`) indicates the number of items to be considered in the range. The location value is zero-relative; the length value can be zero.

NSComparison-Result

Comparisons return one of three values for NSComparison-Result:

- NSOrderedAscending
- NSOrderedSame
- NSOrderedDescending

When two items are compared, if the first (on the left) is of lesser value than the second (on the right), NSOrderedAscending is returned.

NSObject

NSObject and NSProxy are the the basic objects from which all other objects descend—both those that are implemented in OpenStep itself and those that you create. This section describes NSObject in detail. In addition to the NSObject class, there is an NSObject protocol which is also discussed here. NSProxy, which supports objects that do not exist (such as objects that have not yet been instantiated) as well as distributed objects, is discussed in the next section.

Architectural Overview

The issue of a common base object from which all other objects descend has been debated for a long time in the object-oriented community. The main disadvantage of a common ancestor is that a change to its structure requires recompilation of all objects that descend from it—the fragile base class problem. Of course, this argument loses most of its potency in a world where the fragile base class problem has been wrestled into submission (such as the dynamic world of OpenStep).

The argument in favor of a common base object is that all objects in the environment share certain characteristics and that you can rely on any object that you use being able to perform certain tasks. Although static languages such as C++ let you inquire as to an object's class and dynamic languages let you inquire in more detail not only about the object's class but also about the methods that it implements, still it is easier and more efficient to know that a certain set of tasks can be performed by any object with a certain ancestor.

This persistent problem is partly caused by the layering of object-oriented frameworks on top of operating systems. There is a critical barrier between the world of objects and the world of nonobject processing. In almost all cases, hardware does not deal with objects—in fact, it rarely even deals with such sophisticated concepts as data structures. It just moves bits and bytes around.

As you move from the hardware to the operating system, the bits and bytes are aggregated into data structures (strings, colors, files, etc.). Then, in what is traditionally called the application layer, a framework takes the data structures and encapsulates them into objects with which it works.

If you refer to the diagram that is reproduced on the back cover and on the chapter headings of this book, you will notice that the traditional layered structure—hardware, operating system, applications—has been changed in Rhapsody. OpenStep—the object-oriented framework—lies directly on top of the Core OS. The conversion from nonobject data structures to objects occurs at a fairly primitive level in the design. In traditional architectures, a framework lives at a much higher level—in the Mac OS Compatibility (Blue Box) section of the diagram, MacApp, for example, exists as a layer at the *top* of the Blue Box, with Mac OS data structures existing below it and being converted to objects only for the specific needs of MacApp applications.

The point here is that by providing a common object OpenStep doesn't cause problems such as arise with the fragile base class; furthermore, since the OpenStep objects are so pervasive in the system, the features of NSObject can be considered not so much as framework features but as OS features. In other words, object-orientedness encompasses all of what previously would have been considered an application layer and reaches all the way to the operating system.

NSObject Functionalities

NSObject—through its class and its protocol—provides basic functionalities shared by all objects in the framework. These functionalities fall into the following categories:

- They provide means to initialize the class and then to create and destroy instances of it. (This process is referred to generically as fabrication.)

- Methods provide for introspection of the object—what class and superclass it belongs to, what messages it responds to, etc.
- Identification of a specific object—is this object the same as that one, for example—is supported. (Identification differs from introspection in that introspection deals with the attributes and characteristics of the object whereas identification deals with recognizing a specific object.)
- Message handling is provided by NSObject, chiefly using the perform method.
- Garbage collection and memory management are also provided.
- NSObjects can be archived and unarchived; this is done with their own methods and also by using the NSCoder protocol which NSObject adopts.
- Finally, error handling is supported by NSObject.

Each of these functions is described in more detail in the following.

Note that the NSObject class adopts the NSObject protocol; as a result, the methods of the NSObject protocol are available to all descendants of the NSObject class. In all other cases in this book protocols are discussed separately from classes; here the two are lumped together. Methods in the NSObject protocol are identified with [P].

Fabrication Table 8-1 lists the basic NSObject fabrication methods. Remember that allocation and initialization are used together when you manually create an object. You normally use a method like new or a factory methods that directly creates (allocates and initializes) an instance of the class required.

TABLE 8-1. NSObject Fabrication Methods

<code>+initialize</code>	This initializes the class; OpenStep calls it once for each class automatically.
<code>+alloc</code> <code>+allocWithZone</code> <code>init</code>	
<code>+new</code>	By default calls alloc and init; override it if your class needs to do additional processing at that time.
<code>dealloc</code>	Called from a release method. You override this method if you have data structures to release or if you have subsidiary objects to release.

Introspection Particularly in Rhapsody's dynamic environment, it is important that objects be able to examine themselves. The methods shown in Table 8-2 primarily describe the class and object characteristics, rather than the content of the objects.

Identification Identification routines (shown in Table 8-3) let you compare and identify objects based on their content.

Message Handling While all object-oriented languages and frameworks can handle messages (whether they are called messages or function calls), one of the very significant advantages of OpenStep and its dynamic nature is that it can reroute messages on the fly. The previously noted method respondsToSelector (see Table 8-2, NSObject Introspection Methods) lets you find out if an object can handle a message.

If an object receives a message to which it cannot reply (that is, it is asked to perform a method or function that it does not have), it is then sent a new message—forwardInvocation. The original message is wrapped in an NSInvocation* object; you

TABLE 8-2. NSObject Introspection Methods

<code>+class</code> <code>class [P]</code>	Returns the Class object.
<code>+superclass</code> <code>superclass [P]</code>	Returns the id of the superclass's class.
<code>isKindOfClass [P]</code>	Returns a BOOL indicating if this is an instance of aClass (Class) or one of its superclasses.
<code>isMemberOfClass [P]</code>	Returns a BOOL indicating if this is an instance of aClass (Class); superclasses don't count.
<code>+conformsToProtocol</code> <code>conformsToProtocol [P]</code>	Returns a BOOL indicating if this class conforms to aProtocol (Protocol*).
<code>respondsToSelector [P]</code> <code>+instancesRespondToSelector</code>	Returns a BOOL indicating if this class or its instances can respond to aSelector (SEL).
<code>methodForSelector</code> <code>+instanceMethodForSelector</code>	Returns the address (IMP—a pointer to a function that returns an id) of the implementation of aSelector (SEL) for the class or an instance. This lets you access it as a function, rather than as an object's method. Used internally.
<code>methodSignatureForSelector</code>	Returns an NSMethodSignature* for aSelector (SEL). Used internally.
<code>+description</code>	Returns an NSString*; the default implementation returns the class name in the string. You can override it to provide additional information to debuggers.

TABLE 8-3. NSObject Identification Methods

<code>isEqual</code>	Returns a BOOL indicating if <code>anObject</code> is equal to this one; equality is determined by looking at the objects' ids.
<code>hash</code>	Returns an unsigned int that you can use to store and retrieve an object in a hash table.
<code>self [P]</code>	Returns itself. Other frameworks and languages use "this."
<code>zone</code>	Returns an <code>NSZone*</code> ; if not specified at allocation time, it is the default zone.
<code>isProxy</code>	Returns a BOOL if this is a proxy object standing in for an NSObject.

can extract the message to use in a call to `respondToSelector` for another object; if you find an object that can respond, you can redirect the message. Table 8-4 contains the NSObject messaging methods.

TABLE 8-4. NSObject Messaging Methods

<code>forwardInvocation</code>	You normally don't call <code>forwardInvocation</code> ; instead you implement it to examine the single argument, <code>anInvocation</code> (<code>NSInvocation*</code>) and decide what to do with it. This method is called after OpenStep has already determined that you cannot process the message directly.
--------------------------------	---

TABLE 8-4. NSObject Messaging Methods (Continued)
performSelector [P]

The `aSelector` (SEL) argument is passed as a message to the object. If you know in advance which method you want to execute, you can call it directly rather than calling it via a call to `perform`. However, for dynamic processing, you can determine the method you want to call at run time.

One or two arguments (`anObject` and `anotherObject`) may follow `aSelector`; they are passed to the method as arguments.

Garbage Collection and Memory Management Routine memory management and garbage collection are important parts of modern frameworks and languages. Too much time has been spent tracking down memory leaks and mysterious crashes from data structures that were not allocated when necessary—or were pulled out from underneath code that assumed they were there.

The process is made as simple as possible. When you want to make certain an object is around, you send the `retain` message; you balance each `retain` message with a `release` message. For example, the `NSArray` object allows you to put objects into an array that can then be manipulated with its searching and sorting methods. `NSArray` calls `retain` for each object it adds to its array; it calls `release` when it removes objects from the array. In this way, an object in an `NSArray` can never accidentally be deallocated while it is in the array.

Autorelease pools provide an even easier way to manage objects. Here, you send the `autorelease` message to an object instead of a `retain` message; it is automatically added to the

current autorelease pool—which is merely a collection of objects that can be released at a given time. When the pool itself is released, each object in the pool is sent a release message—you don’t have to balance retain and release calls.

A single autorelease pool exists for the life of your application, although you can create others (and usually do for each thread in a multi-threaded application). Since balancing retain and release calls is a potentially time-consuming programming chore, using autorelease is a valuable tool. The disadvantage of autorelease is that the release messages are not sent until the pool is deallocated. Thus, if you have an object that you want to retain and could release almost immediately, using autorelease will keep it around until the current autorelease pool is deallocated.

The actual mechanism that is used is a reference count in each object that is incremented by one for each retain call and decremented by one for each release call. When the count falls to zero, the object is deallocated.

TABLE 8-5. NSObject Garbage Collection and Memory Management Methods

retain [P]	Balance calls to retain and release as described previously
release [P]	
retainCount [P]	Returns an unsigned int which is the reference count for the object. Unless you are reimplementing memory management, don’t call or override this method!
autorelease [P]	

Get in the habit of using these tools (either retain and release or autorelease). Code that works correctly in a single-user en-

vironment has a nasty habit of exploding as it is modified and enhanced to work in a shared environment. As with error handling, retrofitting of memory management is very expensive and prone to errors.

Table 8-5 shows the NSObject garbage collection and memory management methods.

Archiving NSObject provides basic support for archiving and distribution. (Archiving refers to placing representations of objects on disk; distribution refers to copying objects to other address spaces.)

Archiving relies on NSCoder objects which are discussed in “NSCoder” on page 322. Table 8-6 shows the archiving methods of NSObject.

TABLE 8-6. NSObject Archiving Methods

<code>classForArchiver</code> <code>classForCoder</code>	Return a Class object to be used during archiving or coding. The default is the object’s own Class.
<code>replacementObjectForArchiver</code>	Returns an id of an alternative object to archive by anArchiver (NSArchiver*). This allows you to read and write less complex objects than your run-time objects.
<code>awakeAfterUsingCoder</code>	This is the counterpart of the replacement Object methods. Its single argument aCoder (NSCoder*) has just created the object; you can substitute another.

TABLE 8-6. NSObject Archiving Methods (Continued)

<code>replacementObjectForCoder</code>	Returns an id of an alternative object to encode by <code>encoder</code> (<code>NSCoder*</code>). This allows you to read and write less complex objects than your run-time objects.
<code>±setVersion</code> <code>±version</code>	You can provide a version for objects that are archived; this helps you manage change in your software. Set: <code>anInt (int)</code> . Get: returns an <code>int</code> .

Error Handling NSObject itself has only one method to directly handle errors—`doesNotRecognizeSelector`. It is called by the run-time environment when a message sent to the object cannot be processed.

Normally `doesNotRecognizeSelector` is called as the only statement in the default `forwardInvocation` method and an error is generated. You override `forwardInvocation` to attempt to process the message before `doesNotRecognizeSelector` is called. The method is described in Table 8-7.

TABLE 8-7. NSObject Error Handling Method

<code>doesNotRecognizeSelector</code>	The single argument <code>aSelector</code> (<code>SEL</code>) is the unrecognized selector.
---------------------------------------	---

Dynamism The `load` and `poseAsClass` methods help to implement dynamism in NSObjects. Note that `poseAsClass` is another way to accomplish what you can do with category

definitions. See Table 8-8 for the NSObject dynamism methods.

TABLE 8-8. NSObject Dynamism Methods

<code>load</code>	Called automatically to load classes into the program.
<code>poseAsClass</code>	<p>You can dynamically make your class pose as one of its superclasses passed in the <code>aClassObject</code> (Class) argument.</p> <p>This allows you to modify a class's behavior by creating a subclass of it and having that class pose as the original class.</p>

NSInvocation

This is the small object that wraps a message that cannot be processed. Its primary methods are shown in Table 8-9. Note that messages being forwarded must have a fixed number of arguments; some methods (such as `NSArray`'s `initWithObjects`) that use variable numbers of arguments are not allowed.

TABLE 8-9. NSInvocation Methods

<code>selector</code>	Returns the message that was attempted as a SEL.
<code>setTarget</code>	Sets the next destination for the invocation.
<code>dispatch</code>	Sends the message to the target.

NSProxy

NSProxy, like NSObject, is a base object or root class. It adopts the NSObject protocol, so it provides all of the functions of introspection (`isKindOfClass`, `isMemberOfClass`, etc.), messaging (`performSelector`, `respondToSelector`, etc.), and memory management (`retain`, `release`, `autorelease`, etc.) that are included in that protocol.

On its own, NSProxy has only seven methods. In addition to the standard allocation and deallocation methods, it has the `forwardInvocation` and `methodSignatureForSelector` methods of NSObject. These allow NSProxy to do its work. When an `NSInvocation` is received to which it cannot respond (and that is any invocation), it passes it on to the object it is standing in for. At that moment, it may be necessary for that object to be instantiated, or for a link to an object somewhere else to be established. The methods of NSProxy are shown in Table 8-10.

TABLE 8-10. NSProxy Methods

<code>+alloc</code>	
<code>+allocWithZone</code>	
<code>dealloc</code>	Called from a release method. You override this method if you have data structures to release or if you have subsidiary objects to release.
<code>+class</code>	Returns the Class object.
<code>forwardInvocation</code>	You pass an <code>Invocation</code> (<code>NSInvocation</code> *) on to the object that the NSProxy is representing—possibly instantiating that object at the time.

TABLE 8-10. NSProxy Methods (Continued)

<code>methodSignatureForSelector</code>	You override this method to return the <code>NSMethodSignature*</code> for <code>aSelector</code> (SEL). Normally, you call the <code>methodSignatureForSelector</code> method of the object that the <code>NSProxy</code> is representing—possibly instantiating that object at the time.
<code>description</code>	Returns an <code>NSString*</code> ; the default implementation returns the class name in the string. You can override it to provide additional information to debuggers.

Note that since `NSProxy` is a base class and not a descendant of `NSObject`, it only has the `NSObject` protocol methods available.

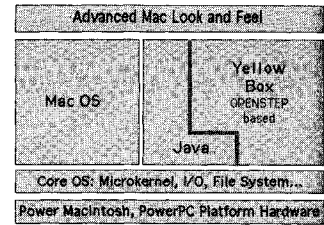
Summary

This chapter has presented some of the OpenStep architectural issues; it also includes a complete description of the `NSObject` class and protocol—the basis for all other objects in the framework.

The next chapter starts with the basic building blocks of the framework—arrays, dates, strings, etc.—and proceeds on to the basic run-time object of the framework, the `NSApplication` object.

Chapter 9

Building Blocks and the Application Object



This chapter covers the building blocks of OpenStep. They range from programmers' tools (such as arrays and numbers) all the way to the application itself—`NSApplication`.

The basic classes and methods are similar to those found in other frameworks; what may be new to you is the degree to which objects are used to support common programming tasks. In Java, everything is an object, so Java programmers are not surprised to see integers as objects. For others, this may be new. Relax: it's not at all complicated and makes your life easier (particularly when debugging code).

Nothing requires you to use some of the objects here—you can certainly create traditional arrays and loop through them with your

own counters; however, most people find these objects incredibly useful. If you haven't used objects for these common programming tasks before, give it a try: you'll save a lot of time and energy.

The highlights of this chapter are:

- *Data types—the wrappers for byte streams and C data types as well as support for dates (page 150)*
- *Error handling—support for assertions and exceptions (page 167)*
- *Programming tools—arrays and enumerators (page 171)*
- *Text tools—dictionaries, character sets, strings, and scanners (page 181)*
- *OpenStep applications—the NSApplication class and all of its methods (page 192)*

Data Types

OpenStep provides support for data types in general as well as for specific data types that you may need to use (such as dates and times). The two primary classes in this area are NSData and NSValue. NSData wraps simple data—integers, floats, etc. without regard to its type; NSData simply wraps bytes. NSValue wraps pointers and other C or Objective-C structures and does have an awareness of the meaning of the data that it wraps.

Strings are a different kettle of fish; they are discussed in “Text” on page 181.

Simple Data

Data objects contain a certain number of bytes. They are untyped, although the methods for creating and initializing them from specific types may mislead you (those methods are

for convenience—the type information is not preserved in the object).

Data objects are most often used when you are communicating with an external application or environment. Within the OpenStep world, it is preferable to use objects; where possible within Rhapsody, try to use data structures or typed data in preference to simple byte streams.

NSData is the primary class; NSMutableData provides additional functionality for data objects that need to be modified.

NSData The NSData class simply wraps bytes. It is used to read and write data from and to files; it can also be used to communicate among different applications that need to share data.

TABLE 9-1. NSData Fabrication Methods

<code>+allocWithZone</code>	
<code>+data</code>	Creates a data object.
<code>+dataWithBytes</code> <code>+dataWithBytesNoCopy</code>	Creates a data object from a buffer. The argument <code>bytes</code> is a pointer to the buffer; the second argument, <code>length</code> , is the number of bytes to use. The first method copies the data; the second uses the original.
<code>+dataWithContentsOfFile</code> <code>+dataWithContentsOfFileMappedFile</code>	Creates a data object with the entire contents of the file specified in the <code>path</code> argument of type <code>NSString*</code> .
<code>initWithBytes</code> <code>initWithBytesNoCopy</code>	Same as <code>dataWithBytes</code> and <code>dataWithBytesNoCopy</code> , but you must have created the data object yourself.

TABLE 9-1. NSData Fabrication Methods (Continued)

<code>initWithContentsOfFile</code> <code>initWithContentsOfMappedFile</code>	Same as <code>dataWithContentsOfFile</code> and <code>dataWithContentsOfMappedFile</code> , but you must have created the data object yourself.
<code>initWithData</code>	Initializes a data object from another; the single argument <code>data</code> is of type <code>NSData</code> .

The accessors of `NSData` objects deal only with bytes: data objects know nothing of the type of their contents. The accessors are shown in Table 9-2.

TABLE 9-2. NSData Accessor Methods

<code>bytes</code>	Returns a read-only pointer (const void*) to the data.
<code>description</code>	Returns an <code>NSString</code> object with a hex representation of the data.
<code>getBytes</code>	<p>The pointer argument <code>buffer</code> is filled with a copy of the data.</p> <p>With the second argument <code>length</code> (unsigned), only that number of bytes is copied.</p> <p>With the second argument <code>range</code> (<code>NSRange</code>), only specific bytes are copied.</p>

TABLE 9-2. NSData Accessor Methods (Continued)

<code>subdataWithRange</code>	Similarly to <code>getBytes</code> with a range, this method returns a new <code>NSDataObject</code> with a copy of the data selected (rather than returning the data in the buffer pointer of <code>getBytes</code>). The single argument is an <code>NSRange</code> <code>range</code> .
-------------------------------	---

`NSData` also contains utility methods for comparisons and for writing to files as shown in Table 9-3.

TABLE 9-3. NSData Utility Methods

<code>isEqual</code>	Returns a <code>BOOL</code> indicating if the data object is equal to <code>anObject</code> , the single argument of any class.
<code>isEqualToData</code>	Returns a <code>BOOL</code> indicating if the data object is equal to the <code>NSData</code> object passed in to the argument <code>other</code> .
<code>hash</code>	This override of the <code>NSObject</code> protocol method returns an unsigned int which is the length of the data object.
<code>length</code>	Returns the number of bytes (unsigned) in the object.

TABLE 9-3. NSData Utility Methods (Continued)

<code>writeToFile</code>	The first argument, <code>path</code> , is an <code>NSString</code> specifying the file to use. The second argument, <code>atomically</code> , is a <code>BOOL</code> indicating if <code>true</code> that the data should be written first to a backup file and then renamed. The method returns a <code>BOOL</code> indicating the result of the process.
--------------------------	---

NSMutableData These objects allow you to modify their data. Their methods fall into two groups: initialization and destruction and data manipulation.

Initialization and destruction methods are shown in Table 9-4.

TABLE 9-4. NSMutableData Fabrication Methods

<code>mallocWithZone</code>	
<code>initWithCapacity</code>	Creates an <code>NSMutableData</code> object with space for <code>aNumItems</code> (unsigned). The object can later be expanded.
<code>initWithLength</code>	Creates an object of <code>length</code> (unsigned) bytes. It is marked autorelease and is zero-filled.
<code>initWithCapacity</code>	Initializes an <code>NSMutableData</code> object to <code>capacity</code> (unsigned) bytes.
<code>initWithLength</code>	Initializes an <code>NSMutableData</code> object to <code>length</code> (unsigned) bytes. It zero fills the object to that length.

The data manipulation routines allow you to access the object's data directly as well as to change its size as shown in Table 9-5.

TABLE 9-5. NSMutableData Utility Methods

<code>increaseLengthBy</code>	Adds to the length of an NSMutableData object by <code>extraLength</code> (unsigned).
<code>setLength</code>	Changes the length of an NSMutableData object to the argument <code>length</code> (unsigned). If necessary, new bytes are zero filled.
<code>mutableBytes</code>	Returns a pointer (void*) to the object's data.
<code>appendBytes</code>	Appends <code>bytes</code> (const void*) of <code>length</code> (unsigned) to the data object.
<code>appendData</code>	Appends the contents of <code>other</code> (NSData*) to this data object.
<code>replaceBytesInRange</code>	Replaces <code>range</code> (NSRange) within the data object by <code>bytes</code> (const void*).
<code>resetBytesInRange</code>	Fills <code>range</code> (NSRange) with zeroes.

Pointers and Typed Data

Unlike the simple byte stream of NSData objects, NSValue objects contain values and an associated type (from the C language). Whereas the NSData objects are most often used to communicate with other environments, NSValue objects are used within the OpenStep environment—often to wrap data to be placed in NSArray's. (One of the criticisms of MacApp's TList classes was the fact that they originally could contain

only descendants of TObject; later they were modified to be able to contain some simple types.)

NSValue The NSValue class itself contains methods for initializing objects and accessing the data (Table 9-6).

TABLE 9-6. NSValue Methods

<code>+value</code>	Return a new NSValue object. The two parameters: <code>value</code> (const void*) and <code>type</code> (const char*) are a pointer to the data itself and the Objective-C type of that data.
<code>+valueWithNonretainedObject</code>	Creates a new NSValue object containing the object <code>anObject</code> which is the single argument stored as a pointer.
<code>+valueWithPointer</code>	Creates the object containing <code>pointer</code> (void*) which is the sole argument.
<code>getValue</code>	This accessor has a single argument <code>value</code> (void*) which returns with a copy of the data in the NSValue.
<code>objCType</code>	Returns as a const char* the type of the data in the NSValue object. Together <code>getValue</code> and <code>objCType</code> return all of the information about the NSValue's data.
<code>isEqual</code>	Returns a BOOL indicating whether the object passed in the single argument <code>aValue</code> is equal to this object.

TABLE 9-6. NSValue Methods (Continued)

<code>pointerValue</code>	Returns a <code>void*</code> to the value in the object.
---------------------------	--

NSNumber Unless you are wrapping a pointer inside an NSValue object, you will usually use the NSValue descendant NSNumber using the methods shown in Table 9-7.

TABLE 9-7. NSNumber Methods

```

+numberWithBool
+numberWithChar
+numberWithDouble
+numberWithFloat
+numberWithInt
+numberWithLong
+numberWithLongLong
+numberWithShort
+numberWithUnsignedChar
+numberWithUnsignedInt
+numberWithUnsignedLong
+numberWithUnsigned
    LongLong
+numberWithUnsignedShort

```

These methods (of NSNumber class) each create and return an instance of NSNumber. Each takes a single value argument, which is of the type in the method name. Thus, `numberWithBool` takes an argument of type Boolean.

```

initWithBool
initWithChar
initWithDouble
initWithFloat
initWithInt
initWithLong
initWithLongLong
initWithShort
initWithUnsignedChar
initWithUnsignedInt
initWithUnsignedLong
initWithUnsignedLongLong
initWithUnsignedShort

```

These methods (of NSNumber instances) are called after you have created the NSNumber object. Each takes a single value argument which is used to set the object's value.

TABLE 9-7. NSNumber Methods (Continued)

<code>boolValue</code> <code>charValue</code> <code>doubleValue</code> <code>floatValue</code> <code>intValue</code> <code>longLongValue</code> <code>longValue</code> <code>shortValue</code> <code>stringValue</code> <code>unsignedCharValue</code> <code>unsignedIntValue</code> <code>unsignedLongLongValue</code> <code>unsignedLongValue</code> <code>unsignedShortValue</code>	These instance functions return the value of the NSNumber object in the type requested. They take no arguments.
<code>compare</code>	Takes a single argument <code>other</code> of type NSNumber and returns an <code>NSComparisonResult</code> . If the argument is not an NSNumber, <code>NSOrderedDescending</code> is returned.
<code>isEqual</code>	Takes a single argument <code>value</code> of any type; if it is an NSNumber, the <code>compare</code> method is called. The result is a <code>BOOL</code> .

NSDecimalNumber This descendant of NSNumber provides decimal arithmetic functionality; it includes error handling through the use of `NSDecimalNumberHandlers` which adopt the `NSDecimalNumberBehaviors` protocol. `NSDecimalNumber` methods are shown in Table 9-8.

TABLE 9-8. NSDecimalNumber Methods

<code>+decimalNumberWithDecimal</code>	Creates and returns an <code>NSDecimalNumber</code> from <code>decimal</code> (<code>NSDecimal</code>).
<code>+decimalNumberWithString</code>	Creates and returns an <code>NSDecimalNumber</code> from <code>numericString</code> (<code>NSString*</code>); the optional second argument <code>locale</code> (<code>NSDictionary*</code>) allows you to customize the <code>NSDecimalSeparator</code> (commas and periods for thousands).
<code>+decimalNumberWithMantissa</code>	Creates and returns an <code>NSDecimalNumber</code> from <code>mantissa</code> (unsigned long long), <code>exponent</code> (short), and <code>isNegative</code> (<code>BOOL</code>).
<code>+one</code> <code>+zero</code> <code>+notANumber</code>	These functions return <code>NSDecimal</code> numbers with the specified values.
<code>initWithDecimal</code> <code>initWithString</code> <code>initwithMantissa</code>	See <code>decimalNumberWithDecimal</code> , <code>decimalNumberWithString</code> , and <code>decimalNumberWithMantissa</code> .
<code>decimalNumberByAdding</code> <code>decimalNumberByDividingBy</code> <code>decimalNumberByMultiplyingBy</code> <code>decimalNumberBySubtracting</code>	Creates and returns a new <code>NSDecimal</code> number which is the result of performing the operation described on the original <code>NSDecimalNumber</code> , using the argument <code>decimalNumber</code> (<code>NSDecimalNumber*</code>). The optional second parameter behavior (<code>id<NSDecimalNumberBehaviors></code>) allows you to modify the default handling of errors, rounding, etc.

TABLE 9-8. NSDecimalNumber Methods (Continued)

<code>decimalNumberByRaisingToPower</code>	Performs the operation using the argument <code>power</code> (short). The optional second parameter <code>behavior</code> (<code>id<NSDecimalNumberBehaviors></code>) can be used as above.
<code>decimalNumberByMultiplyingByPowerOf10</code>	
<code>decimalNumberByRoundingAccordingToBehavior</code>	Applies <code>behavior</code> (<code>id<NSDecimalNumberBehaviors></code>) and returns a new <code>NSDecimalNumber</code>
<code>decimalValue</code>	Returns an <code>NSDecimal</code> —struct, not an object.
<code>+setDefaultBehavior</code> <code>+defaultBehavior</code>	Rather than passing in an <code>NSDecimalNumberBehaviors</code> to a method above, you can set the default behavior. Set: <code>behavior</code> (<code>id<NSDecimalNumberBehaviors></code>) Get: returns (<code>id<NSDecimalNumberBehaviors></code>)

Dates and Times

The `NSDate` class provides basic date manipulation and storage routines. In general, it is always better to use customized data types (such as `NSDate`) than amorphous types (strings, integers, etc.) that you manipulate as dates or other specialized data types. Certainly, as organizations confront the Year 2000 Problem, the benefits of relying on “smarter” data types (like `NSDate`) over “dumber” data types (like strings) are clear.

Three specific dates are important when dealing with date routines:

- The OpenStep reference date of January 1, 2001 is used through the system. Dates are represented by seconds (positive or negative) from that reference date.

- January 1, 1970 is the UNIX reference date against which the `time_t` function in UNIX returns a number of seconds to a given date.
- January 1, 1904 is the reference date for the Mac OS. The original Mac OS date routines represent dates until the year 2013; the longer dates introduced in System 7 represent a broader range; time zero is still January 1, 1904. (This date eliminates the confusion over leap years in century years. 1900 was not a leap year; 2000 is.)

The number of seconds in a year is equal to the number of days (365 or 366) multiplied by the number of seconds in a day (86,400). Because of the unevenness of the Earth's orbit, occasionally leap seconds are added to a year. This keeps the atomic clocks running properly, but leap seconds have no place in computer timekeeping. Stick to 86,400 times the number of days in the year.

NSDate Objects of this class (actually instantiations of private subclasses of this class) represent individual dates. The methods of `NSDate` allow you to compare date objects, to find the length of time (days as well as time) between two objects, and to format them according to user preferences. The `NSDate` fabrication methods are shown in Table 9-9.

TABLE 9-9. NSDate Fabrication Methods

`+allocWithZone`

`+date`

Creates and returns a date object for the current time.

TABLE 9-9. NSDate Fabrication Methods (Continued)

<code>+[NSDate dateWithTimeIntervalSinceNow:]</code> <code>+[NSDate dateWithTimeIntervalSinceReferenceDate:]</code> <code>+[NSDate dateWithTimeIntervalSince1970:]</code>	These routines take a single argument <code>seconds</code> (<code>NSTimeInterval</code>) which is the offset (positive or negative) from specific dates. They return the appropriate date object.
<code>+[NSDate dateWithNaturalLanguageString:]</code>	Parses <code>string</code> (<code>NSString*</code>) and returns the appropriate date object. Rather than use the user defaults, you can pass <code>localeDictionary</code> (<code>NSDictionary*</code>) as the optional second argument.
<code>+[NSDate distantFuture]</code> <code>+[NSDate distantPast]</code>	These routines return valid date objects for use in control loops, etc. They are safer than using 0 as a date.
<code>+[NSDate initWithTimeIntervalSinceNow:]</code> <code>+[NSDate initWithTimeIntervalSinceReferenceDate:]</code>	For already created date objects, same as <code>dateWithTimeInterval...</code>
<code>+[NSDate initWithString:]</code>	Similar to <code>dateWithNaturalLanguageString</code> , but accepts only the single argument <code>description</code> (<code>NSString*</code>).
<code>+[NSDate initWithTimeInterval:]</code>	Initializes a date object to <code>seconds</code> (<code>NSTimeInterval</code>) after another date (<code>NSDate*</code>).
<code>+[NSDate init]</code>	Unless overridden, calls <code>initWithTimeIntervalSinceReferenceDate</code> .
<code>+[NSDate addTimeInterval:]</code>	Adds <code>seconds</code> (<code>NSTimeInterval</code>) to itself and returns the result as an id.

Table 9-10 shows the NSDate utility methods that you use as needed. For details on conversion specifiers in the format strings for the final methods, refer to the standard C strftime function.

TABLE 9-10. NSDate Utility Methods

<code>isEqualToDate</code>	Compares the date to another-Date (NSDate*) and returns a BOOL.
<code>earlierDate</code> <code>laterDate</code>	Compares the date to another-Date (NSDate*) and returns the earlier or later date as an NSDate*.
<code>compare</code>	Returns NSComparisonResult based on this date and the date passed into anotherDate (NSDate*).
<code>timeIntervalSinceDate</code>	Compares anotherDate (NSDate*) to the date and returns an NSTimeInterval.
<code>timeIntervalSinceNow</code> <code>+timeIntervalSince</code> <code>ReferenceDate</code> <code>timeIntervalSince1970</code>	Returns an NSTimeInterval representing the period between the data object and the given date. <code>timeIntervalSinceReferenceDate</code> is both a class and instance method.
<code>description</code>	Returns the date as an NSString* YYYY-MM-DD HH:MM:SS +HHMM (the last value is the difference from universal time—UT/GMT).
<code>descriptionWithLocale</code>	Same as <code>description</code> , but uses <code>localeDictionary</code> (NSDictionary*) for keys and values.

TABLE 9-10. NSDate Utility Methods (Continued)

<code>descriptionWithCalendarFormat</code>	Same as <code>description</code> , but formatted and converted using <code>formatString</code> (<code>NSString*</code>), <code>timeZone</code> (<code>NSTimeZone*</code>), and <code>localeDictionary</code> (<code>NSDictionary*</code>).
<code>dateWithCalendarFormat</code>	Returns an <code>NSDate*</code> using <code>formatString</code> (<code>NSString*</code>) and <code>timeZone</code> (<code>NSTimeZone*</code>).

NSDate `NSDate` (Table 9-11) combines date objects with format strings (based on the C `strftime` function format) and `NSTimeZone*` objects. You can change the format and the time zone for the underlying calendar date without changing the date itself.

TABLE 9-11. NSDate Methods

<code>initWithString</code>	Initializes an <code>NSDate</code> to the date specified in <code>description</code> (<code>NSString*</code>). If <code>format</code> (<code>NSString*</code>) is present it is assumed to be a calendar format; if not, the standard international format is used.
<code>dateWithString</code>	Returns an <code>NSDate</code> based on <code>description</code> (<code>NSString*</code> —the date) and <code>format</code> (<code>NSString*</code> —the <code>strftime</code> format).
<code>dateWithYear</code>	Creates and returns an <code>NSDate</code> object using the arguments <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minute</code> , and <code>second</code> (all unsigned int) and <code>timeZone</code> (<code>NSTimeZone*</code>).

TABLE 9-11. NSDate Methods (Continued)

<code>initWithYear</code>	See <code>dateWithYear</code> .
<code>dayOfMonth</code> <code>dayOfWeek</code> <code>dayOfYear</code> <code>hourOfDay</code> <code>minuteOfHour</code> <code>monthOfYear</code> <code>secondOfMinute</code> <code>yearOfCommonEra</code>	Returns the appropriate value as an int.
<code>addYear</code>	Adds the values in arguments <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minute</code> , <code>second</code> to the date (all int; any can be negative).
<code>description</code>	Returns the date as an NSString*.
<code>descriptionWithCalendarFormat</code>	Returns the date as an NSString* using <code>format</code> (NSString*). The optional <code>timeZone</code> (NSTimeZone*) argument modifies the string according to that time zone.
<code>setCalendarFormat</code> <code>calendarFormat</code>	The format based on strftime. Set: <code>format</code> (NSString*). Get: returns NSString*.
<code>setTimeZone</code>	Sets the NSDate's time-zone to <code>timeZone</code> (NSTimeZone*), replacing either the default or the time zone specified at fabrication.
<code>timeZoneDetail</code>	Returns the NSDate's NSTimeZoneDetail*.

NSTimeZone NSTimeZone objects modify NSDate objects leaving the underlying date unchanged. Since NSTimeZone objects incorporate not only geographic time zones but also daylight savings time, they are useful in helping to provide a

clean chronology unencumbered by the changes generated by timekeeping mechanisms. The `NSTimeZone` methods are shown in Table 9-12.

TABLE 9-12. `NSTimeZone` Methods

<code>+[setDefaultTimeZone]</code> <code>+[defaultTimeZone]</code>	The default system time zone (set in Preferences). Set: <code>aTimeZone</code> (<code>NSTimeZone*</code>). Get: returns <code>NSTimeZoneDetail*</code> .
<code>+[localTimeZone]</code>	
<code>+[timeZoneDetailForDate:]</code>	Returns an <code>NSTimeZoneDetail*</code> for <code>date</code> (<code>NSDate*</code>).
<code>+[timeZoneForSecondsFromGMT:]</code>	Returns and if necessary creates an <code>NSTimeZone*</code> with an offset of <code>seconds</code> (<code>int</code>) from GMT.
<code>+[timeZoneWithAbbreviation:]</code>	Returns the <code>NSTimeZoneDetail*</code> that matches <code>abbreviation</code> (<code>NSString*</code>).
<code>+[timeZoneWithName:]</code>	Returns the <code>NSTimeZone*</code> with the name <code>aTimeZoneName</code> (<code>NSString*</code>).
<code>+[timeZoneArray]</code>	Returns an <code>NSArray*</code> of all current region names for this time zone.
<code>+[timeZoneDetailArray]</code>	Returns an <code>NSArray*</code> with the <code>NSTimeZoneDetail</code> objects for this <code>NSTimeZone</code> .
<code>+[abbreviationDictionary]</code>	Returns an <code>NSDictionary*</code> with the name/abbreviation mappings.
<code>+[timeZoneName]</code>	Returns the <code>NSString*</code> name of a time zone.

NSTimeZoneDetail Within a time zone, `NSTimeZoneDetail` objects specify whether daylight savings time is in effect. Table 9-13 shows the `NSTimeZoneDetail` methods.

TABLE 9-13. NSTimeZoneDetail Methods

<code>isDaylightSavingsTimeZone</code>	Returns a <code>BOOL</code> .
<code>timeZoneAbbreviation</code>	Returns an <code>NSString*</code> .
<code>timeZoneSecondsFromGMT</code>	Returns an <code>int</code> .

Error Handling

Decades ago it became apparent that programs did not always start executing with their first statement and continue on without interruption or incident to the end. Any number of problems could occur, and the history of computers is replete with horror stories in which tremendous damage occurred as programs went awry and failed in unpredicted ways.

You can take nothing for granted in the computing environment. Even if you are absolutely certain that you know how your application will be used, you cannot predict how people will modify your application and use it for purposes you never expected. Particularly in the dynamic environment of Rhapsody, where users, consultants, and other developers are encouraged to combine and recombine software products, you must make certain that your code is as fail-safe as possible.

Error handling in OpenStep is provided by exceptions; assertions use exceptions to help you track logic problems in your code.

Exceptions

Contemporary software design relies on exceptions as a way to keep software running. In the old days, it was considered good programming practice for every method or procedure to return a result code indicating if it had succeeded or failed; sophisticated software returned a result code which specified the type of failure. (See the `OSErr` type in the Mac OS toolbox for many examples of this strategy.)

There are two problems with this strategy. First, it requires the programming to check the result code, and that means extra code to be written. Second, it means that the calling routine must know the possible result codes that could be returned. While this may seem to be a trivial concern, it is not. As you may have experienced for yourself, it means that if you invoke a string manipulation function you suddenly need to know what string result codes could be returned—and all you wanted to do was convert a date to a string!

The exception structure has the following pattern:

1. A critical block in which failure could occur is identified. You mark the beginning and end of that block (in OpenStep with the `NS_DURING` and `NS_HANDLER` macros).
2. If an error occurs within that block, either in its own code or in code that it calls, an exception is raised. In OpenStep, an `NSException` object is created which incorporates not only an error code but also a plain language description of the error. The `raise` message is sent to this object, and execution in the critical block is stopped.
3. Following the end of the critical block (the `NS_HANDLER` macro) is a section of code to which control is transferred if an exception is raised. This code—which is terminated by the `NS_ENDHANDLER` macro—can clean up from the problem. It may notify the user, but often it can manage the problem it-

self. In other cases, it simply propagates the failure further up the line where another section of exception handling code will manage it. OpenStep provides exception handling default code for any exceptions that reach it (i.e., exceptions that have not been fully processed by their own handlers).

As a result, you never have to check whether a method has succeeded or not—there are usually no result codes. If a method fails, it will raise an exception and you are responsible for catching it. If you don't even do that, OpenStep itself will catch the exception.

Furthermore, since the exception object itself contains a description of the exception, you can process an exception—either passing it on or displaying information for the user—without knowing anything about it.

For this structure to work, you must do two things:

1. If you perform an operation that could fail, you should test for the possible failure and, if it could occur, create and raise an exception.
2. You should bracket code with `NS_DURING`, `NS_HANDLER`, and `NS_ENDHANDLER` to catch and clean up from exceptions that are raised.

NSException The methods of the `NSException` class are shown in Table 9-14. An exception's name is a unique identifier of the exception; reason is a more detailed string; userInfo is an `NSDictionary` object that may have more application-specific information.

TABLE 9-14. NSException Methods

<code>+exceptionWithName</code>	Creates the exception object with <code>name</code> (<code>NSString*</code>), <code>reason</code> (<code>NSString*</code>), and <code>userInfo</code> (<code>NSDictionary*</code>).
<code>+raise</code>	Raises an exception you have created (probably with <code>exceptionWithName</code>).
<code>+raise</code>	Creates and raises an exception with <code>name</code> (<code>NSString*</code>), and a reason constructed from <code>format</code> (<code>NSString*</code>) and a list of arguments used as in <code>printf()</code> or <code>vprintf()</code> .
<code>initWithName</code>	Initializes an exception with <code>name</code> (<code>NSString*</code>), <code>reason</code> (<code>NSString*</code>), and <code>userInfo</code> (<code>NSDictionary*</code>).
<code>exceptionName</code> <code>exceptionReason</code>	Returns an <code>NSString*</code> .
<code>exceptionUserInfo</code>	Returns an <code>NSDictionary*</code> .

Assertions

Assertions allow you to check that a particular condition that you assume is true is indeed true. Most often used in debugging, they provide a simple way to make certain that you haven't accidentally wound up with `nil` when you thought you had an object or that you have an object of the class you expect.

Assertions are usually dealt with by using the macros `NSAssert()` and `NSCAssert()`. The first handles assertions involving objects; the second handles assertions in C code.

If an assertion fails, an `NSInternalInconsistencyException` is thrown.

NSAssertionHandler Usually you don't need to worry about NSAssertionHandlers—you just use the macros and wait for NSInternalInconsistencyExceptions to be thrown. Table 9-15 shows the NSAssertionHandler methods.

TABLE 9-15. NSAssertionHandler Methods

<code>currentHandler</code>	Returns the NSAssertionHandler* associated with this thread.
<code>handleFailureInFunction</code>	Creates an assertion exception for the given <code>functionName</code> (NSString*), <code>filename</code> (NSString*), <code>line</code> (int), and description as described in <code>formattedString</code> (NSString*).
<code>handleFailureInMethod</code>	Creates an assertion exception for the given <code>anObject</code> , <code>fileName</code> (NSString*), <code>line</code> (int), and description as described in <code>formattedString</code> (NSString*).

Programming Tools

The objects in this section provide support for the mundane programming chores involving arrays, characters, and strings. In addition, OpenStep provides the NSDictionary class, which is similar to the Smalltalk dictionary object.

Arrays and their enumerators provide support for collections of objects that may or may not be ordered. If they are ordered, their ordering is based on their intrinsic data: that is, an array that contains objects representing people could be ordered by their heights.

Dictionaries and their associated enumerators provide support for collections of objects that are identified by keys that are extrinsic to the object. For example, in the case just cited, a person's height is an integral part of that person; a person's social security number is extrinsic—not a part of the person.

Arrays and Enumerators

One of the most common programming tasks is to iterate over a list of items performing a common test or function on each one. OpenStep's support for array objects provides this capability. Elements of array objects must be non-nil objects, but since all OpenStep objects descend from NSObject, that means that any OpenStep object can be used in an array object.

Array objects retain their elements as they add them and release them as they remove them. As a result, the automatic garbage collection routines in OpenStep sidestep objects that are currently in an array object.

Simple data structures can be wrapped in NSData, NSValue, or NSNumber objects.

NSArray The NSArray fabrication methods are shown in Table 9-16.

TABLE 9-16. NSArray Fabrication Methods

<code>mallocWithZone</code>	
<code>tarray</code>	Creates and returns an array object.
<code>tarrayWithObject</code>	Creates and returns an array object. The array object is populated with anObject.

TABLE 9-16. NSArray Fabrication Methods (Continued)

<code>initWithObjects</code>	Creates and returns an array object populated with the objects in the argument list of the method.
<code>initWithArray</code>	Initializes an array object with the elements from <code>array</code> (<code>NSArray*</code>).
<code>initWithObjects</code>	Initializes an array object and populates it with the objects in this method's argument list.
<code>initWithObjects</code>	Initializes an array object and populates from another array of objects (<code>id*</code>); <code>count</code> (unsigned) objects are moved to the new array.

Table 9-17 shows the NSArray utility methods. A distinction is made between elements of the array that are equal and those that are identical. Equal elements are those that a compare routine says are equal (for example, they may have the same serial number). Identical elements are elements that are the same. Thus, if you have non-unique serial numbers, you could have two distinct objects that are equal in the array (having the same serial number), while only one identical object exists.

TABLE 9-17. NSArray Utility Methods

<code>containsObject</code>	Returns a <code>BOOL</code> if <code>anObject</code> is in the array.
<code>count</code>	Returns an unsigned.

TABLE 9-17. NSArray Utility Methods (Continued)

<code>indexOfObject</code> <code>indexOfObjectIdenticalTo</code>	Returns an unsigned value of the element that is equal (first method) or identical (second method) to <code>anObject</code> . The search is from the back of the array forward.
<code>lastObject</code>	Returns the last object.
<code>objectAtIndex</code>	Returns the object at <code>index</code> (unsigned).
<code>indexOfObject</code> <code>indexOfObjectIdenticalTo</code>	Returns an unsigned representing the index of an object equal (first) or identical (second) to <code>anObject</code> (running from back to front).
<code>objectEnumerator</code> <code>reverseObjectEnumerator</code>	Returns an <code>NSEnumerator*</code> for the array for either forward or reverse enumeration.
<code>makeObjectsPerform</code>	Iterates from back to front of the array, invoking <code>aSelector(SEL)</code> on each element; the optional argument <code>anObject</code> is sent as an argument of the message if it is present.
<code>firstObjectCommonWithArray</code>	Returns the first object from <code>otherArray</code> (<code>NSArray*</code>) that is in this one.
<code>isEqualToArray</code>	Returns <code>BOOL</code> if the elements (and order) of <code>otherArray</code> (<code>NSArray*</code>) are equal (not identical) to this one.

TABLE 9-17. NSArray Utility Methods (Continued)

<code>sortedArrayUsingFunction</code>	<p>Returns a new array object with this array's elements sorted in it. The arguments are a <code>comparator</code> function and a <code>context</code> (<code>void*</code>).</p> <p>The <code>comparator</code> function's header is <code>int*(id, id, void*)</code>; it compares two objects with an optional <code>context</code>.</p>
<code>sortedArrayUsingSelector</code>	Same as above, but the sole argument is <code>comparator</code> (SEL). You can use <code>compare</code> methods from various classes as the argument rather than reinventing the wheel.
<code>subArrayWithRange</code>	Returns an array object with the elements from this array within <code>range</code> (NSRange).
<code>componentsJoinedByString</code>	Returns an <code>NSString*</code> formed by concatenating the array's elements (which must be <code>NSString</code> s) with one another, interposing the <code>separator</code> (<code>NSString*</code>) argument.
<code>description</code> <code>descriptionWithIndent</code>	Returns an <code>NSString*</code> describing the array in the <code>PropertyList</code> format. The second method takes a <code>level</code> (unsigned) argument that lets you indent the output (<code>=1</code>) or not (<code>=0</code>).

NSMutableArray Mutable arrays are modifiable—they can have objects inserted and deleted from them, as shown in the methods of Table 9-18.

TABLE 9-18. NSMutableArray Methods

<code>mallocWithZone</code>	
<code>initWithCapacity</code> <code>initWithCapacity</code>	Creates or initializes an array for <code>numItems</code> (unsigned).
<code>addObject</code>	Adds <code>anObject</code> at the end of the array.
<code>addObjectsFromArray</code>	Adds all the objects from <code>otherArray</code> (NSArray*) at the end of this one.
<code>insertObject</code>	Adds <code>anObject</code> to this array at <code>index</code> (unsigned).
<code>removeAllObjects</code> <code>removeLastObject</code>	
<code>removeObject</code> <code>removeObjectIdenticalTo</code>	Removes <code>anObject</code> from the array. The first method uses equality; the second uses identity.
<code>removeObjectAtIndex</code>	Removes the object at <code>index</code> (unsigned).
<code>removeObjectsFromIndices</code>	Removes several objects as indices (unsigned*); the number of them is given in <code>count</code> (unsigned).
<code>removeObjectsInArray</code>	Removes all the objects from <code>otherArray</code> (NSArray*) from this one.
<code>replaceObjectAtIndex</code>	Replaces the object at <code>index</code> (unsigned) with <code>anObject</code> .

TABLE 9-18. NSMutableArray Methods (Continued)

<code>sortUsingFunction</code>	<p>Sorts this array object. The arguments are a <code>compare</code> function and a <code>context</code> (<code>void*</code>).</p> <p>The comparator function's header is <code>int*(id, id, void*)</code>; it compares two objects with an optional context.</p> <p><code>NSArray</code>'s <code>sortedArrayUsingFunction</code> creates a new array; this sorts the existing array in place.</p>
--------------------------------	--

NSEnumerator Enumerators are created by array objects; you use them to traverse the array. An `NSEnumerator` can give you only the next object: whether it runs forward or backward is determined when you create it. You can't change direction in the middle of iteration. `NSEnumerator` is shown in Table 9-19.

TABLE 9-19. NSEnumerator Method

<code>nextObject</code>	Returns the next object or <code>nil</code> at the end.
-------------------------	---

Dictionaries

Dictionaries contain pairs of objects (associations). A key (`NSString`) is used to identify each data element in the dictionary: the association of the key and the data element is a single dictionary entry. The dictionary class uses optimized and sophisticated searching routines to manage large numbers of keys.

Although the word “dictionary” conjures up lexicological associations, dictionaries can be used productively to store and

organize many types of data. In developing applications for Rhapsody (or in modifying legacy applications), cast a critical eye on your files and data structures that use numeric codes to store and retrieve information. Traditional programming techniques tend to look askance at the use of strings for data keys, and it is true that in the old days string manipulation was expensive enough on computers that non-numeric data keys caused significant performance degradation. Those days have passed, and the benefits of simple code outweigh any possible performance problems in using keys of strings.

NSDictionary The basic dictionary class is NSDictionary. Table 9-20 shows its fabrication methods.

TABLE 9-20. NSDictionary Fabrication Methods

<code>mallocWithZone</code>	
<code>tdictionary</code>	Creates and returns a new dictionary.
<code>tdictionaryWithObjects</code>	Creates and returns a new dictionary filled with entries from the arrays of its two arguments: <code>objects</code> (NSArray*) and <code>keys</code> (NSArray*). The optional third argument <code>count</code> (unsigned) limits the number of elements to be entered.
<code>initWithContentsOfFile</code>	Initializes a dictionary to the contents of a file previously written out using <code>writeToFile</code> . The sole argument is <code>path</code> (NSString*).
<code>initWithDictionary</code>	Copies the contents of <code>otherDictionary</code> (NSDictionary*).

TABLE 9-20. NSDictionary Fabrication Methods (Continued)

<code>initWithObjects</code>	Loads a dictionary with <code>objects (id*)</code> , <code>keys (NSString**)</code> ; <code>count (unsigned)</code> is the number of objects to be loaded.
------------------------------	--

Table 9-21 provides the NSDictionary utility methods that you use to query a dictionary.

TABLE 9-21. NSDictionary Utility Methods

<code>count</code>	Returns an unsigned of the number of entries.
<code>allKeys</code>	Returns an <code>NSArray*</code> with all the key objects; use this method to provide an index to the dictionary.
<code>allKeysForObject</code>	Returns an <code>NSArray*</code> with all the keys associated with <code>anObject</code> . Note that a single object can appear many times in the dictionary with several different keys.
<code>allValues</code>	Returns an <code>NSArray*</code> with all the values (as opposed to keys) in the dictionary.
<code>description</code> <code>descriptionInStrings</code> <code>FileFormat</code> <code>descriptionWithIndent</code>	Returns an <code>NSString*</code> of the dictionary's contents, optionally in strings file format. <code>description-WithIndent</code> takes the argument level (unsigned): if =1, elements are indented from the left margin.
<code>keyEnumerator</code> <code>objectEnumerator</code>	Returns an <code>NSEnumerator*</code> for all the dictionary's keys or data values.

TABLE 9-21. NSDictionary Utility Methods (Continued)

<code>objectForKey</code>	Returns the value for <code>aKey</code> (NSString*).
<code>isEqualToDictionary</code>	Returns a BOOL if this dictionary's contents are equal to the contents of <code>otherDictionary</code> (NSDictionary*).
<code>writeToFile</code>	Writes the dictionary's contents to a file specified in <code>path</code> (NSString*); <code>atomically</code> (BOOL) controls how the file is written. The <code>initWithContentsOfFile</code> method reads files created this way.

NSMutableDictionary You use an NSMutableDictionary (Table 9-22) when you want to add and delete entries from a dictionary. Note that you often create a mutable dictionary for users to create and modify and then recreate it (possibly by writing it to disk and rereading it in the same or other application) as a nonmutable dictionary.

TABLE 9-22. NSMutableDictionary Methods

<code>mallocWithZone</code>	
<code>dictionaryWithCapacity</code> <code>initWithCapacity</code>	Creates or initializes a dictionary with space for <code>numItems</code> (unsigned).
<code>addEntriesFromDictionary</code>	Adds the elements of <code>otherDictionary</code> (NSDictionary*) to this one

TABLE 9-22. NSMutableDictionary Methods (Continued)

<code>removeAllObjects</code>	Removes objects—either all, the object for <code>aKey</code> (<code>NSString*</code>), or all of the objects in <code>keyArray</code> (<code>NSArray*</code>).
<code>removeObjectForKey</code>	
<code>removeObjectForKeys</code>	
 <code>setObject</code>	 Adds <code>anObject</code> to the dictionary with <code>aKey</code> (<code>NSString*</code>).

Text

OpenStep provides sophisticated support for text processing. In this section you will find the basic classes for characters and strings, the scanners that manipulate text, and the spelling classes.

Characters and Strings

Text consists of strings of characters. Today, those characters can be one- or two-byte characters in Roman, Cyrillic, Chinese, and many other alphabets.

NSCharacterSet Character sets let you manipulate characters not only in standard sets but in other sets that you can define. Thus, a character set can consist not only of Roman letters but also of the characters 1, a, G, %, and Z, which might have specific meaning in a given game or other application.

Table 9-23 shows the methods of `NSCharacterSet`.

NSMutableCharacterSet Once you have created a character set, it is not modifiable. If you want to dynamically add or remove individual characters from a character set, you use an `NSMutableCharacterSet` as shown in Table 9-24.

TABLE 9-23. NSString Methods

<code>+alphanumericCharacterSet</code>	Creates and returns standard character sets as <code>NSString*</code> objects.
<code>+controlCharacterSet</code>	
<code>+decimalDigitCharacterSet</code>	
<code>+decomposableCharacterSet</code>	
<code>+illegalCharacterSet</code>	
<code>+letterCharacterSet</code>	
<code>+lowercaseLetterCharacterSet</code>	
<code>+nonBaseCharacterSet</code>	
<code>+uppercaseLetterCharacterSet</code>	
<code>+whitespaceCharacterSet</code>	
<code>+whitespaceAndNewlineCharacterSet</code>	
<code>+characterSetWithRange</code>	Creates and returns an <code>NSString*</code> with the Unicode characters in <code>aRange</code> (<code>NSRange</code>). Since runs of characters have meaning (all lowercase letters are together, all numbers are together), you often can use this method to quickly create a needed character set. Before doing so, though, check the standard character sets in the previous section.
<code>+characterSetWithCharactersInString</code>	Creates and returns an <code>NSString*</code> consisting of the characters in <code>aString</code> (<code>NSString*</code>).
<code>+characterSetWithBitmapRepresentation</code>	Creates and returns an <code>NSString*</code> from data (<code>NSData*</code>).
<code>bitmapRepresentation</code>	Create an <code>NSData*</code> representation of the character set.
<code>characterIsMember</code>	Returns a <code>BOOL</code> indicating if <code>aCharacter</code> (<code>unichar</code>) is a member of the set.
<code>invertedSet</code>	Returns an <code>NSString*</code> of all characters not in this set.

TABLE 9-24. NSMutableCharacterSet Methods

<code>addCharactersInRange</code> <code>removeCharactersInRange</code>	Adds or removes the Unicode characters in <code>aRange</code> (<code>NSRange</code>) to or from the set.
<code>addCharactersInString</code> <code>removeCharactersInString</code>	Adds or removes the Unicode characters in <code>aString</code> (<code>NSString*</code>) to or from the set.
<code>formIntersection</code> <code>WithCharacterSet</code> <code>formUnion</code> <code>WithCharacterSet</code>	Modifies the character set as noted based on <code>otherSet</code> (<code>NSCharacterSet*</code>).
<code>invert</code>	Replaces the elements of the character set with all the elements it did not contain. Note that this is much less efficient than inverting an <code>NSCharacterSet</code> ; also, inverting an <code>NSCharacterSet</code> creates a new set—this modifies the existing one.

NSString Strings are constantly used, and the `NSString` class provides methods for managing them. Before you jump head first into `NSString` and use it for manipulating strings, look at the following section and see if `NSScanner`—a higher level and more sophisticated class—can do the work for you.

The fabrication methods of `NSString` are shown in Table 9-25.

TABLE 9-25. NSString Methods

<code>+stringWithCharacters</code>	Creates and returns an <code>NSString*</code> from <code>chars</code> (<code>const unichar*</code>) of <code>length</code> (unsigned).
------------------------------------	--

TABLE 9-25. NSString Methods (Continued)

<code>initWithCString</code>	Creates and returns an <code>NSString*</code> from <code>byteString</code> (<code>const char*</code>); if <code>length</code> (unsigned) is provided that is used—otherwise it is assumed to be a null-terminated string.
<code>initWithFormat</code>	Creates and returns an <code>NSString*</code> using <code>format</code> (<code>NSString*</code>) and the variable arguments that follow as in the <code>printf()</code> function.
<code>initWithCharacters</code> <code>initWithCString</code> <code>initWithFormat</code>	Comparable to above methods.
<code>initWithCharactersNoCopy</code> <code>initWithCStringNoCopy</code>	Same as methods above, but take an extra argument <code>flag</code> (<code>BOOL</code>) that determines whether or not memory will be released when no longer needed.
<code>initWithContentsOfFile</code>	Initializes an <code>NSString*</code> from a file at <code>path</code> (<code>NSString*</code>).
<code>initWithString</code>	Initializes the string with the characters from <code>aString</code> (<code>NSString*</code>).
<code>init</code>	The default method initializes a string to no characters.

Encodings You use `NSStringEncoding` objects to convert strings from one character set to another. `NSStringEncoding` objects have different characteristics with regard to trade-offs of speed, space, etc.; an `NSString` object can return different types of encodings as you can see from Table 9-26.

TABLE 9-26. NSString Encoding Methods

<code>dataUsingEncoding</code>	Creates an <code>NSData*</code> object using <code>encoding</code> (<code>NSStringEncoding</code>). The optional second argument <code>flag</code> (<code>BOOL</code>) specifies whether to allow conversion that may lose information. This is the complement of <code>initWithData</code> .
<code>initWithData</code>	Initializes the string from data (<code>NSData*</code>) using <code>encoding</code> (<code>NSStringEncoding</code>).
<code>+defaultCStringEncoding</code> <code>+fastestEncoding</code> <code>+smallestEncoding</code>	Returns various <code>NSStringEncoding</code> s.
<code>canBeConvertedToEncoding</code>	Returns a <code>BOOL</code> indicating if the string can be converted to <code>encoding</code> (<code>NSStringEncoding</code>) without losing information.

Table 9-27 presents the string manipulation methods.

TABLE 9-27. NSString String Manipulation Methods

<code>characterAtIndex</code>	Returns the <code>unichar</code> at <code>anIndex</code> (unsigned).
<code>length</code>	Returns the string's length as an unsigned <code>int</code> .
<code>capitalizedString</code> <code>uppercaseString</code> <code>lowercaseString</code>	Returns an <code>NSString*</code> that is capitalized (initial caps—not all caps), uppercased (all caps), or lowercased.

TABLE 9-27. NSString String Manipulation Methods

<code>commonPrefix</code>	Returns an NSString* of the first characters in common with <code>aString</code> (NSString*); <code>mask</code> (unsigned) can be the constants <code>NSCaseInsensitiveSearch</code> , <code>NSLiteralSearch</code> either separately or combined with <code>1</code> .
<code>hasPrefix</code>	Returns a BOOL if <code>aString</code> (NSString*) matches any beginning characters of this string.
<code>rangeOfCharacterFromSet</code>	<p>Returns an NSRange of the first characters found in the string that is in <code>aSet</code> (NSCharacterSet*). The optional second argument <code>mask</code> (unsigned int) can have the value <code>NSLiteralSearch</code>, <code>NSBackwardsSearch</code>, or <code>NSAnchoredSearch</code>.</p> <p>An alternative version allows you to specify the same options in <code>mask</code> as for <code>commonPrefix</code> and then to specify a third argument <code>aRange</code> (NSRange) to limit the search.</p>
<code>rangeOfString</code>	This method is the same as <code>rangeOfCharactersFromString</code> except it works with <code>aString</code> (NSString*).
<code>rangeOfComposed CharacterSequenceAtIndex</code>	

TABLE 9-27. NSString String Manipulation Methods

compare	Returns an <code>NSComparisonResult</code> based on this string and <code>aString</code> (<code>NSString*</code>). The same constants as <code>commonPrefix</code> uses may be applied to the optional argument <code>mask</code> (unsigned). A final optional argument, <code>range</code> (<code>NSRange</code>), lets you specify the part of the strings to be compared.
isEqual	Returns a <code>BOOL</code> if they are the same object or if their contents are equal. The argument is an <code>Object</code> .
isEqualToString	Returns a <code>BOOL</code> if they are the same object or if their contents are equal. The argument is a <code>String</code> (<code>NSString*</code>).
propertyList propertyListFrom StringsFileFormat	
substringToIndex substringFromIndex substringFromRange	Returns an <code>NSString</code> of the characters from the start to an <code>Index</code> (unsigned) or from an <code>Index</code> to the end of the string; alternatively, returns an <code>NSString</code> of the characters within a <code>Range</code> (<code>NSRange</code>).
componentsSeparated ByString	Returns an <code>NSArray*</code> of the string's elements where elements are assumed to be separated by the sole argument <code>separator</code> (<code>NSString*</code>). <code>NSArray</code> implements <code>componentsJoinedByString</code> , which is the complement of this.
cString cStringLength	Returns the string as a <code>const char*</code> and its length as unsigned.

TABLE 9-27. NSString String Manipulation Methods

<code>getCharacters</code>	Returns the characters into <code>buffer</code> (<code>unichar*</code>); if specified, <code>aRange</code> (<code>NSRange</code>) is used.
<code>getCString</code>	Returns the characters as a C string into <code>buffer</code> (<code>char*</code>); you can specify <code>maxLength</code> (unsigned) for <code>buffer</code> as well as <code>aRange</code> (<code>NSRange</code>). If you specify <code>aRange</code> , you must pass in a final argument <code>leftoverRange</code> (<code>NSRange*</code>) in which the unused character range will be returned.
<code>floatValue</code> <code>intValue</code>	Returns the contents of the string as a float or int.
<code>stringByAppendingFormat</code> <code>stringByAppendingString</code>	Adds to this string either by applying the <code>printf()</code> rules to <code>aFormat</code> (<code>NSString*</code>) and the variable arguments that follow or by adding the contents of <code>aString</code> (<code>NSString*</code>).
<code>description</code>	Returns an <code>NSString*</code> which is the string surrounded by quotes.

A lot of string manipulation is used to manage file path names as shown in Table 9-28.

NSMutableString If you want to modify a string after you create it, you create an `NSMutableString`; a descendant of `NSString`, it has all the `NSString` methods in addition to its own methods shown in Table 9-29.

TABLE 9-28. NSString File Path Name Methods

```

completePathIntoString
lastPathComponent
pathExtension
stringByAbbreviating
    WithTildeInPath
stringByAppending
    PathComponent
stringByAppendingPath
    Extension
stringByDeletingLast
    PathComponent
stringByDeletingPath
    Extension
stringByExpandingTilde
    InPath
stringByResolving
    SymlinksInPath
stringByStandardizingPath

```

TABLE 9-29. NSMutableString Methods

<code>initWithCapacity</code>	Creates or initializes a string for capacity (unsigned) characters.
<code>initWithCharacters</code> <code>initWithCString</code> <code>initWithFormat</code>	Like NSString methods, but these create and return NSMutableString objects.
<code>appendFormat</code> <code>appendString</code>	Using <code>format</code> (NSString*) and the variable arguments that follow as for <code>printf()</code> or using <code>aString</code> (NSString*), add to this string.
<code>deleteCharactersInRange</code>	Deletes the characters in <code>aRange</code> (NSRange) from the string.
<code>replaceCharactersInRange</code>	Replaces the characters in <code>aRange</code> (NSRange) with those in <code>otherString</code> (NSString*).

TABLE 9-29. NSMutableString Methods (Continued)

<code>setString</code>	Sets this string to the characters in <code>aString</code> (NSString*).
<code>insertString</code>	Inserts <code>aString</code> (NSString*) at <code>anIndex</code> (unsigned).

Scanning Text

The NSScanner class combines string manipulation utilities that you may have used in other languages with code that can interpret numbers. The methods are shown in Table 9-30.

The scanning methods at the end of Table 9-30 all have a similar structure. They scan for a substring or set of characters; characters are accumulated into a second argument, value (NSString**). This argument may be nil, in which case the ef-

TABLE 9-30. NSScanner Utility Methods

<code>ts scannerWithString</code> <code>initWithString</code> <code>string</code>	Creates or initializes an NSScanner with <code>aString</code> (NSString*). <code>string</code> is the accessor; it returns the NSString*.
<code>setScanLocation</code> <code>scanLocation</code>	You can position the scanner to an absolute position for the next scan to start at. Set: index (unsigned). Get: returns an unsigned.
<code>setCaseSensitive</code> <code>caseSensitive</code>	Accessor for case sensitivity (default is not case sensitive). Set: <code>flag</code> (BOOL). Get: returns a BOOL.

TABLE 9-30. NSScanner Utility Methods (Continued)

<code>setCharactersToBeSkipped</code> <code>charactersToBeSkipped</code>	<p>Accessor for the <code>NSCharacterSet</code> of characters to ignore in scanning. Default is a set of the blank and newline characters.</p> <p>Set: <code>aSet</code> (<code>NSCharacterSet*</code>). Get: returns an <code>NSCharacterSet*</code>.</p>
<code>scanDouble</code> <code>scanInt</code> <code>scanFloat</code> <code>scanLongLong</code>	<p>All methods return a <code>BOOL</code> if successful at scanning into <code>value</code> (<code>double*</code>, <code>int*</code>, <code>float*</code>, or <code>longlong*</code>).</p>
<code>isAtEnd</code>	<p>Returns a <code>BOOL</code> if there are no characters left to scan (characters to be skipped are ignored, so you may not be at the end of the string).</p>
<code>scanString</code> <code>scanUpToString</code>	<p>Scans while characters match (or until they match) <code>aString</code> (<code>NSString*</code>). If found, sets the second argument <code>value</code> (<code>NSString**</code>) to a reference to the string object of those characters. Returns <code>BOOL</code> if successful.</p> <p><code>value</code> can be <code>nil</code>.</p>
<code>scanCharactersFromSet</code> <code>scanUpToCharactersFromSet</code>	<p>Scans while the string characters match (or until they match) <code>aSet</code> (<code>NSCharacterSet*</code>), accumulating the characters in <code>value</code> (<code>NSString**</code>).</p> <p><code>value</code> can be <code>nil</code>.</p>

fect of the `scan` method is simply to reposition the scan location to the beginning or end of a string or set of characters. You can use this to scan past filler words or characters. All of

these routines start scanning at the current scan location, which may not be the beginning of the string.

OpenStep Applications

The heart of an OpenStep application is `NSApplication`. OpenStep applications come in bundles—groups of files that contain resources, code, and other information. Utilities for managing bundles are also described in this section.

Finally, the nib awaking protocol is mentioned—it is invoked when your application awakes either for the first time or after having been temporarily suspended.

`NSApplication`

The `NSApplication` object is the core of an OpenStep application. You create it in your `main()` function, send it a run message, and it then takes care of handling events and user interactions until it is done. It terminates itself, control returns to your `main()` function, and that's the end of your program.

Unlike the application objects in some other frameworks (such as `MacApp`), you rarely subclass `NSApplication`. There are only three reasons to subclass an application object in most frameworks:

1. You subclass an application object if you want to change default application-wide behavior. This is almost always a bad idea (do you really want experienced users to have to relearn how to resize windows just for your application?).
2. You want to provide added functionality at the application level (as opposed to adding functionality to windows). Even in this case you could create a category and add it to the application object rather than overriding it.

3. Since the application object is in charge of the event loop and user interaction, it is the logical place to make modifications if you want your application to intercept commands, log them, or notify other processes of its doings.

Here is why you rarely subclass `NSApplication`:

1. Changing application-wide behavior is a bad idea in any framework, so you wouldn't do it in OpenStep either.
2. As discussed in "Responder Chain" on page 129, OpenStep implements the concept of delegate objects. Messages are sent to `NSApplication` and it passes many of them on to a delegate object. A delegate object is a descendant of `NSObject`, and you can use it to encapsulate your application-specific functionality. You attach a delegate to an `NSApplication` object using the `setDelegate` method. Since OpenStep can quite happily handle messages that it doesn't recognize, `NSApplication` has no problem receiving a message for your application-specific functionality and passing it on to your custom delegate. You can also add functionality at the application level by modifying the menus in your application bundle (which doesn't affect the application itself).
3. The `NSApplication` object posts a number of default notifications to its notification center (see "NSNotificationCenter" on page 333). Since there is a single default notification center for each task, you can post additional notifications to it from any of your objects without modifying `NSApplication`.

The remainder of this section details the `NSApplication` methods. Since you rarely override `NSApplication`, you rarely override them. You call very few of them, but the complete list should give you a sense of what the framework is doing for

you—and the demands that you can place on it. On initialization, the global variable `NSApp` points to your `NSApplication` instance; once you have `NSApp` (`gApplication` in `MacApp`), you can invoke any of these methods. If you need to access your application's specific data structures, you can send messages to `NSApp` that will be forwarded to your delegate object, where you can do any processing you need. (Your custom delegate object is an important part of your system design. If you're tempted to override `NSApplication`, question whether you can handle the situation more effectively with a delegate. You probably can.)

The functionality of `NSApplication` is divided into:

- application management
- event management
- window management
- menu management

Application Management The basic application management routines are shown in Table 9-31.

TABLE 9-31. `NSApplication` Management

`†sharedApplication`

This returns the `NSApplication*` object; if it hasn't been created, it creates it and does all necessary initializations and establishes connections to the operating system. You call this from `main()`.

TABLE 9-31. NSApplication Management (Continued)

<code>run</code>	<p>As soon as your <code>main()</code> function has an instance of your application (by calling <code>sharedApplication</code>), you send a <code>run</code> message to it to start the main event loop.</p> <p>You can restart the event loop after it has been stopped by using this method.</p>
<code>stop</code> <code>terminate</code>	<p>These methods take a single sender argument. The first stops the current event loop (modal events or main events). Control is transferred either from the modal event loop to the main event loop or from the main event loop back to your <code>main()</code> function. You can restart the main event loop in this case with <code>run</code>.</p> <p><code>Terminate</code> simply stops the main loop and exits to <code>main()</code>.</p>
<code>isRunning</code>	Returns a <code>BOOL</code> that is <code>YES</code> unless the main event loop was stopped.
<code>setApplicationIconImage</code> <code>applicationIconImage</code>	<p>Accessors for the application's icon.</p> <p>Set: <code>anImage (NSImage*)</code>. Get: returns an <code>NSImage*</code>.</p>
<code>showHelp</code> <code>activateContextHelpMode</code>	Both methods take a single argument sender.
<code>context</code>	Returns the Display PostScript context for the application as <code>NSDPSCContext*</code> .

TABLE 9-31. NSApplication Management (Continued)

<code>setDelegate</code> <code>delegate</code>	Accessors for the delegate. You almost always create a delegate object to handle application-specific messages, functionalities, and data structures. Delegates may be any descendant of NSObject. Set: <code>anObject</code> . Get: returns an id.
<code>reportException</code>	This reports (but does not raise) an exception for <code>anException</code> (NSException*). Use it for logging as opposed to error handling.

Event Management Events and actions are detailed in Table 9-32. As in most modern systems, there is a queue into which events (mouse clicks, etc.) are placed by the system; the application dequeues them in sequence in most cases. Routines are provided to interrogate the event queue, add and remove events from it, and quickly locate events of a certain type so that they can be processed out of sequence. OpenStep normally handles all events for you; rarely do you need to interfere.

Window Management The NSApplication object interacts with the Window Server to display windows and to process events from them. Window management falls into three general areas

- general window management
- managing modal windows
- managing panels

In Table 9-33 you will find the basic window management routines.

TABLE 9-32. NSApplication Events and Actions

<code>sendEvent</code> <code>postEvent</code>	Both methods take an argument <code>anEvent</code> (<code>NSEvent*</code>). <code>postEvent</code> takes a second argument <code>flag</code> (<code>BOOL</code>) which is true if the event is to be put at the front of the event queue. Since events are normally sent to the application object by the system, you rarely use these methods, although if you construct your own event, you would need to post it yourself.
<code>currentEvent</code>	Returns the most recent <code>NSEvent*</code> retrieved from the event queue.
<code>nextEventMatchingMask</code>	Comparably to the Mac OS <code>GetNextEvent</code> , this method retrieves the first event in the queue that matches <code>mask</code> (unsigned int), has a timestamp before <code>expiration</code> (<code>NSDate*</code>), and <code>mode</code> (<code>NSString*</code>). If the final argument <code>deqFlag</code> (<code>BOOL</code>) is true, the event is removed from the queue. (For mask values, see the OpenStep documentation.) The return result is an <code>NSEvent*</code> . Use this method if you want to pull specific types of events out of the queue. Traditionally, this has been done for efficiency in mouse tracking and other time-critical routines. Fortunately, hardware designers have made much of this unnecessary with faster (and multiple) processors.

TABLE 9-32. NSApplication Events and Actions (Continued)

<code>discardEventsMatchingMask</code>	Removes all events matching mask (unsigned int) with timestamps before <code>lastEvent</code> 's (NSEvent*).
<code>sendAction</code>	<p>Sends anAction (SEL) to aTarget; the third argument is sender. If aTarget is nil, the action is sent to the head of the responder chain and passed up through the key and main windows' responders to their delegates and thence (if still unprocessed) to the application and its delegate.</p> <p>The result is a BOOL that indicates if the action has been responded to. (Response is used in the sense that the object in question can respond to the message—by having an appropriate method; whether the object actually does respond is not an issue.)</p>
<code>tryToPerform</code>	Whereas sendAction lets you send an action to a specific target (or to the whole responder chain if the target is nil), tryToPerform sends an action to the application object. If it can't perform it, the delegate is given a chance. The result is a BOOL indicating whether or not the attempt was successful. The arguments are aSelector (SEL) and anObject which is passed into the recipient's perform method. (See NSObject perform.)
<code>targetForAction</code>	The result is the object that will receive aSelector (SEL).

TABLE 9-33. NSApplication Window Management

<code>keyWindow</code> <code>mainWindow</code>	Returns an <code>NSWindow*</code> for the key or main window. If there key or main window belongs to another application, nil is returned; nil is also returned if there is no key window (for <code>keyWindow</code>) or if the main window (for <code>mainWindow</code>) is hidden.
<code>windows</code>	Returns an <code>NSArray*</code> containing the application's windows.
<code>makeWindowsPerform</code>	<p>Sends <code>aSelector</code> (SEL) to each window in the application until one returns a non-nil value. (The <code>aSelector</code> method takes no arguments.) The second argument, <code>inOrder</code> (BOOL) is YES for front-to-back ordering.</p> <p>Without overriding <code>NSWindow</code>, you can add simple utility functions to your windows to allow them to respond to a query that you send with <code>makeWindowsPerform</code>. Add a customized delegate to your standard <code>NSWindow</code> objects and have it respond to the <code>aSelector</code> message that you will use in <code>makeWindowsPerform</code>.</p>

TABLE 9-33. NSApplication Window Management (Continued)

<code>setWindowsNeedUpdate</code> <code>updateWindows</code>	<p>The <code>updateWindows</code> message is sent automatically after each event is processed in your main event loop; it updates windows and menus as necessary.</p> <p>The <code>setWindowsNeedsUpdate</code> method, which has a single argument <code>flag</code> (BOOL) argument, lets you request an update at the end of the current event cycle; you do this if a change has occurred that was not the result of an event within your application (i.e., the event loop won't know to send an <code>updateWindows</code> message).</p>
<code>miniaturizeAll</code>	<p>Miniaturizes all the application's windows. The only argument is <code>sender</code>.</p>
<code>preventWindowOrdering</code>	<p>Prevents the window receiving a mouse-down event from being moved to the front.</p>
<code>hide</code> <code>unhide</code>	<p>Hide or unhide all of the application's windows. Takes a single <code>sender</code> argument.</p>
<code>isHidden</code>	<p>Returns a BOOL letting you know if the application is hidden.</p>
<code>unhideWithoutActivation</code>	<p>This method doesn't activate the application but does unhide its windows. There are no arguments.</p>

Modality is the bane of existence for many users and developers. A simple window displaying a trivial question—or comment—can suddenly bring the entire system to a screeching halt. Perhaps the classic example is shown in many network

communication packages. Most offer a timeout feature: if nothing has happened for a period of time, the network connection is broken. For people who pay for network connections on a per-minute basis, this is a very valuable feature. Of course, when the helpful application puts up a dialog that says, “Are you sure you want to disconnect from the network?” and leaves it there until it gets a response, the value of the automatic disconnection is somewhat dissipated, since the connection remains alive until the dialog is dismissed (perhaps first thing the next morning).

Modality requires a direct interaction between the user and the computer—on the computer’s terms. It can make scripting and automation of tasks difficult, and it moves the locus of control from the user to the computer.

However, if you must be modal, Table 9-34 shows you the OpenStep support for modality.

TABLE 9-34. NSApplication Modality

`runModalForWindow`

Starts a modal event loop for `aWindow` (`NSWindow*`); you stop it with `stopModal` or `abortModal`. Until stopped, the application responds only to events associated with `aWindow`. See how dangerous this is?

`stopModal`
`stopModalWithCode`
`abortModal`

Stops or aborts (raising an exception) the current modal loop; `stopModalWithCode` takes a single `code` (`int`) argument.

TABLE 9-34. NSApplication Modality (Continued)

<code>runModalForWindow</code>	Starts a modal event loop for <code>aWindow</code> (<code>NSWindow*</code>); you stop it with <code>stopModal</code> or <code>abortModal</code> . Until stopped, the application responds only to events associated with <code>aWindow</code> . See how dangerous this is?
<code>beginModalSessionForWindow</code>	Instead of <code>runModalForWindow</code> , this method returns an <code>NSModalSession*</code> for <code>aWindow</code> (<code>NSWindow*</code>). You start and stop the sessions with <code>runModalSessionForWindow</code> and <code>endModalSession</code> .
<code>runModalSession</code> <code>endModalSession</code>	Starts and stops <code>session</code> (<code>NSModalSession*</code>).

In Table 9-35 you will find the methods that support the standard panels—the color panel, the data link panel, and the page layout panel. In all cases, the panel is created if it does not already exist; also, each method takes the single argument `sender`.

TABLE 9-35. NSApplication Panels

<code>orderFrontColorPanel</code>	<code>sender</code>
<code>orderFrontDataLinkPanel</code>	<code>sender</code>
<code>runPageLayout</code>	<code>sender</code>

Menu Management Menus have been (and remain) a primary source of user input to a graphical user interface. Over the last decade, floating palettes and other interface elements

such as hypertext links have brought functionality closer to the user's focus of attention; to some people, moving the mouse away from the focus of attention to a menu is as distracting as working with a command line.

Nevertheless, menus are usually indispensable. Devour everything that you can find on interface design that has anything to do with menus; watch yourself as you struggle with awkward menus in the applications that you use. Poor menu design is one of the most obvious—and annoying—faults of software.

Each application has a main menu and a Window menu. The main menu contains both commands and submenus. As such, it is a cross between a Mac OS menu and the menubar. It looks like a single menu with its commands and submenus; however, since there is only a single application menu, it has much in common with the menubar. (Note that the windows menu can be nil if it has not been created.)

Services are provided by applications to provide their functionality to users who are running other applications. You can set your own Services menu; you also register a service provider—that is the object that receives requests for services from the outside world. In order to use other applications' services, you must register the types that your application can send and receive via the pasteboard to services.

The `NSApplication` menu management routines are shown in Table 9-36. The Window menu methods are normally called for you by `NSWindow`.

TABLE 9-36. NSApplication Managing Menus

<code>setMainMenu</code> <code>mainMenu</code> <code>setWindowsMenu</code> <code>windowsMenu</code>	<p>Accessor for the application's main and windows menus.</p> <p>Set: <code>aMenu</code> (NSMenu*). Get: returns an NSMenu*.</p>
<code>arrangeInFront</code>	<p>Places all the windows in the Window menu in front of the other application windows. Takes a single argument <code>sender</code>.</p>
<code>addWindowsItem</code> <code>changeWindowsItem</code>	<p>Adds or changes an entry in the Window menu for <code>aWindow</code> (NSWindow*) with title <code>aString</code> (NSString*) to the Window menu. If <code>isFilename</code> (BOOL) is YES, a String is a converted path name.</p>
<code>removeWindowsItem</code> <code>updateWindowsItem</code>	<p>Removes or updates the editing status of <code>aWindow</code> (NSWindow*) in the Window menu.</p>
<code>setServicesMenu</code> <code>servicesMenu</code>	<p>Accessor for Services menu.</p> <p>Set: <code>aMenu</code> (NSMenu*) Get: returns an NSMenu.</p>
<code>setServiceProvider</code> <code>servicesProvider</code>	<p>Accessor for services provider.</p> <p>Set: <code>aProvider</code>. Get: returns an id.</p>
<code>registerServicesMenu</code> <code>SendTypes</code>	<p>Lists the types that can be sent and received via the pasteboard to services; the arguments are <code>sendTypes</code> (NSArray*) and <code>returnTypes</code> (NSArray*). You can call this method as you encounter new types that you are dealing with in your application that may need services.</p>

TABLE 9-36. NSApplication Managing Menus (Continued)

<code>validRequestorForSendType</code>	Returns the id of an object (or nil) that can process <code>sendType</code> (NSString*) and <code>returnType</code> (NSString*). NSApplication gets this message only after all other objects in the responder chain have been asked (i.e., views and windows are asked before the application if they can handle the send/receive type combination).
--	---

NSBundleAdditions

Application bundles (Table 9-37) are part of the basic architecture; they contain resources and objects from nib files. For applications built with the Application kit, this class and these methods are available (and are used internally).

TABLE 9-37. NSBundleAdditions Methods

<code>+loadNibFile</code>	Loads objects from <code>fileName</code> (NSString*); they can then use <code>context</code> (NSDictionary*) as needed. The objects are allocated in <code>zone</code> (NSZone*). A BOOL result is returned indicating success or failure.
<code>+loadNibNamed</code>	Similar to <code>loadNibNamed</code> , this method creates a dictionary with a single entry that has the key <code>NSOwner</code> and the value <code>owner</code> (id).
<code>pathForResource</code>	Returns the full path for the image resource <code>name</code> (NSString*) located within the bundle.

**NSNib-
Awaking[P]**

The single method in this protocol (Table 9-38) is called after objects have been loaded from a nib file. You can use this method to do postinstantiation processing.

TABLE 9-38. NSNibAwaking Method

`awakeFromNib`

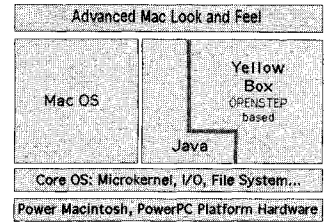
Summary

These are the building blocks of OpenStep. The programming tools—dates, numbers, values, arrays, enumerators, etc.—you use routinely and explicitly. `NSApplication` (reached through the global variable `NSApp`) is used explicitly (if at all) as a means to get to other services. Normally, OpenStep objects such as windows call the `NSApplication` methods for you.

Remember in constructing your application that a custom-built delegate object can be used to handle application-specific data and functionality needs. Before you subclass `NSApplication`, make certain that you can't do what you want with a delegate.

Chapter 10

Visualization



This chapter covers the basic visualization classes and methods of the framework. Display PostScript is used to draw on the monitor; PostScript is used to print output (whether on printers or other devices). Programmers from Mac OS are familiar with the QuickDraw libraries; Display PostScript provides comparable functionality. However, since Display PostScript uses the same routines that are used in output to printers, there is no conversion process at that step: the results on the screen and on the printer are much closer to one another.

In comparing images on the screen with printed pages, remember that you are dealing with two quite different ways of presenting images. The reflected images of a printed page are created and perceived

differently from the transmitted images on a monitor. Colors on the two media may be physically the same (as measured by a spectrometer), but they are often perceived differently.

PostScript (including Display PostScript) has significantly more image manipulation powers than QuickDraw—particularly with regard to text. Curved lines of text can easily be created and manipulated, for example.

For much of your work, you need not know much if anything about PostScript: after all, if the framework doesn't insulate you from the specifics of an implementation, what good is it? One PostScript concept that you might need to know is the drawing context—it's not a QuickDraw grafport, but it's awfully close. It is the set of values (pen color, coordinate space, etc.) that are used for current drawing operations.

The heart of any framework's visualization is its windows and views. The NSWindow and NSView objects are similar to those you may have worked with before. Of particular interest, however, are two objects that are new in this framework: cells, and images. Cells and images are not views; they live within views and are very lightweight and targeted to specific purposes. This architecture is repeated in other areas where a complex object (such as a menu) has common functionality and where a small object can provide the customized functionality to the larger one.

Color

For most of your color needs, the framework provides the range of functionality that you expect from any sophisticated graphical user interface framework. The use of Display PostScript on the screen and PostScript for printing means that you have a much closer commonality between the images on the two media.

Don't fool yourself into thinking that in fact you have identical images on screen and printer: transmitted light (screen)

and reflected light (paper) are very different. Furthermore, the colors that you see—even if they are the same—may appear different because the interactions among the colors in different media are varied.¹

Fortunately, for most of your work—including very high-end sophisticated graphics—these issues do not concern you.

NSColor

As with other drawing environments, the framework's color model relies on identifying colors by triple values—often red, green, and blue values that combine to make a single color. It adds an alpha value which is a measure of opacity. Alpha has meaning on some devices (computer monitors, for example); on media such as printed paper, opacity is simulated by the use of different colors.

You actually use these color routines fairly infrequently. The `NSColorPanel` allows users to choose colors to use: you simply record the values that the user has chosen. Similarly, if you are using Interface Builder (which you should be doing), you can use the Color Panel to set the colors for your interface objects. Again, no direct text-oriented programming work is needed.

Colors have a color space associated with them—it may be based on the RGB (red/green/blue or hue/saturation/brightness) model or on the CMYK (cyan/magenta/yellow/black) model; gray-scale colors are in white color spaces. Each color space may have an alpha component; it may also be calibrated (more or less device-independent) or it may be specific to a given device. An additional color space—a named color space—contains within it colors that are named using system defaults (purple, brown, etc.).

Table 10-1 shows the methods of `NSColor`.

1. The definitive work on this subject is Josef Albers's *Interaction of Color*. It is published by Yale University Press in an abridged paperback and on CD-ROM.

TABLE 10-1. NSColor Methods

<code>+colorWithCalibratedHue</code> <code>+colorWithDeviceHue</code>	Creates and returns an <code>NSColor*</code> based on <code>hue</code> (float), <code>saturation</code> (float), <code>brightness</code> (float), and <code>alpha</code> (float). The first returns a calibrated color; the second a device-dependent color.
<code>+colorWithCalibratedRed</code> <code>+colorWithDeviceRed</code>	Creates and returns an <code>NSColor*</code> based on <code>red</code> (float), <code>green</code> (float), <code>blue</code> (float), and <code>alpha</code> (float). The first returns a calibrated color, the second a device-dependent color.
<code>+colorWithCalibrated</code> <code>white</code> <code>+colorWithDeviceWhite</code>	Creates and returns an <code>NSColor*</code> based on <code>white</code> (float) which is its gray-scale value and <code>alpha</code> (float). The first returns a calibrated color, the second a device-dependent color.
<code>+colorWithCatalogName</code>	Creates and returns an <code>NSColor*</code> from the catalog <code>listName</code> (<code>NSString*</code>), with <code>colorName</code> (<code>NSString*</code>); the color is created in the <code>NSNamedColorSpace</code> .
<code>+darkGrayColor</code> <code>+grayColor</code> <code>+lightGrayColor</code> <code>+whiteColor</code>	These routines all return <code>NSColor*</code> objects with the appropriate values; they all include an <code>alpha</code> value of 1.0 and are in the <code>NSCalibratedWhiteColorSpace</code> .
<code>+blackColor</code> <code>+blueColor</code> <code>+brownColor</code> <code>+cyanColor</code> <code>+greenColor</code> <code>+magentaColor</code> <code>+orangeColor</code> <code>+purpleColor</code> <code>+redColor</code> <code>+yellowColor</code>	These routines all return <code>NSColor*</code> objects with the appropriate values; they all include an <code>alpha</code> value of 1.0 and are in the <code>NSCalibratedRGBColorSpace</code> .

TABLE 10-1. NSColor Methods (Continued)

<code>+clearColor</code>	This returns an <code>NSColor*</code> with an alpha value of 0.0.
<code>+colorFromPasteboard</code>	Returns the <code>NSColor*</code> from pasteBoard (<code>NSPasteBoard*</code>); nil if there is no color on the pasteboard supplied. (Note: this is a color that has been placed on the pasteboard, not the color of an object on the pasteboard.)
<code>writeToPasteboard</code>	Writes the color to pasteBoard (<code>NSPasteBoard*</code>).
<code>cyanComponent</code> <code>magentaComponent</code> <code>yellowComponent</code> <code>blackComponent</code>	Returns the specified component as a float from a CMYK color.
<code>redComponent</code> <code>greenComponent</code> <code>blueComponent</code>	Returns the specified component as a float from an RGB color.
<code>hueComponent</code> <code>saturationComponent</code> <code>brightnessComponent</code>	
<code>whiteComponent</code>	Returns the specified component as a float from a gray-scale color.
<code>getCyan</code>	For a CYMK color, sets arguments to cyan (float*), magenta (float*), yellow (float*), black (float*), and alpha (float*).
<code>getRed</code>	For an RGB color, sets arguments to red (float*), green (float*), blue (float*), and alpha (float*).
<code>getHue</code>	For an RGB color, sets arguments to hue (float*), saturation (float*), brightness (float*), and alpha (float*).

TABLE 10-1. NSColor Methods (Continued)

<code>getWhite</code>	For a gray-scale color, sets arguments to <code>white</code> (float*) and <code>alpha</code> (float*).
<code>setIgnoresAlpha</code> <code>ignoresAlpha</code>	If you do not want to allow alpha (opacity), use these accessors. Set: <code>flag</code> (BOOL). Get: returns a BOOL.
<code>alphaComponent</code>	Returns the color's alpha component as a float; if not set, it is 1.0.
<code>catalogNameComponent</code> <code>localizedCatalogNameComponent</code>	For a color in a named color space, returns an NSString*, which is the color's catalog in standard or localized versions.
<code>colorNameComponent</code> <code>localizedColorNameComponent</code>	For a color in a named color space, returns an NSString*, which is the color's name in standard or localized versions.
<code>colorSpaceName</code>	Returns the name of the color's color space as an NSString*.
<code>colorUsingColorSpaceName</code>	Returns a new color based on this object but located in <code>colorSpace</code> (NSString*). The optional second argument, <code>deviceDescription</code> (NSDictionary*), lets you specify a device-specific color.
<code>blendedColorWithFraction</code>	Creates and returns a new NSColor* object based on the sum of the RGB values of this object and the <code>fraction</code> (float) of each of the RGB values for <code>color</code> (NSColor*).
<code>colorWithAlphaComponent</code>	Returns a new NSColor* object based on the current object but with <code>alpha</code> (float) set to the value supplied.

TABLE 10-1. NSColor Methods (Continued)

<code>drawSwatchInRect</code>	Draws the color in <code>rect</code> (NSRect). This method is used for user interface elements.
<code>set</code>	Sets this color to be the color for subsequent PostScript calls.

NSColorList

Color lists are pairs of strings and NSColor objects that are suitable for displaying in interface elements. You (and your users) probably use color lists all the time—but you rarely bother about them and code them even less often.

Color lists live in files; since the default routines automatically load color lists, you can expand the system's repertoire just by adding files and not worrying about code.

The methods are shown in Table 10-2.

TABLE 10-2. NSColorList Methods

<code>initWithName</code>	Initializes a color list using <code>name</code> (NSString*) and reading in from <code>path</code> (NSString*), which can be nil for an empty list.
<code>+availableColorLists</code>	Returns an NSArray* of system color lists.
<code>+colorListNamed</code>	Returns an NSColorList* with <code>name</code> (NSString*) from available-ColorLists.

TABLE 10-2. NSColorList Methods (Continued)

<code>name</code>	Returns the name of the color list as an <code>NSString*</code> .
<code>allKeys</code>	Returns an <code>NSArray*</code> of the color name keys from the list in the order in which they were inserted.
<code>colorWithKey</code> <code>removeColorWithKey</code>	Returns (or removes) the <code>NSColor*</code> with key (<code>NSString*</code>).
<code>insertColor</code>	Inserts color (<code>NSColor*</code>) into the list with key (<code>NSString*</code>) at location (unsigned). As with all numbering in the framework, it starts at zero. The name must be unique in the color list; if it already exists as a key, the prior key/color combination is removed.
<code>setColor</code>	Sets color (<code>NSColor*</code>) for an existing key (<code>NSString*</code>); if key does not exist, this adds the color/key combination to the end of the list.
<code>isEditable</code>	Returns a <code>BOOL</code> indicating if the list is editable (which it is unless it is from a nonwritable file).
<code>removeFile</code>	Removes the file from which the list was read.
<code>writeToFile</code>	Writes the list to file path (<code>NSString*</code>) and returns a <code>BOOL</code> indicating success.

Controls

Boxes and cells are lightweight objects that are targeted to specific needs. `NSBox` is a descendant of `NSView`; `NSCell` is not (but it is designed to live within an `NSView`).

`NSBox`

An `NSBox` lives within a view and is used to group views (which are usually related views, such as buttons). Boxes have borders and titles—but most important they have a content view into which you can add subviews. They are useful objects that can save you significant amounts of coding time. Views (which you place within an `NSBox`) are discussed later in this chapter; `NSCells`—which are used to display the text of an `NSBox`'s title—are discussed in the following section.

They are in many ways comparable to the MacApp `TCluster` object. However, you will notice from the values for `NSTitlePosition` that they are able to draw outside their borders (e.g., a title can be above the box). Confining a view's drawing to its interior has long been a problem for some interface designers.

The methods of `NSBox` are shown in Table 10-3.

TABLE 10-3. `NSBox` Methods

`setBorderType`
`borderType`

Accessors for the box's border type: `NSNoBorder`, `NSLineBorder`, `NSBezelBorder`, and `NSGrooveBorder`.

Set: `aType` (`NSBorderType`).
Get: returns an `NSBorderType`.

`borderRect`

Returns the border rectangle as an `NSRect`.

TABLE 10-3. NSBox Methods (Continued)

setTitle setTitleWithMnemonic title	Sets the title to aString (NSString*); the second method assumes that the & symbol precedes the mnemonic character. title returns an NSString*.
setTitleFont titleFont	Set: aFont (NSFont*). Get: returns an NSFont.
setTitlePosition titlePosition	Accessors for the box's title position: NSNoTitle, NSAboveTop (default), NSAtTop, NSBelowTop, NSAboveBottom, NSAtBottom, and NSBelowBottom. Set: aPosition (NSTitlePosition). Get: returns an NSTitlePosition
titleCell	Returns the box's text-based NSCell for its title.
setContentView contentView	Set: aView (NSView*). Get: returns an NSView*.
setContentViewMargins contentViewMargins	Set: offsetSize (NSSize). Get: returns an NSSize.
setFrameFromContentFrame	Moves the NSBox so that its content view is at the origin of contentFrame (NSRect).
sizeToFit	The box's content view and the box itself are changed to enclose the subviews and to display the box's title with as little wasted space as possible. Call this method after modifying the box's title or subviews.

NSCell

Cells are lightweight objects that have many of the characteristics of views without their overhead. Cells are customized to

display text and images; they manage some of the typical view functions such as mouse tracking, text editing, highlighting, selection, and the storing of values.

Because of their specialized purpose, cells have far more attributes than do views. Not only does this save you from re-implementing common code for drawing borders, etc., it also preserves the look and feel of Rhapsody across various environments with a minimum of effort.

Because cells are not views, they can do some things that are a little unusual: for example, a cell can not only track the mouse—it can track the mouse beyond its borders (although this is rare). The major descendants of `NSCell` are `NSImageCell`, `NSTextFieldCell`, and `NSActionCell`—the last of which is more concerned with managing events than contents. As is common in the framework, methods that apply only to a single subclass are actually declared in the abstract superclass. Thus, you will find an action method in `NSCell`—although it is normally implemented only by `NSActionCell` and its subclasses. One by-product of this is that the framework is able to convert cells on demand from one type to another. If you call `setStringValue`, for example, it converts a nontext cell to a text cell and sets its string value to `aString`. (Converting a cell to a text cell may have other consequences: the title may be changed to “Cell” and the font may be changed to the system font at 12 points. In general, you should keep track of what kinds of cells you are using and avoid automatic conversions.)

While not being views, `NSCells` almost always live within views (which is how they can partake of some viewlike behavior).

`NSCells` come with a data structure prepared for them that consists of (potentially) both an object and a represented object. The object is what the view contains; the optional represented object is what that object represents. The most common example of this is a cell whose object is a string and whose represented object is what that string is—“blue”/an

NSColor* that is blue, a thumbnail image/ the document containing that image, etc.

NSCell Fabrication Methods Table 10-4 lists the fabrication and initialization methods for NSCells.

TABLE 10-4. NSCell Fabrication Methods

<code>initWithImageCell</code>	Initializes a cell with anImage (NSImage*) as its object.
<code>initWithTextCell</code>	Initializes a cell with aString (NSString*) as its object.
<code>takeIntValueFrom</code> <code>takeStringValueFrom</code> <code>takeDoubleValueFrom</code> <code>takeFloatValueFrom</code> <code>takeObjectValueFrom</code>	These are all action methods with the argument sender (id); you normally use them to wire interface elements in Interface Builder.

Although NSCells are not views (and not even descendants of NSResponder), they have the ability to manage a number of actions and typical responder-type processes, as shown in Table 10-5.

TABLE 10-5. NSCell Action and Responder Methods

<code>+defaultMenu</code>	The class method returns nil; if your subclass of NSCell has its own menu, return it as an NSMenu*.
---------------------------	---

TABLE 10-5. NSCell Action and Responder Methods (Continued)

<code>acceptsFirstResponder</code>	Returns a BOOL. (Note that since NSCell is not a descendant of NSResponder, it must explicitly declare this method in order to behave in some circumstances like a responder.)
<code>performClick</code>	An action method with <code>sender (id)</code> as argument. It simulates a mouse click on the cell; you can wire it to another interface element (perhaps transferring a mouse click from another place to the cell).
<code>trackMouse</code>	Returns a BOOL indicating if the mouse button goes up at the end of tracking within the cell. The arguments are <code>theEvent (NSEvent*)</code> —the originating mouse event, <code>cellFrame (NSRect)</code> , <code>controlView (NSView*)</code> , and <code>flag (BOOL)</code> . If <code>flag</code> is true, tracking continues even if it goes beyond the borders of the cell.
<code>startTrackingAt</code>	Called by <code>trackMouse</code> , this method returns a BOOL if the receiver responds continuously. The arguments are <code>startPoint (NSPoint)</code> and <code>controlView (NSView*)</code> .
<code>continueTracking</code>	Called by <code>trackMouse</code> , this method returns BOOL indicating if tracking should continue. The arguments are <code>lastPoint (NSPoint)</code> , <code>currentPoint (NSPoint)</code> , and <code>controlView (NSView*)</code> .

TABLE 10-5. NSCell Action and Responder Methods (Continued)

<code>stopTracking</code>	Called by <code>trackMouse</code> , this method's arguments are <code>lastPoint</code> (NSPoint), <code>stopPoint</code> (NSPoint)—the current point, <code>controlView</code> (NSView*), and <code>flag</code> (BOOL) indicating if the mouse button is up.
<code>mouseDownFlags</code>	Returns an int with the last mouse-down flags for this cell.
<code>tprefersTrackingUntilMouseUp</code>	This class method returns NO—the cell stops worrying about tracking when the mouse leaves it; if your subclass of NSCell wants to continue tracking beyond the bounds of the cell, override this method and return YES.
<code>calcDrawInfo</code>	Override this if necessary to prepare for redrawing the cell's <code>aRect</code> (NSRect) area.
<code>cellSize</code> <code>cellSizeForBounds</code>	Returns an NSSize, which is the smallest size to display the cell based on its current contents and border type. The second method takes an <code>aRect</code> (NSRect) argument: for text cells, the contents are resized within that rectangle before the cell size is returned.
<code>drawingRectForBounds</code> <code>imageRectForBounds</code> <code>titleRectForBounds</code>	Given a bounds rectangle <code>theRect</code> (NSRect), these methods return the appropriate NSRect, which takes into account the current border.

TABLE 10-5. NSCell Action and Responder Methods (Continued)

drawWithFrame <i>drawInteriorWithFrame</i>	Draws the cell in <code>cellFrame</code> (NSRect) inside <code>controlView</code> (NSView*); <code>cellFrame</code> is in the <code>controlView</code> 's coordinates.
	The first method calls the second
<code>controlView</code>	Returns the NSView* (possibly nil) in which this NSCell was last drawn.
<code>highlight</code>	Use this method to change the cell's highlighting to <code>flag</code> (BOOL) and to redraw it in <code>cellFrame</code> (NSRect) inside <code>controlView</code> (NSView*). If <code>flag</code> doesn't represent a change, no drawing takes place.
<code>isHighlighted</code>	Returns a BOOL.

Since cells are usually placed within views (often controls), their drawing and tracking coordinates are often expressed in the coordinates of their containing views. You can recognize these methods by their inclusion of an NSView* argument (usually named `controlView`).

General Cell Accessors Whether they are text, image, or action cells, all cells share the accessors listed in Table 10-6.

TABLE 10-6. NSCell General Accessors

<code>setEnabled</code>	Set: <code>flag</code> (BOOL).
<code>isEnabled</code>	Get: returns a BOOL.

TABLE 10-6. NSCell General Accessors (Continued)

<code>setBordered</code> <code>setBezeled</code> <code>isBordered</code> <code>isBezeled</code>	<p>Cells can be either bordered or bezeled (or neither).</p> <p>Set: flag (BOOL). Get: returns a BOOL.</p>
<code>isOpaque</code>	<p>Returns a BOOL.</p>
<code>setObjectValue</code> <code>objectValue</code>	<p>This is the cell's own object (such as a string or an image) not its represented object.</p> <p>Set: object (id). Get: returns an id.</p>
<code>setRepresentedObject</code> <code>representedObject</code>	<p>Set: object (id). Get: returns an id.</p>
<code>setMenu</code> <code>menu</code>	<p>If the cell has a menu associated with it, this is its accessor. Many cells have no menus of their own.</p> <p>Set: aMenu (NSMenu*). Get: returns an NSMenu*.</p>
<code>menuForEvent</code>	<p>Returns an NSMenu* for anEvent (NSEvent*) and cellFrame (NSRect) inside aView (NSView*). This lets you return a specific menu based on the event that has been received.</p>
<code>setShowsFirstResponder</code> <code>showsFirstResponder</code>	<p>Cells can optionally show a dotted outline when they are first responders.</p> <p>Set: flag (BOOL). Get: returns a BOOL.</p>

TABLE 10-6. NSCell General Accessors (Continued)

<code>setTitleWithMnemonic mnemonic</code>	<p>You can set a title with a mnemonic embedded in it; the & symbol precedes the character to be used as the mnemonic.</p> <p>Set: <code>aString</code> (NSString*). Get: returns an NSString*.</p>
<code>setMnemonicLocation mnemonicLocation</code>	<p>Where mnemonics are supported, this is the accessor for the mnemonic—the <i>n</i>th character in the cell's title.</p> <p>Set: location (unsigned int). Get: returns an unsigned int.</p>
<code>setType type</code>	<p>The values for <code>aType</code> are <code>NSTextTypeCell</code>, <code>NSImageTypeCell</code>, and <code>NSNullCellType</code>. Conversions of other cells to image cells store a nil image; conversions of other cells to text cells set the font and title to defaults.</p> <p>Set: <code>aType</code> (int). Get: returns an int.</p>
<code>setCellAttribute cellAttribute</code>	<p>Attributes can be accessed using <code>NSCellAttributes</code>; each is an int.</p> <p>Set: <code>aParameter</code> (NSCellAttribute), <code>value</code> (int). Get: returns an int for <code>aParameter</code> (NSCellAttribute).</p>
<code>setState state</code>	<p>Although it is an int, the state can only have a value of 1 or 0. It is set based on the sign of the value that is passed in.</p> <p>Set: <code>state</code> (int). Get: returns an int.</p>

Text Accessors Text-based NSCells have their own methods and accessors as shown in Table 10-7. Note that these are methods of NSCell, but they apply only to text-based cells. Setting some of these text-cell-only attributes changes the cell to a text-based cell (these are marked in the table).

TABLE 10-7. NSCell Accessors for Text

setSelectable setEditable setScrollable setWraps isSelectable isEditable isScrollable wraps	Note: scrollable and wraps are mutually exclusive. Set: flag (BOOL). Get: returns a BOOL.
setStringValue stringValue	Set: aString (NSString*). Get: returns an NSString*.
setIntValue intValue	Set: anInt (int). Get: returns an int.
setDoubleValue doubleValue	Set: aDouble (double). Get: returns a double.
setFloatValue floatValue	Set: aFloat (float). Get: returns a float.
setFormatter formatter	Set: aFormatter (NSFormatter*). Get: returns an NSFormatter*.
isEntryAcceptable	Returns a BOOL indicating if aString (NSString*) passes the formatter.

TABLE 10-7. NSCell Accessors for Text (Continued)

<code>setFloatingPointFormat</code>	<p>You can control formats for floating point numbers. You specify the maximum number of digits to the left of the decimal point and the absolute number of digits to the right. You further specify if autoranging is allowed—autoranging allows numbers that don't fit to be reformatted by simply adding the total number of digits on both sides of the decimal point and displaying the number in the best possible way.</p> <p>Arguments are <code>autoRange</code> (BOOL), <code>leftDigits</code> (unsigned), and <code>rightDigits</code> (unsigned).</p>
<code>editWithFrame</code>	<p>This method places an NSText object over the cell and starts editing. The arguments are <code>aRect</code> (NSRect), <code>controlView</code> (NSView*), <code>textObj</code> (NSText*), <code>anObject</code> (id)—the text object's delegate, and <code>theEvent</code> (NSEvent*)—whatever started this.</p>
<code>selectWithFrame</code>	<p>The text object can be used to highlight and select text at will with this method (no event or mouse click is involved). The arguments are <code>aRect</code> (NSRect), <code>controlView</code> (NSView*), <code>textObj</code> (NSText*), <code>anObject</code> (id), <code>selStart</code> (int), and <code>selLength</code> (int).</p>
<code>endEditing</code>	<p>Ends editing in the cell that is going on with <code>textObj</code> (NSText*).</p>

TABLE 10-7. NSCell Accessors for Text (Continued)

<code>setUpFieldEditorAttributes</code>	Sets text colors and other attributes to those of <code>textObj</code> (<code>NSText*</code>) and returns them as an <code>NSText*</code> object.
<code>resetCursorRect</code>	Sets the text cursor for <code>cellFrame</code> (<code>NSRect</code>) in <code>controlView</code> (<code>NSView*</code>).
<code>hasValidObjectValue</code>	Returns a <code>BOOL</code> . Validity here means the decision of the cell's formatter object: an invalid object may be a perfectly fine object, but it may be invalid because its format or value is unacceptable.
<code>setEntryType</code> <code>entryType</code>	The constants that you should use are <code>NSIntType</code> , <code>NSPositiveIntType</code> , <code>NSFloatType</code> , <code>NSPositiveFloatType</code> , <code>NSDoubleType</code> , <code>NSPositiveDoubleType</code> , and <code>NSAnyType</code> . Set: <code>aType</code> (<code>int</code>). Get: returns an <code>int</code> .
<code>setAlignment</code> <code>alignment</code>	Set: <code>mode</code> (<code>NSTextAlignment</code>). Get: returns an <code>NSTextAlignment</code> .
<code>setFont</code> <code>font</code>	Set: <code>fontObj</code> (<code>NSFont*</code>). Get: returns an <code>NSFont*</code> .
<code>setAttributedStringValue</code> <code>attributedStringValue</code>	Attributed strings are strings that carry attributes (fonts, colors, sizes, etc.) along with them. Set: <code>attribStr</code> (<code>NSAttributedString*</code>). Get: returns an <code>NSAttributedString*</code> .

TABLE 10-7. NSCell Accessors for Text (Continued)

<code>setImportsGraphics</code> <code>setAllowsEditingText</code> Attributes <code>importsGraphics</code> <code>allowsEditingText</code> Attributes	<p>The <code>importsGraphics</code> flag is a superset of <code>allowsEditingTextAttributes</code>. Either implies that the text cell supports RTFD text.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>compare</code>	<p>Returns an <code>NSComparisonResult</code> based on this cell and the text cell passed in <code>otherCell</code> (id).</p>

Image Accessors Similarly, the methods in Table 10-8 are methods of `NSCell`, but they apply only to image-based cells.

TABLE 10-8. NSCell Image Accessors

<code>setImage</code> <code>image</code>	<p>Set: <code>image</code> (NSImage*).</p> <p>Get: returns an NSImage*.</p>
---	---

Action Accessors Action cells have a number of accessors that are listed in Table 10-9. The Tag is a means for you to identify a given action cell. The three methods control its actions. The action method is sent to the target periodically (`NSPeriodicMaskSend`) or in response to specific events (`NSLeftMouseDownMaskSend` or `NSLeftMouseDraggedMaskSend`).

TABLE 10-9. NSCell Action Accessors

<code>setTag</code> <code>tag</code>	Set: <code>anInt (int)</code> . Get: returns an <code>int</code> .
<code>setAction</code> <code>action</code>	Set: <code>aSelector (SEL)</code> . Get: returns a <code>SEL</code> .
<code>setTarget</code> <code>target</code>	Set: <code>anObject (id)</code> . Returns an <code>id</code> .
<code>sendActionOn</code>	Sets the action bits for the cell to <code>mask (int)</code> and returns the previous value as an <code>int</code> . The values are given in the preceding text.
<code>setContinuous</code> <code>isContinuous</code>	Set: <code>flag (BOOL)</code> . Get: returns a <code>BOOL</code> .
<code>getPeriodicDelay</code>	The arguments <code>delay (float*)</code> and <code>interval (float*)</code> specify the time before the first action being sent and the timing of subsequent actions. This methods serves to access the values in both directions (set and get).

Images

The biggest difference between imaging in this framework and imaging in other frameworks is the coordinate system that is used. In both cases, the horizontal axis running from left to right starts at zero and goes up. In the case of the Mac OS, however, the value of zero on the vertical access is at the top of the screen; in OpenStep, it is at the bottom. As a result, Mac OS objects are often located on the screen (or in a view) by means of their top left point; in OpenStep, it is often the lower left point.

Drawing normally takes place within views. Images are lightweight objects that you can draw without using a view. The image thus drawn can then be quickly moved to a view for display or printing.

Images can be in various formats; a single image may contain multiple representations of the same image using different formats. Each representation is an instance of a descendant of `NSImageRep`.

If you are coming from another framework, you can realize the efficiencies that images offer you: you have very little overhead and you don't have to worry about the representation.

NSImage

`NSImage` is responsible for managing all of its representations and for selecting (and drawing) the appropriate one for a given device. For convenience, images may be named.

Each `NSImage` may have a delegate that implements `imageDidNotDraw(aRect—NSRect)`. This method is called only if `NSImage` was unable to draw the image. It returns an `NSImage*`—which could be an alternate image to draw. If it returns `nil`, `NSImage` no longer tries to draw the image, assuming either that the delegate has done so or that no image should be drawn.

The methods of `NSImage` are shown in Table 10-10.

TABLE 10-10. NSImage Methods

`initWithReferencingFile`

Initializes the `NSImage` from `filename` (`NSString*`). The file is not actually read at this time.

TABLE 10-10. NSImage Methods (Continued)

<code>initWithContentsOfFile</code>	Initializes the NSImage from <code>fileName</code> (NSString*) and does read it at this time.
<code>initWithData</code>	Initializes the NSImage from data (NSData*).
<code>initWithPasteboard</code>	Initializes the NSImage from <code>pasteboard</code> (NSPasteboard*).
<code>initWithSize</code>	Initializes the NSImage to <code>aSize</code> (NSSize) without giving it any image.
<code>setSize</code> <code>size</code>	Accessors for the image's size. If not explicitly set, it is set from the first representation. Set: <code>aSize</code> (NSSize). Get: returns an NSSize.
<code>imageNamed</code>	Returns an image named <code>name</code> (NSString*). The constant <code>NSApplicationIcon</code> can be used to get the application's icon.
<code>setName</code> <code>name</code>	Accessors for the image's name. Set: <code>aName</code> (NSString*). Get: returns an NSString*.
<code>addRepresentation</code> <code>addRepresentations</code>	Adds <code>imageRep</code> (NSImageRep*) or an array of <code>imageReps</code> (NSArray*) to the NSImage object.
<code>removeRepresentation</code>	Removes <code>imageRep</code> (NSImageRep*).

TABLE 10-10. NSImage Methods (Continued)

lockFocus lockFocusOnRepresentation unlockFocus	<p>These methods lock and unlock the focus for either the best representation or a specific representation—<code>imageRepresentation</code> (<code>NSImageRep*</code>). You do this in preparation for—and after—drawing.</p>
compositeToPoint	<p>Composites the image to <code>aPoint</code> (<code>NSPoint</code>) in the current view's coordinate system.</p> <p>An optional argument <code>aRect</code> (<code>NSRect</code>) specifies—in the image's coordinates—the rectangle to composite. The final argument, <code>op</code> (<code>NSCompositingOperation</code>) specifies the PostScript operation to use.</p>
dissolveToPoint	<p>Instead of compositing, this method uses the dissolve operator to dissolve the image to <code>aPoint</code> (<code>NSPoint</code>), optionally using <code>aRect</code> (<code>NSRect</code>) of the image; the final argument <code>aFloat</code> (float) is the amount of the image to be combined with the destination.</p>
setPrefersColorMatch prefersColorMatch	<p>Accessors to determine if matching should be primarily on color and secondarily on resolution (YES) or vice versa.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>

TABLE 10-10. NSImage Methods (Continued)

<code>setUsesEPSOnResolutionMismatch</code> <code>usesEPSOnResolutionMismatch</code>	<p>If no representations satisfy the criteria, these accessors let you determine if an EPS representation should be used.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>setMatchesOnMultipleResolution</code> <code>matchesOnMultipleResolution</code>	<p>Accessors to determine if resolution matching for a device and the image must be exact (YES) or if the resolutions can be an integer multiple of one another.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>bestRepresentationForDevice</code>	<p>Returns the image's <code>NSImageRep*</code> which is best for deviceDescription (NSDictionary*).</p>
<code>representations</code>	<p>Returns an NSArray* of the image's representations.</p>
<code>setCachedSeparately</code> <code>cachedSeparately</code>	<p>Accessors to determine if each representation has its own off-screen window or if they are all in one off-screen window.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>setDataRetained</code> <code>dataRetained</code>	<p>Accessors for whether the NSImage retains its data after the representation(s) is(are) created.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>

TABLE 10-10. NSImage Methods (Continued)

setCacheDepthMatches ImageDepth cacheDepthMatches ImageDepth	Accessors determining if the image's representations off screen use their own depth or the application's default. Set: <code>flag</code> (BOOL). Get: returns a BOOL.
isValid	Returns a BOOL if a representation can be drawn; if you used <code>initWithReferencingFile</code> , this will cause the file to be read.
setScalesWhenResized scalesWhenResized	Accessors for scaling of image representations to resized NSImages. Set: <code>flag</code> (BOOL). Get: returns a BOOL.
setBackgroundColor backgroundColor	The background color for the image; <code>clearColor</code> is the default and is usually correct—you want the underlying view's visualization to appear, rather than a white/black rectangle. Set: <code>aColor</code> (NSColor*). Get: returns an NSColor*.
setFlipped isFlipped	Accessors for whether or not the image is vertically flipped (horizontal flipping is not supported). Set: <code>flag</code> (BOOL). Get: returns a BOOL.
drawRepresentation	Draws <code>imageRep</code> (NSImageRep*) in the current window at <code>rect</code> (NSRect). Returns a BOOL indicating success.

TABLE 10-10. NSImage Methods (Continued)

<code>recache</code>	Frees all off-screen caches; the next composite operation will redraw the representation. Use this for efficiency.
<code>setDelegate</code> <code>delegate</code>	Accessors for the image's delegate (if any).
	Set: <code>anObject (id)</code> . Get: returns an <code>id</code> .
<code>TIFFRepresentation</code>	These methods create and return <code>NSData*</code> objects with TIFF for all representations; the second takes the arguments <code>comp</code> (<code>NSTIFFCompression</code>) and <code>aFloat</code> (<code>float</code>). You can use these methods together with <code>initWithReferencingFile</code> or <code>initWithContentsOfFile</code> .
<code>+canInitWithPasteboard</code>	Returns a <code>BOOL</code> indicating whether the data on <code>pasteboard</code> (<code>NSPasteboard*</code>) can be used to create an instance of <code>NSImage</code> .
<code>+imageFileTypes</code> <code>+imageUnfilteredFileTypes</code>	Returns an <code>NSArray*</code> of file types for which representations can be created; the first array includes file types that can be created either directly or by using filtering services.
<code>+imagePasteboardTypes</code> <code>+imageUnfiltered PasteboardTypes</code>	Returns an <code>NSArray*</code> of pasteboard types for which representations can be created; the first array includes file types that can be created either directly or by using filtering services.

NSImageRep

The image representation is what actually does the work for the image object. NSImageRep methods are shown in Table 10-11.

NSImageReps are created from raw images in files, on the pasteboard, and data objects in memory. The NSImageRep class maintains a registry of the image types that it can handle and the subclasses of NSImageRep that should be created for each one. Filtering services that may be present on a given system can expand the range of image types that are handled; NSImageRep provides methods for including or excluding filtering service conversions.

If you are not adding a new type of image representation, you can probably ignore this section: NSImage manages the process for you. The most probable case where you may need to be involved is one in which your application relies on specific filtering services.

TABLE 10-11. NSImageRep Methods

<code>+imageFileTypes</code> <code>+imageUnfilteredFileTypes</code>	Return NSArray* objects with lists of all file types supported by NSImageRep objects and file types supported by NSImageRep objects without using filtering services. (The former is usually a superset of the latter.)
<code>+imageRepClassForFileType:</code>	Returns the Class for the file type (NSString*).
<code>+imageRepWithContentsOfFile:</code>	Returns a newly created NSImageRep subclass with the data from filename (NSString*) using the appropriate conversions.

TABLE 10-11. NSImageRep Methods (Continued)

<code>+canInitWithPasteboard</code>	Returns a <code>BOOL</code> if a subclass can use the image in <code>pasteboard</code> (<code>NSPasteboard*</code>).
<code>+imagePasteboardTypes</code> <code>+imageUnfiltered PasteboardTypes</code>	Comparable to the file type methods.
<code>+imageRepClassFor PasteboardType</code>	The last method takes the argument <code>pasteboard</code> (<code>NSPasteboard*</code>) rather than a filename argument.
<code>+imageRepWithPasteboard</code>	
<code>+canInitWithData</code>	Returns a <code>BOOL</code> if a subclass can use the image in <code>data</code> (<code>NSData*</code>).
<code>+imageRepClassForData</code>	Returns the class for data type <code>data</code> (<code>NSData*</code>).
<code>+registerImageRepClass</code>	Adds <code>imageRepClass</code> (<code>Class</code>) to the list of registered classes.
<code>+registeredImageRepClasses</code>	Returns an <code>NSArray*</code> of all registered classes.
<code>+unregisterImageRepClass</code>	Removes <code>imageRepClass</code> (<code>Class</code>) from the list of registered classes.
<code>setSize size</code>	Set: <code>aSize</code> (<code>NSSize</code>). Get: returns an <code>NSSize</code> .
<code>setAlpha</code>	Sets the image rep's alpha flag to <code>flag</code> (<code>BOOL</code>).
<code>setBitsPerSample bitsPerSample</code>	Bits per sample (bits per pixel/pixel depth). Set: <code>anInt</code> (<code>int</code>). Get: returns an <code>int</code> .
<code>setColorSpaceName colorSpacename</code>	Set: <code>string</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code> .

TABLE 10-11. NSImageRep Methods (Continued)

<code>setOpaque</code>	Set: flag (BOOL).
<code>isOpaque</code>	Get: returns a BOOL.
<code>setPixelsHigh</code>	Set: anInt (int).
<code>setPixelsWide</code>	Get: returns an int.
<code>pixelsHigh</code>	
<code>pixelsWide</code>	
draw	Attempts to draw the image at (0,0) and returns a BOOL indicating success.
<code>drawAtPoint</code>	Attempts to draw the image at aPoint (NSPoint) and returns a BOOL indicating success.
<code>drawInRect</code>	Attempts to draw the image within rect (NSRect) and returns a BOOL indicating success.

Programming Tools

The heart of any graphical user interface framework consists of its window and view system. One difference between this framework and others is that windows are totally different objects from views (in frameworks like MacApp, the window is a descendant of the view class). Windows have a special relationship with the system's Window Server and explicitly are given more complex responsibilities than in other frameworks.

NSView

Views in the framework are not terribly different from views in other frameworks. They manage:

- view creation

- view hierarchies—subviews and superviews
- geometry—size and shape
- notification
- display
- drawing
- events
- menus, focus, and scrolling
- paper handling

There are a few differences between this architecture and others that you may be used to. Two stand out in particular:

1. As noted previously, coordinates are expressed vertically on an ascending scale from bottom to top, with the (0,0) point—the point of reference—being the lower left corner of a view. This contrasts with Mac OS, where the vertical scale is on an ascending scale from top to bottom, with the (0,0) point being the top left corner of a view.
2. There is explicit support for finding an opaque background view. In the old days, when all view backgrounds were white, this didn't matter; now, it is important to be able to find quickly whatever superview's image serves as the background for a particular view.

View Fabrication Table 10-12 lists the single view fabrication method. You are most likely to create views and their hierarchies in Interface Builder rather than creating them dynamically.

View Hierarchies Hierarchies of views function as they do in any framework—they let you create groups of views that may contain others. Note that the run-time view hierarchy is not

TABLE 10-12. NSView Fabrication Method

<code>initWithFrame</code>	Initializes a view to <code>frameRect</code> (NSRect).
----------------------------	--

the inheritance hierarchy of object-oriented programming. A superview is not a superclass.

The hierarchy of views is logical and not related to their physical locations. In other words, a view that contains another view simply contains it; its location is irrelevant to the containment. In fact, the list of contained views may be ordered in a way that is totally different from their visual representations.

Each view may have one or more subviews that are contained within it (both physically on the imaging device and logically within the object). A view may also have a superview in which it is contained. The topmost view in a window hierarchy is the content view. The content view has no superview; it is contained within its window, and the window has a specific reference to its content view. From the content view's subviews (and their subviews) you can reach all of the views within a window.

Another way to identify individual views is to use their tag field—an integer which by default is -1, but which you can set to any value you want.

Table 10-13 lists the NSView hierarchy methods. Note that many of these methods may return nil: views may not have supervIEWS, they may not have common ancestors, etc.

TABLE 10-13. NSView Hierarchy Methods

<code>superview</code>	Returns the superview as an <code>NSView*</code> .
<code>subviews</code>	Returns the subviews and an <code>NSArray*</code> .
<code>window</code>	Returns the view's <code>NSWindow*</code> .
<code>addSubview</code>	<p>Adds <code>aView</code> (<code>NSView*</code>) to the view's subview list in front of the others.</p> <p>The optional second and third arguments (which must be used together) let you specify the new subview's place (<code>NSWindowOrderingMode</code>) relative to <code>otherView</code> (<code>NSView*</code>). The values for <code>NSWindowOrderMode</code> are <code>NSWindowAbove</code> and <code>NSWindowBelow</code>—don't worry, you can use them for ordering views as well as windows.</p>
<code>removeFromSuperview</code>	Removes the view from the superview. If you are temporarily removing it in order to replace it in another view hierarchy, you should retain the view before removing it; this method releases the view, and therefore it may no longer exist after it has been removed.
<code>replaceSubview</code>	Replaces <code>oldView</code> (<code>NSView*</code>) with <code>newView</code> (<code>NSView*</code>) in the subview list. As noted above, <code>oldView</code> may be destroyed after removal unless you explicitly retain it.

TABLE 10-13. NSView Hierarchy Methods (Continued)

<code>ancestorSharedWithView</code>	Returns the <code>NSView*</code> (if any) that is a superview of this view and <code>aView</code> (<code>NSView*</code>); if one view is a superview of the other, the result could be one of <code>aView</code> or the view that receives this message.
<code>isDescendantOf</code>	Returns a <code>BOOL</code> indicating if this view is a subview (however indirectly) of <code>aView</code> (<code>NSView*</code>). Ancestry/decendancy are used in the hierarchical sense, not in the class-inheritance sense.
<code>opaqueAncestor</code>	Returns the <code>NSView*</code> which is the first view (possibly this one) that completely fills the area of this view. Drawing the opaque ancestor and then this view will leave no holes in the resulting image.
<code>sortSubviewsUsingFunction</code>	Sorts the view's subviews; syntax and use are identical to those of "sortedArrayUsingFunction" on page 175.
<code>viewWillMoveToSuperview</code> <code>viewWillMoveToWindow</code>	These methods are called just before the view is added to <code>newSuperview</code> (<code>NSView*</code>) or to <code>newWindow</code> (<code>NSWindow*</code>). You override them to perform special processing.
<code>viewWithTag</code>	Returns the <code>NSView*</code> with <code>aTag</code> (<code>int</code>).
<code>tag</code>	Returns the view's tag as an <code>int</code> .

View Geometry The basic view geometry methods are called as needed—and they're needed relatively infrequently. You normally size and shape views in Interface Builder. Although

they are often resized during program execution, this is commonly in response to user actions, and you don't get involved.

As in most frameworks, each view has its own coordinate system; in the framework, the (0,0) point is typically the lower left corner of the view. You can flip the view's coordinate system so that the vertical axis increases from top to bottom and the (0,0) point is at the top left. Each view has conversion routines that let it convert points and rectangles from other views' coordinate systems to its own.

One difference from some other frameworks is that views can be rotated: they need not always appear with their "vertical" dimension pointing up.

Two rectangles are important for views:

1. A view's frame identifies its location in the coordinates of its superview. The frame determines the view's size and shape.
2. A view's bounds are its size in its own coordinates. The view's bounds may lie beyond its frame, in which case only the portion of its bounds within the frame is visible.

The geometry methods of views are listed in Table 10-14.

TABLE 10-14. *NSView* Geometry Methods

<code>setFrame</code>	Accessors for the view's frame and bounds.
<code>setBounds</code>	
<code>frame</code>	
<code>bounds</code>	
	Set: <code>frameRect/boundsRect</code> (<code>NSRect</code>).
	Get: return <code>NSRects</code> .

TABLE 10-14. NSView Geometry Methods (Continued)

<code>setFrameOrigin</code>	Sets the view's frame origin to <code>newOrigin</code> (NSPoint), thus effectively moving it within its super-view.
<code>setFrameSize</code>	Sets the view's frame size to <code>newSize</code> (NSSize).
<code>setFrameRotation</code> <code>frameRotation</code>	Accessors for the frame's rotation. Set: <code>angle</code> (float). Get: returns a float.
<code>setBoundsOrigin</code>	Sets the bounds origin to <code>newOrigin</code> (NSPoint), which is located at the frame's (0,0) point. Changing your view's bounds origin may make drawing simpler (a single change to the coordinate system can replace multiple manipulations of coordinates as you draw).
<code>translateOriginToPoint</code>	Adjusts the origin by <code>newOrigin</code> (NSPoint)—a relative movement, compared to the absolute movement of <code>setBoundsOrigin</code> .
<code>setBoundsRotation</code> <code>rotateByAngle</code> <code>boundsRotation</code>	Accessors for the bounds rotation. Note that if the rotations of the bounds and frame are not identical, there will be some areas of the frame that are not covered by bounds—and some areas of bounds that are outside the frame. Set: <code>angle</code> (float). Get: returns a float.

TABLE 10-14. NSView Geometry Methods (Continued)

<code>setBoundsSize</code>	Sets the bounds the <code>newSize</code> (NSSize*). If the size of the bounds is not equal to the size of the frame (the most common case), the bounds rectangle is scaled to the frame. Setting your bounds size to a size double that of the frame will cause it to be scaled 50%; setting it to a size half that of the frame will cause it to be scaled 200%.
<code>scaleUnitSquareToSize</code>	You can explicitly scale the view to <code>newUnitSize</code> (NSSize).
<code>isFlipped</code>	Returns a BOOL indicating if the view's coordinate system has been flipped.
<code>isRotatedFromBase</code> <code>isRotatedOrScaledFromBase</code>	Returns a BOOL indicating if the view or any of its superviews has ever been rotated or scaled (regardless of current state). Use for optimization of drawing in high-performance tasks.
<code>centerScanRect</code>	Centers <code>aRect</code> (NSRect); if you have been scaling the coordinate system, PostScript may have overscanned. If you need to worry about this, you'll understand it.
<code>convertPointToView</code> <code>convertPointFromView</code>	These routines take (in sequence) <code>aPoint</code> (NSPoint), <code>aSize</code> (NSSize), and <code>aRect</code> (NSRect). Their second argument is <code>aView</code> (NSView*). They convert the geometry between the coordinate system of <code>aView</code> and their own. They return the same type as their first argument.
<code>convertSizeToView</code> <code>convertSizeFromView</code>	
<code>convertRectToView</code> <code>convertRectFromView</code>	

Note that these methods do not force a redisplay of the view; call `display` or `setNeedsDisplay`(see Table 10-16) to do that.

Notification One of the important roles of views is handling events and directly or indirectly notifying others of these events.

The accessors shown in Table 10-15 let you control whether changes to the view’s geometry are passed on as notifications. You may want to limit a view’s notifications to content-related events; particularly in the case of views that are frequently resized, rotated, scaled, or relocated, this can provide significant performance improvements.

TABLE 10-15. `NSView` Notification Methods

<code>setPostsFrameChanged</code>	Set: flag (BOOL).
Notifications	Get: returns a BOOL.
<code>setPostsBoundsChanged</code>	
Notifications	
<code>postsFrameChanged</code>	
Notifications	
<code>postsBoundsChanged</code>	
Notifications	

Sizing, Focusing, and Display These are the methods that you use to manage sizing, focusing, and display. Sizing in this context is the relative resizing behavior that a view is expected to display when it is actually resized by a user. (That is to say, you use the routines in the geometry section to actually size a view; these methods deal with the consequences.)

The behavior of views when their superview is resized is not a simple issue. Both size and location come into play. For example:

- When a window is resized, buttons within it are normally not resized: buttons are often of a standard size.
- When a window is resized, buttons that are located in specific locations are often moved relative to the edge of the view (that is, the OK button is usually a fixed distance in from the lower right corner of the view).
- The frame of a view within a resized window is often resized directly with the window: making the window an inch wider normally increases the size of its content by one inch.
- The information within a frame (text or graphics—the contents of an embedded view's bounds rectangle) is normally unaffected, although the change in the embedded view's frame may make more or less of the content visible.

The framework provides two sets of three constants that you can OR together to control the resizing behavior of a view when its superview is resized. For width, the three constants are:

1. `NSViewMinXMargin` allows the left (min) horizontal margin to vary.
2. `NSViewMaxXMargin` allows the right (max) horizontal margin to vary.
3. `NSViewWidthSizable` allows the view itself to be resized.

Three similar constants are provided for the YMargin and the height.

As in other frameworks, a view must be focused before it is drawn in. There are utility routines for locking and unlocking a view's focus.

Finally, you will find the critical display routines here: these are the methods that tell a view to display itself. What it actually displays is discussed later in the section on drawing, specifically in the `drawRect` method. You set a view (or a rectangle within it) as needing display; the next time it has an opportunity (usually during the event loop) the framework will redisplay the view. If you call a display method, redrawing is immediate.

Table 10-16 lists these methods. You may never call or override them—managing views and their redrawing is what the framework does.

TABLE 10-16. *NSView* Size, Focus, and Display Methods

<code>setAutoresizesSubviews</code> <code>autoresizesSubviews</code>	Accessors to control whether subviews are resized (in whatever way) when this view is resized. Normal behavior is YES. Set: <code>flag</code> (BOOL). Get: returns a BOOL.
<code>setAutoresizingMask</code> <code>autoreresizingMask</code>	Accessors for the mask controlling resizing using the two sets of three constants described above. Set: <code>mask</code> (unsigned int). Get: returns an unsigned int.
<code>resizeWithOldSuperviewSize</code> <code>resizeSubviewsWithOldSize</code>	These methods are called to resize this view or its subviews from <code>oldFrameSize</code> (NSSize). The frame size argument in the first method is for the superview; in the second, it is the former size of this view. In both cases, the new size has been set in the appropriate view before calling the method.

TABLE 10-16. NSView Size, Focus, and Display Methods (Continued)

<code>†focusView</code>	Returns the <code>NSView*</code> that currently has the focus. Note that this is a class method: only one view in Rhapsody can have focus at any moment; the view that this method returns (if any) may have nothing to do with your view's hierarchy or window.
<code>lockFocus</code> <code>unlockFocus</code>	Locks and then unlocks the focus for this view; all PostScript commands take place in this view until unlocked. If you call the standard display methods, these methods are called for you.
<code>setNeedsDisplay</code> <code>needsDisplay</code>	Marks the bounds of the view (and hence all its subviews) for display (updating) at the next opportunity—either the next pass through the event loop or the next display command. You call this routine as needed; the framework does the drawing as necessary. Set: <code>flag</code> (BOOL). Get: returns a BOOL.
<code>setNeedsDisplayInRect</code>	Marks <code>invalidRect</code> (<code>NSRect</code>) within the view's bounds as needing to be displayed. This method marks the rectangle only as needing display: to mark a rectangle within a view as not needing display, you must call <code>needsDisplay(NO)</code> for the view—provided no other rects have been marked.

TABLE 10-16. NSView Size, Focus, and Display Methods (Continued)

<code>display</code> <code>displayIfNeeded</code> <code>displayIfNeededIgnoringOpacity</code>	Displays the view (and hence its subviews) unconditionally, or only if any areas are marked as needing display. If there are “holes” in the view, the first opaque ancestor is redrawn to provide a background, and all views between it and this view are drawn. The last method ignores this processing and displays (if needed) this view without regard to its background.
<code>displayRect</code> <code>displayIfNeededInRect</code> <code>displayIfNeededInRectIgnoringOpacity</code>	Comparable to the previous three methods, each of these takes a <code>rect</code> (<code>NSRect</code>) argument; their processing and decision making are confined to that rectangle.
<code>displayRectIgnoringOpacity</code>	Same as <code>displayIfNeededInRectIgnoringOpacity</code> except that the display is unconditional.
<code>isOpaque</code>	Returns a <code>BOOL</code> indicating if it covers its extent totally with its drawing.

Drawing The final drawing methods for views contain PostScript calls (rather than calls to toolbox routines like `QuickDraw`). This may be daunting on a conceptual level; however, when you actually look at the PostScript calls that you usually make, you will find that they are trivial—and comparable to the routines that you are used to.

The heart of these routines is `drawRect`—which is normally called from a `display` method. This is one routine that you do have to override: if you don’t, your view isn’t drawn. (Of

course, descendants of views like buttons, rulers, and scrollers draw themselves and you don't have to override their `drawRect` methods.)

The drawing methods of `NSView` are shown in Table 10-17.

TABLE 10-17. `NSView` Drawing Methods

<code>drawRect</code>	<p>Override this method to draw that part of your view in <code>aRect</code> (<code>NSRect</code>). This method contains PostScript calls (see the sample applications that ship with Rhapsody for examples—you can probably figure it out for yourself without ever looking at the syntax).</p> <p>If you must control drawing, you call a display method that does the preparation and cleanup—not this method.</p>
<code>visibleRect</code>	<p>Returns an <code>NSRect</code> that is the part of this view not obscured by its superviews. Other views, other windows, and window placement off the screen do not affect this value. Thus, the fact that a view has a visible rectangle doesn't mean that drawing in it will be visible.</p>
<code>canDraw</code>	<p>This method can be used to optimize performance; it returns a <code>BOOL</code> indicating whether or not the view can be drawn. You can then decide whether or not to lock focus on the view. The display methods do this for you.</p>

TABLE 10-17. NSView Drawing Methods (Continued)`shouldDrawColor`

Another optimizing routine, this one returns a BOOL indicating if drawing can use color (gray-scale is not a color). You may choose to change your drawing code, but more likely the default PostScript behavior will be appropriate. A more appropriate use might be to refuse to even attempt to draw certain types of color images without color.

```
allocateGState
setUpGState
renewGState
releaseGState
gState
```

These routines let you use your own PostScript graphics status object, which you might do for optimization (it is a trade-off of memory—using your own—vs. performance—the cost of resetting parameters with each view focus).

Events and Mouse Handling After displaying themselves, the next most important task of views is to manage events such as mouse clicks. Views also manage the cursor when it is over their area. You may set any number of cursor rectangles within your view, with a cursor for each one (or a single cursor for all of them). This lets you easily change the cursor as it travels across your view's data. Of course, if the data itself is displayed in subviews, those subviews can declare cursor rectangles for their entire areas so that your view doesn't have to worry about the locations of its subviews.

The event and mouse handling methods of NSView are shown in Table 10-18.

TABLE 10-18. NSView Event and Mouse Handling Methods

<code>acceptsFirstMouse</code>	Returns a BOOL indicating if the view will act on an initial mouse click from <code>theEvent</code> (NSEvent*). Note that <code>theEvent</code> is not a proto-type: it is the specific event that the view is being asked about. Normally, the first mouse click in an inactive window activates it; however, you may decide that a mouse click in a scroll bar or control should be accepted.
<code>hitTest</code>	Returns the frontmost NSView* in which <code>aPoint</code> (NSPoint) is located. The point is in the superview's coordinate system; this is the method that NSWindow uses to identify the view that received a mouse click.
<code>mouse</code>	Returns a BOOL indicating if <code>aPoint</code> (NSPoint) is in <code>aRect</code> (NSRect) within the view.
<code>performKeyEquivalent</code>	Returns a BOOL indicating if the view responded to <code>theEvent</code> (NSEvent*) using its shortcut. If you don't process the event, call <code>super's</code> method before returning NO.
<code>performMnemonic</code>	Returns a BOOL indicating if the view responded to <code>aString</code> (NSString*) which is a mnemonic. If you don't process the event, call <code>super's</code> method before returning NO.

**TABLE 10-18. NSView Event and Mouse Handling
Methods (Continued)**

<code>dragImage</code>	This is the dragging feedback method. Its arguments are <code>anImage</code> (NSImage*)—the tracking image; <code>imageLoc</code> (NSPoint)—the location of the image to start with; <code>mouseOffset</code> (NSPoint)—the offset from the mouse down event where dragging starts which often is a zero point; <code>theEvent</code> (NSEvent*)—the event that started this; <code>pboard</code> (NSPasteboard*)—the pasteboard that contains whatever is actually being dragged; <code>sourceObject</code> (id)—the controller: typically this view or its window; and <code>flag</code> (BOOL)—which indicates if an unsuccessful drag should slide back the image.
<code>dragFile</code>	This is the method you call for the special case of dragging a file icon. Its arguments are <code>fullPath</code> (NSString*)—the file's path, <code>aRect</code> (NSRect)—the icon's position within the view, <code>flag</code> (BOOL)—whether to end an unsuccessful drag with sliding back the icon, and <code>theEvent</code> (NSEvent*)—the instigator of this process. Returns a BOOL indicating success.
<code>registerForDraggedTypes</code>	Registers the view as a destination for the types in <code>pboardTypes</code> (NSArray*).
<code>unregisterDraggedTypes</code>	Unregisters the view for all destination dragging.

TABLE 10-18. NSView Event and Mouse Handling Methods (Continued)

<code>shouldDelayWindowOrderin ForEvent</code>	Returns a BOOL indicating if the window should not be reordered in response to <code>theEvent</code> (NSEvent*). Typically this is the case with the start of a drag which you don't want to activate a source window.
<code>addCursorRect</code>	Adds <code>aRect</code> (NSRect) with <code>aCursor</code> (NSCursor*) to the cursor rectangles used by this view.
<code>removeCursorRect</code>	Removes <code>aRect</code> (NSRect) for <code>aCursor</code> (NSCursor*). Note that <code>resetCursorRects</code> (next) may reestablish it.
<code>discardCursorRects resetCursorRects</code>	Discards (forever) or resets (in response to a resize, etc.) all cursor rectangles in a view.

Menus and Scrolling The methods in Table 10-19 manage menus and scrolling. Scrolling is actually accomplished by the interaction of an NSScrollView, NSScrollers (possible two—one horizontal and one vertical), and an NSClipView in which the topmost scrolling view is contained. These are the objects that actually scroll this view.

TABLE 10-19. NSView Menus and Scrolling Methods

<code>scrollPoint</code>	Scrolls the view so that <code>aPoint</code> (NSPoint) is at (0,0).
--------------------------	---

TABLE 10-19. NSView Menus and Scrolling Methods

<code>scrollRectToVisible</code>	Returns a BOOL indicating if it was possible to scroll so that <code>aRect</code> (NSRect) is visible.
<code>autoscroll</code>	Returns a BOOL indicating if the view has been scrolled by the amount of the distance of <code>theEvent</code> (NSEvent*) outside the view. <code>theEvent</code> should be in window coordinates. This method is useful when dragging slightly beyond a view (the Finder implements a similar behavior when you drag an icon to the top or bottom of a scrolling window).
<code>scrollRect</code>	This rarely used method scrolls <code>aRect</code> (NSRect) by <code>offset</code> (NS-Size). Use <code>scrollPoint</code> , <code>scrollRectToVisible</code> , or <code>autoScroll</code> instead.
<code>adjustScroll</code>	You can override this method so that scrolling doesn't leave inappropriate edges of data within your view. The argument <code>proposedVisibleRect</code> (NSRect) can be modified as you see fit and returns as an NSRect to accommodate the data. The default implementation simply returns <code>proposedVisibleRect</code> .
<code>enclosingScrollView</code>	Returns the first NSScrollView* that is an ancestor of this view.

TABLE 10-19. NSView Menus and Scrolling Methods

<code>setNextKeyView</code>	Puts this view into the window's key view loop after a view (NSView*).
<code>nextKeyView</code>	The other routines return the next and previous key view (or nil) to this one as an NSView*.
<code>previousKeyView</code>	
<code>nextValidKeyView</code>	These routines return NSView* objects that are the appropriate key views and that have first responder status.
<code>previousValidKeyView</code>	

Paper Routines Finally, Table 10-20 lists the paper handling and printing routines for windows. Many of these routines are PostScript calls; they are documented in the PostScript documentation—and you probably won't have to deal with them at all.

TABLE 10-20. NSView Paper Handling Methods

<code>print</code>	Opens the print or fax panel and, if necessary, prints or faxes. The sole argument, <code>sender (id)</code> , is the sender of this action. (A button, a menu command, etc.)
<code>fax</code>	
	You don't call this method often: you connect it in Interface Builder to the appropriate interface element.

TABLE 10-20. NSView Paper Handling Methods (Continued)

<code>dataWithEPSInsideRect</code>	Returns an <code>NSData*</code> object that is this view within <code>aRect</code> (<code>NSRect</code>). Use it for data transfer to the pasteboard or other destinations that need a digital image of the view (rather than its contents).
<code>writeEPSInsideRect</code>	Instead of creating an <code>NSData*</code> object, this method writes the part of the view within <code>aRect</code> (<code>NSRect</code>) to <code>pboard</code> (<code>NSPasteboard*</code>).
<code>heightAdjustLimit</code> <code>widthAdjustLimit</code>	Returns the portion (0-1) of the page's image that can be moved to the next page when paginating. Returns a float. The smaller the number, the less flexibility the system has in pagination.
<i><code>adjustPageWidthNew</code></i>	During pagination, this method is called (and you can override it so that bad breaks are avoided). The arguments are <code>newRight</code> (<code>float*</code>)—which you should return based on <code>left</code> (<code>float</code>), <code>proposedRight</code> (<code>float</code>), and <code>rightLimit</code> (<code>float</code>). <code>newRight</code> may be any value between <code>left</code> and <code>rightLimit</code> that makes the content look good.
<i><code>adjustPageHeightNew</code></i>	Similar to <code>adjustPageWidthNew</code> . This method's arguments are <code>newBottom</code> (<code>float*</code>), <code>top</code> (<code>float</code>), <code>proposedBottom</code> (<code>float</code>), and <code>bottomLimit</code> (<code>float</code>).

TABLE 10-20. NSView Paper Handling Methods (Continued)

<code>knowsPagesFirst</code>	Returns a <code>BOOL</code> indicating if this view wants to do its own pagination from <code>firstPageNum</code> (<code>int *</code>) to <code>lastPageNum</code> (<code>int *</code>). These values may be modified if <code>YES</code> is returned and the view's pagination provides a different range of actual pages than the system's estimate.
<code>rectForPage</code>	Returns the <code>NSRect</code> for <code>pageNumber</code> (<code>int</code>). This is the pagination routine that you use if <code>knowsPagesFirst</code> returns <code>YES</code> .
<code>locationOfPrintRect</code>	Returns the <code>NSPoint</code> that is the page location of <code>aRect</code> (<code>NSRect</code>). Unless centered, this is usually the upper left corner of the page.
<code>drawPageBorderWithSize</code> <code>drawSheetBorderWithSize</code>	These routines allow you to draw registration marks, borders, etc. on pages and sheets. You override them if you are using the system's pagination and you want to provide additional marks on the output.
<code>beginPrologueBBox</code> <code>forWhom</code> <code>endHeaderComments</code> <code>endPrologue</code> <code>beginSetup</code> <code>endSetup</code> <code>beginPage</code> <code>beginPageSetupRect</code> <code>addToPageSetup</code> <code>endPageSetup</code> <code>endPage</code> <code>beginTrailer</code> <code>endTrailer</code>	These methods write PostScript code to the output.

NSScreen

The NSScreen object represents the screen (or screens) currently in use. This is an example of the pervasiveness of objects in Rhapsody: in most other frameworks, the screen device is a data structure and the framework must mediate between the flat world of the data structure and that of its objects. In Rhapsody, the screen itself becomes an object and is part of the framework.

The methods for NSScreen are shown in Table 10-21.

TABLE 10-21. NSScreen Methods

<code>+mainScreen</code>	Returns the appropriate NSScreen* object for the computer at this time.
<code>+deepestScreen</code>	
<code>+screens</code>	Returns an NSArray* of the screens currently in use.
<code>depth</code>	Returns an NSWindowDepth.
<code>frame</code>	Returns an NSRect.
<code>supportedWindowDepths</code>	Returns an array of NSWindowDepth structs representing all of the screen's possible window depths.
<code>deviceDescription</code>	Returns the NSDictionary* for the screen.

NSWindow

Very much like views, windows are critical and pervasive objects in the framework—but they are overridden even less often than view objects. Windows, after all, not only display their views' data but also are responsible for interacting with the Window Server and managing events and menus. Changing their appearance or behavior is often a mistake.

Occasionally, it is appropriate to add functionality to windows (rather than changing default behavior). You can do this easily by adding a delegate to the window—and this is common, easy to implement, and quite appropriate.

Window Creation You create windows even less often than you create views. Interface Builder is the appropriate programming tool for creating windows. However, should you need to do so, Table 10-22 lists the `NSWindow` fabrication method.

The methods use a `styleMask`, which is an unsigned int composed of constants that you can OR together. The possible values are:

- `NSTitledWindowMask`—adds a title bar to the window.
- `NSClosableWindowMask`—adds a close box.
- `NSMiniaturizableWindowMask`—adds a miniaturize button.
- `NSResizableWindowMask`—makes the window resizable.

Most document windows have a style mask of `NSTitledWindowMask OR NSClosableWindowMask OR NSMiniaturizableWindowMask OR NSResizableWindowMask`.

The `NSBackingStoreType` can have the values of `NSBackingStoreRetained`, `NSBackingStoreNonretained` and `NSBackingStoreBuffered`. It determines the buffering used with the actual device.

The `flag` argument lets you delay creation of the device until the window is actually displayed on screen. If you have a lot of windows created at application start up but don't display them until they are needed, this can provide performance improvements.

TABLE 10-22. NSWindow Fabrication Methods

<code>initWithContentRect</code>	Initializes a window with <code>contentRect</code> (NSRect), <code>styleMask</code> (unsigned int), <code>backingType</code> (NSBackingStoreType), and <code>flag</code> (BOOL).
----------------------------------	--

Geometry The roster of routines in Table 10-23 is used to manage window geometry. Again, as with views, these routines are constantly used—but usually not by you. The framework manages window geometry in response to user actions; you rarely call the methods and if you override them you risk disrupting the user experience.

TABLE 10-23. NSWindow Geometry Methods

<code>contentRectForFrameRect</code>	Returns the NSRect that is the <code>contentRect</code> for the window with <code>frameRect</code> (NSRect) and a style mask of <code>aStyle</code> (unsigned int). The style mask determines title bars, resize boxes, etc. and therefore influences how much of the frame is left over for the content.
<code>frameRectForContentRect</code>	The opposite of the previous method, this applies to <code>contentRect</code> (NSRect) the style mask in <code>aStyle</code> (unsigned int) and returns an NSRect for the frame rectangle.

TABLE 10-23. NSWindow Geometry Methods (Continued)

<code>+minFrameWidthWithTitle</code>	Given <code>aTitle</code> (<code>NSString*</code>) and <code>aStyle</code> (<code>unsigned int</code>)—a style mask, this method returns the minimum width as a float for the window frame so that everything is displayed.
<code>convertBaseToScreen</code> <code>convertScreenToBase</code>	Converts <code>aPoint</code> (<code>NSPoint</code>) between the window and screen coordinate systems; returns an <code>NSPoint</code> in the other system.
<code>setFrame</code> <code>frame</code>	The window's frame (its size and location) is expressed in screen coordinates (just as view frames are expressed in the coordinates of their superviews). Set: <code>frameRect</code> (<code>NSRect</code>), <code>flag</code> (<code>BOOL</code>)—determines whether the window should be displayed immediately. Get: returns an <code>NSRect</code> .
<code>setFrameOrigin</code>	Moves the window's lower left point to <code>aPoint</code> (<code>NSPoint</code>).
<code>setFrameTopLeftPoint</code>	Moves the window's top left point to <code>aPoint</code> (<code>NSPoint</code>).
<code>setContentSize</code>	Sets the window's content size to <code>aSize</code> (<code>NSSize</code>). Depending on the window's style mask, its size will be somewhat larger than <code>aSize</code> .
<code>center</code>	Centers the window but does not necessarily display it.

TABLE 10-23. NSWindow Geometry Methods (Continued)

<code>cascadeTopLeftFromPoint</code>	Given another window's <code>topLeftPoint</code> (NSPoint), this method returns an NSPoint that will result in a pleasant cascade (staggering) of windows if you place this window at the new point.
<code>resizeFlags</code>	Returns an int containing the flags field of the mouse-down event that is being processed. Used internally.
<code>setMinSize</code> <code>setMaxSize</code>	Accessors for the window's maximum and minimum sizes.
<code>minSize</code> <code>maxSize</code>	Set: <code>aSize</code> (NSSize). Get: returns an NSSize.
<code>setAspectRatio</code> <code>aspectRatio</code>	Accessors for the window's (optional) aspect ratio; if used, the window can only be resized to multiples of this size. Set: <code>ratio</code> (NSSize). Get: returns an NSSize.
<code>setResizeIncrements</code> <code>resizeIncrements</code>	Accessors for the field that lets you specify the size by which the window can be resized in both directions. Set: <code>increments</code> (NSSize). Get: returns an NSSize.

TABLE 10-23. NSWindow Geometry Methods (Continued)

<code>constrainFrameRect</code>	This method changes the window's <code>frameRect</code> (<code>NSRect</code>) if necessary so that at least its top lies on <code>aScreen</code> (<code>NSScreen*</code>). It will also attempt to bring the bottom onto the screen if the window is resizable. No horizontal adjustments are made.
<code>saveFrameUsingName</code> <code>setFrameUsingName</code> <code>removeFrameUsingName</code>	<p>You can save the window's frame rectangle in user defaults with <code>name</code> (<code>NSString*</code>), which is concatenated to the end of "NSWindow Frame" using the first method; the second method retrieves that default setting also using <code>name</code> (<code>NSString*</code>) and returns a <code>BOOL</code> indicating if it could successfully find it and set the window's frame rectangle.</p> <p>You remove the default information using the third method, which also takes a single <code>name</code> (<code>NSString*</code>) argument.</p>
<code>setFrameAutoSaveName</code> <code>frameAutosaveName</code>	<p>Accessors for a name to be used to save the window's frame rectangle automatically in user defaults whenever it is resized. (If the name is not set, you can manually save the frame rectangle using the preceding methods.)</p> <p>Set: <code>name</code> (<code>NSString*</code>); returns a <code>BOOL</code> indicating that the name has been accepted by the user defaults system.</p> <p>Get: returns an <code>NSString*</code>.</p>

TABLE 10-23. NSWindow Geometry Methods (Continued)

<code>stringWithSavedFrames</code>	You can use a string to set a window's frame. These are the accessors for that string. The first returns an <code>NSString*</code> ; the second sets the window's frame using the sole argument named <code>aString</code> (<code>NSString*</code>).
<code>setFrameFromString</code>	

Window Management The methods in this section let you manage windows with regard to their placement from front to back on the screen as well as with regard to their position in the interface. Each application normally has a main window, which is the currently active window; in addition, a key window may be placed in front of the main window. Often the key window is a panel or palette that temporarily provides a special-purpose interface.

The screen of course is a two-dimensional object; windows appear to be placed in order from front to back because they obscure parts of one another. Two levels within this imaginary three-dimensional world are defined:

1. `NSNormalWindowLevel` is where most windows live. You can normally see windows from several applications in this level; clicking on a window from any application brings that window forward and activates the application. (An option on Rhapsody lets users control whether bringing the application forward brings all of its windows to the front of this layer rather than just the clicked window.)
2. `NSFloatingWindowLevel` is a level that appears to be in front of all other windows; all palettes and panels appear to exist in this level. Typically, only one application's windows appear in the floating window level:

inactive applications generally hide their palettes and panels.

Within each level, you can position windows relatively using the `NSWindowOrderingMode` constants: `NSWindowAbove`, `NSWindowBelow`, and `NSWindowOut` (which simply removes the window from the level).

When windows are ordered within their level, reordering them does not affect their status (that is, if they were key or main windows they remain so; if they weren't they don't become so).

The concept of window levels has evolved over time and is continuing to evolve. The OpenStep interface on some platforms defined three additional levels—a dock for commonly used applications and levels for main and sub-menus.

All of this is managed using the methods of Table 10-24.

TABLE 10-24. `NSWindow` Window Management Methods

<code>orderBack</code> <code>orderFront</code>	These methods move the window to the front or back of the window's level.
<code>orderFrontRegardless</code>	This method orders the window to the front of its level, regardless of whether the application is active.
<code>orderOut</code>	Removes the window from the ordering of its level.

**TABLE 10-24. NSWindow Window Management
Methods (Continued)**

<code>windowNumber</code>	Returns the number of the Post-Script window device; you can use this to identify the window in further calls.
<code>orderWindow</code>	Rather than the absolute ordering of the previous methods, you can order a window relative to another by specifying its <code>place</code> (<code>NSWindowOrderMode</code> —see previous description) and <code>otherWindowNumber</code> (int—obtained from the <code>windowNumber</code> method).
<code>setLevel</code> <code>level</code>	Accessors for the window level; use the constants described previously. Set: <code>newLevel</code> (int). Get: returns an int.
<code>isVisible</code>	Returns a <code>BOOL</code> indicating if the window is on the screen; it says nothing about whether or not the window is obscured by other screen objects. In other words, it is quite possible for you not to be able to see a visible window. [This method is dedicated to the memory of Lewis Carroll.]
<code>becomeKeyWindow</code>	Used internally. If you need special processing when your window becomes a key window, register to receive the <code>NSWindowDidBecomeKeyNotification</code> and act on that instead.
<code>canBecomeKeyWindow</code>	Returns a <code>BOOL</code> indicating if it can become a key window.

TABLE 10-24. NSWindow Window Management Methods (Continued)

<code>isKeyWindow</code>	Returns a BOOL indicating if it is a key window.
<code>makeKeyAndOrderFront</code>	This action method takes the single argument <code>sender (id)</code> ; it does what it says. You normally wire this method to an interface element (button, menu command, etc.) in Interface Builder.
<code>makeKeyWindow</code>	Makes this the key window.
<code>resignKeyWindow</code>	Used internally. For specially processing on resignation, register to receive the <code>NSWindowDidResignKeyNotification</code> .
<code>becomeMainWindow</code> <code>canBecomeMainWindow</code> <code>isMainWindow</code> <code>makeMainWindow</code> <code>resignKeyWindow</code>	These methods are comparable to their key window counterparts.

Drawing Drawing routines for windows are concerned with the management of drawing and with some optimizations. One particularly nice feature of the framework is the ability to cache rectangles in the window after they have been drawn.

You do this by calling `cacheImageInRect`, which copies the rectangle of the fully drawn window to an off-screen buffer. You can then draw directly on the screen to highlight the contents or otherwise change things—and then quickly restore the original cached image. This is the architecture that Apple Guide uses for displaying coach marks on Mac OS: the portion of the window where the coach mark will be drawn is cached to a buffer, the coach mark is drawn, and when the mouse is moved or some other user event occurs the off-screen (pre-coach mark) image is restored.

Another useful feature is the ability to flush the window image from an off-screen buffer (if the window device allows this). This brings the efficiencies of off-screen drawing explicitly into the framework. Flushing moves the off-screen image to the screen.

A buffered window normally draws itself into the buffer and moves the image onto the screen as each view is drawn. If you disable flushing, the drawing is done to the buffer, but nothing is moved to the screen until you explicitly flush it to the screen.

The window drawing methods are shown in Table 10-25.

TABLE 10-25. NSWindow Drawing Methods

<code>display</code> <code>displayIfNeeded</code>	If you think of the window as a superview of its content view (it's not, since it's not an <code>NSView</code>), these methods behave very similarly to the <code>NSView</code> <code>display</code> , <code>displayIfNeeded</code> , <code>setNeedsDisplay</code> , and <code>needsDisplay</code> methods.
<code>setViewsNeedDisplay</code> <code>viewsNeedDisplay</code>	
<code>useOptimizedDrawing</code>	Unless you have subviews that overlap one another, you can turn this on using <code>flag</code> (BOOL) for efficiency improvements.
<code>setAutodisplay</code> <code>isAutodisplay</code>	Normally the window draws its views as needed (autodisplay mode). If for some reason you want the views only to be drawn in response to your explicit display commands, turn autodisplay off.
	Set: <code>flag</code> (BOOL). Get: returns a BOOL.

TABLE 10-25. NSWindow Drawing Methods (Continued)

<code>update</code>	Called when the window is updated. You can override this to do pre- or postprocessing before or after calling the superclass's update method.
<code>flushWindow</code> <code>flushWindowIfNeeded</code>	These methods move the buffered image to the screen; the second minimizes flickering.
<code>enableFlushWindow</code> <code>disableFlushWindow</code>	These methods control window flushing; when it is disabled, drawing is done to the buffer (if it exists) and nothing is drawn to the screen until it is explicitly flushed. You should balance disable/enable calls (the normal state is enabled).
<code>isFlushWindowDisabled</code>	Returns a BOOL indicating if flushing is disabled.
<code>cacheImageInRect</code>	Caches the window image in <code>aRect</code> (NSRect). Any number of cached rectangles can be created for a window; all are discarded when the window is next displayed.
<code>restoreCachedImage</code> <code>discardCachedImage</code>	<p>The first method restores all of the window's cached rectangles to its buffer; to move them onto the window, you must then call <code>flushWindowIfNeeded</code>.</p> <p>The second method discards all cached images; it is invoked automatically when the window is re-displayed.</p>

Window Server and Screens You sometimes need to interact with the screen or the device on which the window is displayed. The methods in Table 10-26 let you do this.

There is an `NSDictionary` associated with each window that contains a number of useful values:

- `NSDeviceResolution` is the window's resolution; it is an `NSize`.
- `NSDeviceColorSpaceName` is the window's color space; it is an `NSString*`.
- `NSDeviceBitsPerSample` is the bit depth; it is an `NSNumber`.
- `NSDeviceIsScreen` is a `BOOL`.
- `NSDeviceSize` is an `NSSize`.

You access the dictionary by calling `deviceDescription`.

TABLE 10-26. `NSWindow` Window Server and Screen Methods

<code>gState</code>	Returns an int for the PostScript graphics state used for this window and all of its views (except those that have opted for their own graphics state).
<code>deviceDescription</code>	Returns the <code>NSDictionary*</code> for the device.
<code>setBackingType</code> <code>backingType</code>	Accessors for the window's backing store type. Set: <code>backingType</code> (<code>NSBackingStoreType</code>). Get: returns an <code>NSBackingStoreType</code> .

TABLE 10-26. NSWindow Window Server and Screen Methods (Continued)

<code>setOneShot</code> <code>isOneShot</code>	When true, the PostScript device for the window is freed when the window is no longer displayed. Set: <code>flag</code> (BOOL). Get: returns a BOOL.
<code>defaultDepthLimit</code>	Returns the systemwide default depth limit as an <code>NSWindowDepth</code> .
<code>setDepthLimit</code> <code>depthLimit</code>	You can explicitly set the window's depth limit. Set: <code>limit</code> (<code>NSWindowDepth</code>). Get: returns an <code>NSWindowDepth</code> .
<code>setDynamicDepthLimit</code> <code>hasDynamicDepthLimit</code>	Alternatively, you can let the system manage the depth for you. This is the best way to handle situations in which there may be multiple monitors. Set: <code>flag</code> (BOOL). Get: returns a BOOL.
<code>canStoreColor</code>	Returns a BOOL indicating the color status of the window's depth.
<code>deepestScreen</code>	Returns the deepest <code>NSScreen*</code> on which the window is displayed.
<code>screen</code>	Returns the <code>NSScreen*</code> where most (if not all) of the window is shown.

Events and Key Views The methods in Table 10-27 allow windows to manage events and their key views.

TABLE 10-27. NSWindow Event and Key View Methods

<code>currentEvent</code>	Returns the application's current <code>NSEvent*</code> (not necessarily the most recent event for the window).
<code>postEvent</code> <code>nextEventMatchingMask</code> <code>discardEventsMatchingMask</code>	These methods call the corresponding methods of <code>NSApplication</code> . In addition, the <code>NSWindow nextEventMatchingMask</code> method can be called just with <code>mask</code> (unsigned int); it will use default values for the other arguments when it calls the <code>NSApplication</code> method.
<code>sendEvent</code>	Called by <code>NSApplication</code> 's corresponding method; for internal use.
<code>tryToPerform</code>	This override of the <code>NSResponder</code> method includes the window's delegate as a candidate to perform actions.
<code>keyDown</code>	This is the method that handles appropriate events for <code>theEvent</code> (<code>NSEvent*</code>). You may need to override this method if your window responds to keyboard events (as opposed to its views responding to keyboard events such as typing text).
<code>mouseLocationOutsideOfEventStream</code>	Returns an <code>NSPoint</code> that is the current mouse position—wherever it may be.

**TABLE 10-27. NSWindow Event and Key View Methods
(Continued)**

<code>setAcceptsMouseMovedEvents</code> <code>acceptsMouseMovedEvents</code>	<p>Accessor controlling the window's acceptance of mouse-moved events.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>setInitialFirstResponder</code> <code>initialFirstResponder</code>	<p>Accessors for the window's first responder view.</p> <p>Set: <code>aView</code> (NSView*). Get: returns an NSView*.</p>
<code>selectKeyViewFollowingView</code> <code>selectKeyViewPrecedingView</code>	<p>The argument <code>aView</code> (NSView*) is a view that is asked for its next/previous key view, which is then made the first responder.</p>
<code>selectNextKeyView</code> <code>selectPreviousKeyView</code>	<p>These action methods—argument <code>sender</code> (id)—try to find the appropriate key view and make it the first responder.</p>
<code>keyViewSelectionDirection</code>	<p>Returns an <code>NSSelectionDirection</code>—<code>NSSelectingPrevious</code>, <code>NSSelectingNext</code>, <code>NSDirectSelection</code>—to indicate how the window is changing key views.</p>

Editing Entering text data always poses problems for graphical user interfaces. The display of a stream of text—even a complex stream involving various scripts and directions of writing—is almost trivial next to the problems of displaying a constantly changing stream of text that is being typed, re-typed, backspaced over, selected, cut, pasted, and otherwise mangled by a user.

MacApp uses a floating text editing view that it inserts over text views when you are actually entering data. This floating

text view (which experienced MacApp programmers grew to know and respect) handled the data entry problems reasonably well. When the (underlying) text view was deactivated, the floating text view inserted its text into the real text view and no one was the wiser.

A somewhat similar architecture is used here, but it is a little simpler and much more powerful. It is shown in Table 10-28.

TABLE 10-28. NSWindow Editing Methods

<code>fieldEditor</code>	<p>Returns an <code>NSText*</code> to be used for editing text; <code>createFlag</code> (BOOL) indicates if the window should create one if it doesn't have one; <code>anObject</code> (id) is the object requesting the field editor.</p> <p>The delegate has a method that is called by this method; you can return a customized field editor for certain views within your window (even asking custom view types to create their own editors for you).</p>
<code>endEditingFor</code>	<p>Ends editing for <code>anObject</code> (id) and removes the field editor from use; it can be reused by the window on request for other views that need editing.</p>
<code>setDocumentEdited</code> <code>isDocumentEdited</code>	<p>These are the accessors that let you keep track of whether a document is "dirty" or not.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>

More Window Manipulation The methods in Table 10-29 let you manipulate window attributes like titles and let you close

and miniaturize windows. Be careful with closing and miniaturizing; remember that the essence of the graphical user interface is that the user should be in control. You may think the user experience is enhanced if you miniaturize every window except the active one, but not everyone would agree with you.

Windows often represent the contents of documents; you can associate a file with a window explicitly. You can also title a window with a file's path that is presented in a recognizable file name format, or with a title that is of your own choosing and formatting.

TABLE 10-29. `NSWindow` More Window Management Methods

<code>close</code> <code>performClose</code>	The first method closes the window without any notification; call this method to close a window if that is your case. Call <code>performClose</code> by preference, since that will notify the window's delegate and invoke any custom close behaviors.
<code>miniaturize</code> <code>performMiniaturize</code>	These methods are comparable to <code>close</code> and <code>performClose</code> .
<code>demiaturize</code>	This expands a miniaturized window.
<code>isMiniaturized</code>	Returns a <code>BOOL</code> indicating if the window is miniaturized.
<code>setTitle</code> <code>title</code>	Accessors for the window's title. Set: <code>aString</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code> .

TABLE 10-29. `NSWindow` More Window Management Methods (Continued)

<code>setRepresentedFilename</code> <code>representedFilename</code>	Accessors for the window's file. Set: <code>path</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code> .
<code>setTitleWithRepresentedFilename</code>	This method combines <code>SetRepresentedFilename</code> and <code>setTitle</code> . Its single argument is <code>path</code> (<code>NSString*</code>).
<code>setReleasedWhenClosed</code> <code>isReleasedWhenClosed</code>	Accessors for what to do with the window when it is closed. For windows, the default is YES; for panels, the default is NO. Set: <code>flag</code> (<code>BOOL</code>). Get: returns a <code>BOOL</code> .
<code>setMiniwindowImage</code> <code>miniwindowImage</code>	Accessors for the image of the miniaturized window. Set: <code>anImage</code> (<code>NSImage*</code>). Get: returns an <code>NSImage*</code> .
<code>setMiniwindowTitle</code> <code>miniwindowTitle</code>	Accessors for the miniaturized window's title. Set: <code>aString</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code> .

Menus, Cursors, and Dragging One of the features of the framework that is attractive to many programmers is the fact that it maintains a windows menu for you. This is one of those common routines that has been written over and over and over again: it's a relief to have it done with.

Managing cursors and dragging are shown as well in Table 10-30; these are fairly routine chores that are done in other

frameworks and should be familiar to you. And since much of this work is similar to the work that `NSView` does, it should be even more familiar.

TABLE 10-30. `NSWindow` Menus, Cursor, and Dragging Methods

<code>setExcludedFromWindowsMenu</code> <code>isExcludedFromWindowsMenu</code>	Accessors to determine if this window should be in the Windows menu. Set: flag (BOOL). Get: returns a BOOL.
<code>disableCursorRects</code> <code>enableCursorRects</code>	These methods turn off/on the default cursor handling.
<code>areCursorRectsEnabled</code>	Returns a BOOL indicating if the window's cursor rectangles are enabled.
<code>discardCursorRects</code>	Discards all the window's cursor rectangles. You rarely need to call this.
<code>invalidateCursorRectsForView</code>	Invalidates the cursor rectangles for <code>aView</code> (<code>NSView*</code>). Call this as necessary.
<code>resetCursorRects</code>	Discards and resets all cursor rectangles for the windows and the view that it contains. This is what you call if you haven't a clue which view has invalid cursor rectangles.
<code>dragImage</code> <code>registerForDraggedTypes</code> <code>unregisterDraggedTypes</code>	These methods are comparable to the corresponding <code>NSView</code> methods.

Finally... Table 10-31 wraps up the last methods of `NSWindow`.

TABLE 10-31. NSWindow Final Methods

<code>setHidesOnDeactivate</code> <code>hidesOnDeactivate</code>	<p>The default for panels is YES (as it should be for all windows in that level); the default for document windows is NO.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>worksWhenModal</code>	<p>Returns a BOOL indicating whether this window (typically a panel) is active while another is modal. (Most windows are active while they themselves are modal.)</p>
<code>setContentView</code> <code>contentView</code>	<p>Accessors for the window's content view. Normally you set this in Interface Builder; however, in cases where a window can display alternate views of the same data, it often makes sense to have alternate view hierarchies ready to swap in and out. If you do, remember that a view is released when it is removed from the window unless you retain it explicitly (and then release it explicitly).</p> <p>Set: <code>aView</code> (NSView*). Get: returns an NSView*.</p>
<code>setBackgroundColor</code> <code>backgroundColor</code>	<p>Accessors for the window's background color.</p> <p>Set: <code>aColor</code> (NSColor*). Get: returns an NSColor*.</p>
<code>styleMask</code>	<p>Returns the window's style mask as an unsigned int.</p>

TABLE 10-31. NSWindow Final Methods (Continued)

<code>validRequestorForSendType</code>	Returns the id of an object (if any) that responds to <code>sendType</code> (NSString*) and <code>returnType</code> (NSString*).
	This method is called from the Services menu; if the window doesn't respond, its delegate and then the application get a chance to return an id.
<code>fax</code> <code>print</code>	These are action methods with the argument <code>sender</code> (id); you usually wire them to interface elements.
<code>dataWithEPSInsideRect</code>	Comparable to the <code>NSView</code> method.
<code>setDelegate</code> <code>delegate</code>	Accessors for the window's delegate. Set: <code>anObject</code> (id). Get: returns an id.

Window Delegate Methods

Rather than override an `NSWindow`, you typically add a delegate to the window. The delegate receives any message that the window can't handle—so you can send to your window any custom message that you want and can rely on the delegate receiving it. (For C++ programmers, the notion of calling an `NSWindow`'s `MyMethod` routine may seem risky, but that's what dynamic languages are all about.)

In addition to possible custom methods, your delegate should implement the methods in Table 10-32. These methods can be empty, but they are there to receive notifications and messages that you are likely to need to override.

TABLE 10-32. NSWindow Delegate Methods

<code>windowShouldClose</code>	This action method from <code>sender</code> (<code>id</code>) returns a <code>BOOL</code> if <code>sender</code> should be allowed to close the window.
<code>windowWillResize</code>	Based on a request from <code>sender</code> (<code>NSWindow*</code>) and a <code>proposedFrameSize</code> (<code>NSSize</code>); returns an <code>NSSize</code> . You can use this method to constrain window resizing.
<code>windowWillReturnFieldEditor</code>	<code>sender</code> (<code>NSWindow*</code>) an <code>Object</code> (<code>id</code>).
<code>windowDidBecomeKey</code> <code>windowDidBecomeMain</code> <code>windowDidChangeScreen</code> <code>windowDidDeminiaturize</code> <code>windowDidExpose</code> <code>windowDidMiniaturize</code> <code>windowDidMove</code> <code>windowDidResignKey</code> <code>windowDidResignMain</code> <code>windowDidResize</code> <code>windowDidUpdate</code> <code>windowWillClose</code> <code>windowWillMiniaturize</code> <code>windowWillMove</code>	Each of these methods takes the single <code>aNotification</code> (<code>NSNotification*</code>) argument.

Summary

The basic concepts in this chapter—views and windows—are not new to experienced users of object-oriented frameworks. The concepts of cells, boxes, and images are probably also not new: many people have implemented (or thought of implementing) such lightweight objects.

You can bring whatever experience you have with other frameworks along here: you need not use cells, boxes, and images—you'll write more code and you'll have a more sluggish application—but you can stick with the architectures that you know.

Other lightweight objects mentioned in this chapter such as formatters and text objects contribute to the flexibility of the framework.

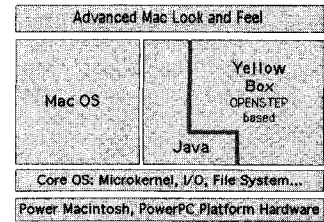
This is the longest chapter in the book: not surprising, since the framework is about building applications with sophisticated and powerful graphical user interfaces. However, since so much of the material is common to other frameworks, with any luck it won't discourage or daunt you.

In fact, there is even more to visualization—special views to manage scrolling and text, for example. They are similar to objects in other frameworks, and you generally use them in Interface Builder without modifying them.

The next chapter—which deals with controls—also touches on features that are common to other frameworks. It, though, is more special than this chapter. Interface Builder and the ability to wire objects together easily using a graphical user interface (for the programmer!) make the deployment and programming of controls much easier in the Rhapsody environment.

Chapter 11

Interface Design and Controls



The objects described in this chapter form the basis of the framework's interface. They build on visualization objects previously discussed (views and cells in particular). You normally use these objects by programming in Interface Builder.

You don't often have to override or subclass these methods: the areas where you need to customize them are changeable by replacing or overriding delegate objects or special-purpose objects associated with them.

The classes discussed here—command-related objects such as menus, panels, and the drag-and-drop protocols and controls such as buttons—are enough for you to implement many very sophisticated interfaces. The design and architecture of these objects are re-

peated through the framework: if you need to create custom interface objects or if you want to get an understanding of other Rhapsody interface elements, this chapter should do the trick.

Commands

The major command support for the framework's interface falls into three areas:

1. menus
2. drag-and-drop
3. panels

Other elements of the interface allow users to generate commands, but these are the primary sources of commands that must be carried out by the application's code.

Menus

For a graphical user interface, menus are somewhat of an embarrassment. The basic functioning of the interface is quite simple:

1. The user selects something that is to be acted on. Using a pointing device (generally a mouse, sometimes with the addition of a shift key for multiple selections), the user clicks on or outlines the object(s) to be selected. This is very simple and straightforward—graphical user interface deities of direct manipulation and user control are certainly smiling.
2. The user selects a command from the menu bar. These commands are expressed (to a greater but usually lesser) extent in a word or two. They are organized together in a logical structure of menus and submenus according to the designs of the software developers. And they insist on acting on the last item(s) selected

rather than on what you choose to select as soon as you have decided to do something. (How many people have been frustrated by word processing applications that refuse to italicize the current—unselected—word in response to the *Italics* command?) Menus—text-based, organized at the developer's convenience not the user's, and requiring users to do things in a certain way—would seem a little out of place in contemporary graphical user interfaces.

The Evolution of Menus in Graphical User Interfaces In fact, menus are assuming less and less importance in many system designs. They were incredibly useful in the early days of graphical user interfaces, but the advent of more powerful computers and larger monitors means that it is possible to pull control out of the menu bar and place it in palettes and panels that are closer to the user's activities and the appearance of which can be modified by users much more easily than can the appearance of menu bars.

But hark! What is that sound of grisly battle? As menus assume less and less importance in system designs, developers are suiting up for yet another holy war: this one about where menus should be. The choices are numerous: at the top of the screen (standard Mac OS behavior), at the top of each window (standard Windows behavior), at the left of the screen (standard NextStep/OpenStep behavior), or contextually wherever the mouse happens to be when you click a menu modifier key that pops up the menus. Subbattles of the holy war are fought over pull-off menus that you can park wherever you want them on the screen. As for keyboard equivalents (command key combinations such as command-P for print) and mnemonics (Receipt, for example, with c being the keyboard equivalent), feelings run just as high as they do for everything else involved with menus.

The differences here are really not great; Rhapsody in its own environment or when running on other platforms presents its menus in the appropriate ways. The classes and methods in

this section let you manage them easily and let the framework worry about their display.

There are a number of user interface issues that Apple is working on during the early releases of Rhapsody, and menus are one of the most important. The issue involves not just where the menus are, but the definition of a menu as well. In NextStep/OpenStep, each application had a menu that contained a number of menu items. In traditional Mac OS terms, this “menu” was a cross between a menu and a menu bar—its menu items could be commands or could open submenus. If you use Interface Builder and the classes and methods described here, your application will work no matter how these design issues are resolved. However, if your application relies on a specific way to present menus, it may break as changes to the Rhapsody user experience are made.

Long-time users of graphical user interfaces are familiar with this: applications that use the Menu Manager in Mac OS have no trouble supporting Apple Guide menu coach marks. Applications that simulate menu bars using the most imaginative and extraordinary procedures break not only Apple Guide menu coaches but also contextual menus in Mac OS 8.

In designing your software and thinking about its interface, you may want to give serious thought to the menu issue. Although menus are very useful in many cases, in many more cases they represent an interface that has outlived much of its usefulness. Watch yourself as you use software—whether compilers, word processors, or games. How often do you use the menu bar, and how often do you use other control mechanisms?

Of course, you can’t put every command on a panel or palette—just as you can’t put every command in the two or three-word description in a menu. The software community has done relatively well at finding ways to communicate the concepts of italicization and printing; if software is ever to do anything of use to the billions of people in the world who are not particularly interested in producing memos and happy birthday party banners, sophisticated commands must be able to be invoked by users to do the sorts of things that users expect computers to be able to do.

The principal components of the framework's menu structure are:

1. Menus.
2. Menu action responders—this is a protocol that controls the automatic enabling and disabling of menu items; you almost always use it.
3. Menu items—these are the items within a menu: they can be either commands or submenus. Menu items are both a class and a protocol.

You rarely override the classes involved in menu handling; if you use Interface Builder, you will find that you have little need to call the methods described here—it and the framework do all the work for you. If you are absolutely determined to change the default behavior of menus, consider implementing this behavior with objects other than menus. You will probably break Rhapsody's cross-platform implementations, produce unmaintainable code, and confuse your users if you insist on "improving" the menu classes.

Menus and Dynamism If you have used frameworks based on static languages (such as MacApp with C++) you are probably used to the menu architecture:

- Each menu item is assigned a command number in its resource description.
- When a menu is selected, that command number is passed through the application to a number of methods that may respond to it. Each method is then free to do whatever it wants in order to implement the command; often another method is called, but many times the dispatching method itself implements a number of menu commands.

This structure allows the menu names and their locations to be changed in the resource file independently of the code; it has proved to be very resilient.

With a dynamic environment, however, this architecture is greatly simplified. No longer is there any need for a dispatching table of command numbers for menu items. Instead, the menu item can have a selector for a particular method associated with it; at run time, that method and the argument(s) are invoked by the command. Interface Builder lets you link menu items to action methods of any object in your interface so that you can construct these menus much more easily than if you had to build the dispatching tables.

Although dispatching tables with command numbers are not particularly difficult to implement or maintain, nevertheless they do provide the opportunity for bugs to creep in. For example, you can associate command number 887 with the menu command Footnote and with the call to the `DoCreateFootnote` method using two simple but separate programmatic statements. Both lines of code must be correct—and no other assignment to command number 887 can take place within the application. In an environment of shared libraries, this last condition has started to become increasingly problematic.

If you are used to the other structure, you may have to do some minor rethinking of the typical architectures that you create; the bottlenecks of menu commands in views and documents do not necessarily exist any longer—the menu command goes directly to the object that acts on it. This may affect applications that are interested in logging their actions and in providing command-level security. It's not particularly a problem, but you should be aware of it if you're not used to the structure.

NSMenu The `NSMenu` object is created by Interface Builder and manages the application's menus. The only methods that you normally invoke are those that remove, add, or retitle menu items. However, before you do this consult the very large literature on the subject of changing menus in a running

application. Or better yet, remember how you felt when you couldn't find a menu item that was there a minute ago in some application that you regularly use. Disabling items is fine: moving them around tends to be very annoying.

The menu items within a menu all have a title and a selector. They may also have a key equivalent, and—if the platform supports it—a mnemonic. Finally, you may add a tag (an integer) to any menu item so that you can locate it quickly. Titles, key equivalents, and tags should all be unique within a menu (although if you have not assigned a key equivalent or a tag to a menu item all of those items will have equal values—and that is fine). In the case of duplicates, routines that find menu items by these values normally return the first occurrence.

NSMenu methods are shown in Table 11-1 as they existed in the Spring of 1997. The implementation of menus is changing for Rhapsody Premiere Release, and the internal architecture may differ in your version of Rhapsody. Many of the general methods remain.

TABLE 11-1. NSMenu Methods

<code>+setMenuZone</code> <code>+menuZone</code>	Accessors for the zone from which NSMenu objects should be allocated. Set: <code>zone</code> (NSZone*). Get: returns an NSZone*.
<code>initWithTitle</code>	Initializes an NSMenu with <code>aTitle</code> (NSString*). Remember that this object contains menu items; it can be a Mac OS menu bar, a regular menu, or any collection of menu items.

TABLE 11-1. NSMenu Methods (Continued)

<code>setTitle</code>	<p>Accessors for the menu's title.</p> <p>Set: <code>aString</code> (NSString*).</p> <p>Get: returns an NSString*.</p>
<code>addItemWithTitle</code> <code>insertItemWithTitle</code>	<p>These methods return the id of a new NSMenuItem with title <code>aString</code> (NSString*), <code>selector</code> (SEL), and <code>keyEquiv</code> (NSString*)—key equivalent.</p> <p>The first inserts the new menu item at the end of the menu; the second takes a final index (unsigned int) argument that locates the menu item in the menu.</p>
<code>itemArray</code>	Returns an NSArray* of the menu's menu items.
<code>itemWithTag</code>	Returns the id of the menu item with <code>aTag</code> (int).
<code>itemWithTitle</code>	Returns the id of the menu item with title <code>aString</code> (NSString*).
<code>removeItem</code>	Removes <code>anItem</code> (id <NSMenuItem>) from the menu.
<code>setSubmenu</code>	<p>You create a submenu by creating an NSMenu and attaching it to a menu item with this method. The arguments are <code>aMenu</code> (NSMenu*)—the menu to be added, and <code>anItem</code> (id<NSMenuItem>)—the item to open the submenu.</p>

TABLE II-1. NSMenu Methods (Continued)

<code>submenuAction</code>	All menu items should have action methods associated with them. Those that perform commands have action methods from other application objects wired to them; submenu commands—those that open submenus—are wired to this method of the submenu. Its argument is <code>sender</code> (<code>id</code>).
<code>locationForSubMenu</code>	Returns the <code>NSPoint</code> where <code>aSubMenu</code> (<code>NSMenu*</code>) of this menu will open; not valid on all Rhapsody platforms.
<code>supermenu</code>	For submenus, returns the <code>NSMenu*</code> they are in; returns <code>nil</code> otherwise.
<code>attachedMenu</code>	
<code>isAttached</code> <code>isTornOff</code>	Returns a <code>BOOL</code> indicating the status of menus.
<code>update</code>	Explicitly causes all menu items to be reenabled or disabled, depending on their status.
<code>performKeyEquivalent</code>	Returns a <code>BOOL</code> if a menu command with the key equivalent embodied in <code>theEvent</code> (<code>NSEvent*</code>) can be found; if so, the action is performed.
<code>setAutoEnablesItems</code> <code>autoenablesItems</code>	Accessors determining if the menu automatically enables and disables items. The default is <code>YES</code> . Set: flag (<code>BOOL</code>). Get: returns a <code>BOOL</code> .

TABLE 11-1. NSMenu Methods (Continued)

setMenuChangedMessages Enabled menuChangedMessagesEnabled	Accessors determining if notifications are sent to the application when the menu changes. Set: flag (BOOL). Get: returns a BOOL.
sizeToFit	On platforms where implemented; the menu is sized to fit its items exactly; on other platforms, menus are exactly the size of their items.
helpRequested	Used internally. Rhapsody's help architecture is being reimplemented from the OpenStep design. Consult the documentation with your version of the framework for further details of the help system.

NSMenuValidation [P] The dynamic nature of menu management in the framework permits menu enabling to be done slightly differently than it is in other frameworks. In frameworks based on static languages and that contain menu command dispatch tables, the framework normally disables all menu items when it starts to set up menus (which it does frequently in response to user events). After having disabled all menu items, the application itself is allowed to enable those items that it can process; the target or responder chain is then followed down to the application target or first responder allowing each item in the chain to modify menu commands—all based on their command numbers.

Here, since there is no command dispatch table, it is much simpler. Each menu item has the selector of an action associated with it; the menu item's target (if any) and then the responder chains of the key window (if any), main window (if any), and application object are asked if they respond to the

selector. As soon as one of these objects is found that responds to the selector, the item is enabled; otherwise, it is disabled.

This processing relies on finding an action method in one of these objects that matches the action method in the menu. It is very fast and efficient, but it does not allow the more complex enabling and disabling that can be done when objects implement methods such as `DoSetupMenus` (in `MacApp`). For these cases, the `NSMenuValidation` protocol exists. You implement its single method, `validateMenuItem`, if you want sometimes to disable a menu item that would otherwise be enabled because your object does implement the action method. For example, you may implement a `DoItalicize` method, but you may not want to enable the `Italicize` command if the text is already italicized. In general, data-dependent commands require you to implement the `validateMenuItem` method.

`NSMenuValidation` is described in Table 11-2;

TABLE 11-2. `NSMenuValidation` [P] Method

`validateMenuItem`

Returns a `BOOL` indicating if `aMenuItem` (`NSMenuItem*`) should be enabled. This method is only called if your object has an action method that is referenced from the menu item.

`NSMenuItem` [P] There is a `NSMenuItem` class, but it is not yours to play around with. It adopts the `NSMenuItem` protocol, and you can override the methods of that protocol if you need to. Remember that each menu item has a title, a target, an action, a key equivalent, and a mnemonic (although any of them—except the title—can be `nil`). You may also assign an integer tag to a menu item in order to locate it. Using tags lets

you manipulate menu items without regard to their localized titles.

Table 11-3 lists the protocol's methods.

TABLE 11-3. NSMenuItem [P] Methods

<code>setTitle</code> <code>title</code>	Set: <code>aString</code> (NSString*). Get: returns an NSString*.
<code>setTarget</code> <code>target</code>	Set: <code>anObject</code> (id). Get: returns an id.
<code>setAction</code> <code>action</code>	Set: <code>aSelector</code> (SEL). Get: returns a SEL.
<code>setKeyEquivalent</code> <code>keyEquivalent</code>	Although the key equivalent must be able to be generated with a standard key on the user's keyboard, it may in fact consist of more than one Unicode character. Set: <code>aString</code> (NSString*). Get: returns an NSString*.
<code>setMnemonicLocation</code> <code>mnemonicLocation</code>	The location in the menu item's title for the mnemonic character. Set: <code>location</code> (unsigned). Get: returns an unsigned.
<code>setTitleWithMnemonic</code>	Sets the title to <code>aString</code> (NSString*) where the string is assumed to contain a single & symbol that precedes the mnemonic; the & is stripped from the title.
<code>setTag</code> <code>tag</code>	Set: <code>aTag</code> (unsigned int). Get: returns an unsigned int.

TABLE 11-3. NSMenuItem [P] Methods (Continued)

setEnabled isEnabled	<p>You normally rely on the mechanism described previously to enable and disable menu items. If necessary, implement validateMenuItem to refine enabling.</p> <p>Set: flag (BOOL). Get: returns a BOOL.</p>
hasSubmenu	Returns a BOOL.
mnemonic	Returns an NSString*.
+setUsesUserKeyEquivalents +usesUserKeyEquivalents	<p>Accessors allowing users (for all menu items) to modify your assigned key equivalents. If YES, the following method has meaning.</p> <p>Set: flag (BOOL). Get: returns a BOOL.</p>
userKeyEquivalent	Returns an NSString* that the user may have assigned.
setKeyEquivalentModifierMask keyEquivalentModifierMask	<p>Platform-specific modifiers for keyboard equivalents. You should probably leave the defaults in place to avoid confusion.</p> <p>Set: mask (unsigned int). Get: returns an unsigned int.</p>

TABLE 11-3. NSMenuItem [P] Methods (Continued)

<code>setRepresentedObject</code> <code>representedObject</code>	<p>You can associate any object with a menu item; if the command involves an action by an object (usually the target) that is carried on by another object, this is where you can prepare for that. For example, the target could be a panel or window; the represented object could be a view to be inserted, a color to be changed, etc.</p> <p>Set: anObject (id). Get: returns an id.</p>
---	---

Drag-and-Drop Drag-and-drop technology provides the direct manipulation and user control that are so welcome in the graphical user interface. Whereas menu commands must always be reduced to some short verbal phrase, drag-and-drop allows you to move objects of great complexity around on the screen.

Drag-and-drop requires significant processing power to support its constant feedback; it also requires support from the operating system and the cooperating applications between and through which the dragging occurs.

Many people feel that this technology will provide one of the gateways to the more sophisticated applications of the future. While dragging selected text and images, fonts, and colors around is common today, consider the possibilities of dragging complex objects.

- You could create an age range object that represents the features of a certain age group. You could drag such an object onto a document and have its text and graphics be modified (or at least have suggestions for modifications created). Such modifications might well involve parsing the text and possibly restructuring

sentences; other modifications might involve changes to the size and complexity of illustrations.

- You could also create the schematic of a physical location, complete with light source, walls of various heights, reflecting surfaces, etc. You could then drag it onto an image to simulate placing the image in such a context. (This is far more complicated than changing a light source, but it is not beyond the capabilities of current computers and developers.)

Drag-and-drop is primarily implemented with three protocols that any object can adopt: `NSDraggingInfo`, `NSDraggingSource`, and `NSDraggingDestination`.

Dragging is controlled by the `NSView`'s `dragImage` method, which calls the various methods of these protocols as needed. The two endpoints of the drag are the source and the destination objects; during the drag, the operation relies on an image that moves in response to the user's action. At the start of the operation, the data to be dragged is placed on a custom pasteboard; on completion, it is removed from that pasteboard and inserted appropriately in the destination.

As far as the drag-and-drop operation is concerned, the source and destination must be views or windows. Drag-and-drop is an interface activity, and no other possibility exists. Moving data from one document to another—which may be the result of the drag-and-drop operation—can be done without any visual feedback.

Since a drag can initiate any of several data transfers that depend on the destination, you need to specify them repeatedly depending on where the drop might occur. These are the kinds of drops that can occur: `NSDragOperationCopy`, `NSDragOperationLink`, `NSDragOperationGeneric` (the destination defines the drag), `NSDragOperationPrivate` (the source and destination together define the drag), `NSDragOperationAll`, and `NSDragOperationNone`.

NSDraggingSource [P] The start of a dragging session is controlled by the view from which something is being dragged. This protocol adds four methods to views that support dragging. You need only to implement one of them, as noted in Table 11-4. If the drag moves data from the source to the destination, you should implement the `draggedImage` method that is called on completion in order to remove data from the source; if the drag functions as a copy, nothing need be done to the source.

TABLE 11-4. NSDraggingSource [P] Methods

<code>draggedImage</code>	At the start of a drag, this method is called with <code>anImage</code> (<code>NSImage*</code>) and <code>aPoint</code> (<code>NSPoint</code>). The view's <code>dragImage</code> method will move the image; if you want to provide additional feedback within the document you can override this method. Be aware, though, that doing something else at this point (like rearranging text in a paragraph) may confuse the user and distract from the primary dragging feedback.
<code>draggedImage</code>	This method is called on completion of a drag. Its arguments are <code>anImage</code> (<code>NSImage*</code>), <code>aPoint</code> (<code>NSPoint</code>)—the end of the drag in screen coordinates, and <code>flag</code> (<code>BOOL</code>), which indicates if the drag has been successful. You override this method to remove the data from the source window. (This method differs from the previous method by the addition of the third argument.)

TABLE 11-4. NSDraggingSource [P] Methods (Continued)

<i>draggingSourceOperationMaskForLocal</i>	The single argument <code>flag</code> (BOOL) indicates if the method is being called while the potential destination is part of this application's environment. You override this method to return one or more of the drag types outlined previously (you can combine them with OR).
<i>ignoreModifierKeysWhileDragging</i>	This method returns a BOOL indicating if the user can modify the drag with the keyboard. You override it to return YES if you do want to ignore modifier keys.

NSDraggingDestination [P] Remember that the ultimate destination of the drop may be one in a long list of destinations that the dragging operation has encountered. The methods of the NSDraggingDestination protocol shown in Table 11-5 may be implemented as you see fit. Views and windows that actually handle dragging do not rely on any of these methods existing; if they do, they are used during the operation.

TABLE 11-5. NSDraggingDestination [P] Methods

<i>draggingEntered</i> <i>draggingUpdated</i>	Returns a mask using the drag types described previously. The single argument is <code>sender</code> (id<NSDraggingInfo>).
<i>draggingExited</i>	The single argument is <code>sender</code> (id<NSDraggingInfo>).

TABLE 11-5. NSDraggingDestination [P] Methods (Continued)

<i>prepareForDragOperation</i> <i>performDragOperation</i>	These methods are invoked in this order with the single argument <i>sender</i> (id<NSDraggingInfo>). They return a BOOL indicating success at preparing for and then performing the operation. (A NO for the first stops the operation.)
<i>concludeDragOperation</i>	After a YES from <i>performDragOperation</i> (which must follow a YES from <i>prepareForDragOperation</i>), this method is called to allow you to do final cleanup. The single argument is <i>sender</i> (id<NSDraggingInfo>).

NSDraggingInfo [P] The NSDraggingInfo protocol can be added to any object that is draggable; the id of such an object is used in several of these methods. The methods of NSDraggingInfo (Table 11-6) provide the link between source and destination. While dragging sources and destinations must be views or windows, you can quite easily add the NSDraggingInfo protocol to text, a color, or a custom object of your own (such as the age range object in the example).

Each dragging operation has a unique source and is assigned a unique identifier by the system. It has an image associated with it to provide feedback and a pasteboard that contains the data to be moved. During the drag, the operation has a location and a potential destination. The NSDraggingInfo protocol is designed to service the NSDraggingDestination protocol; it is passed into many of the destination methods. You implement the methods of the dragging info protocol that you want to call from the dragging destination.

TABLE 11-6. NSDraggingInfo [P] Methods

<code>draggingSource</code>	Returns the id of the source.
<code>draggingSequenceNumber</code>	Returns an int which is the system-defined unique drag identifier.
<code>draggedImage</code>	Returns an <code>NSImage*</code> .
<code>draggingPasteboard</code>	Returns an <code>NSPasteboard*</code> .
<code>draggingLocation</code>	Returns an <code>NSPoint</code> which is the current location in the coordinates of what would be the destination window if the drag concluded now.
<code>draggedImageLocation</code>	You override this method to return an <code>NSPoint</code> which is constrained to a grid or otherwise modified based on conditions in the destination (for example, the dragged object might be insertable only between paragraphs and the image location should reflect this).
<code>draggingDestinationWindow</code>	Returns the <code>NSWindow*</code> of the destination.
<code>draggingSourceOperationMask</code>	Returns an unsigned int which is the mask returned by the <code>NSDraggingSource's draggingSourceOperationMaskForLocal</code> method.
<code>slideDraggedImageTo</code>	This method moves the image to a <code>Point</code> (<code>NSPoint</code>) in screen coordinates. It is called only at the end of the drag; to make modifications along the line, override <code>draggedImageLocation</code> .

Panels

Between the interfaces of menus and drag-and-drop, panels present an approach to the graphical user interface that is usu-

ally less text oriented than menus but is not quite so intensive as drag-and-drop (if the user lets up on the mouse button, drag-and-drop ends).

Panels are windows that live in a level in front of application windows. By convention, all of the panels in this level are hidden when the application is inactive, although the application's content windows remain visible. Panels can become key windows but never can be main windows of an application. You often use standard panels for fonts and colors; if you need to create your own panels, you can subclass the `NSPanel` method.

Even within the level of panels, some panels can float above others. In addition, you can specify that a panel becomes active (the key window) only when clicked on: this allows you to create panels that are used for informational purposes rather than (or in addition to) data entry. You also can set a panel to function when another window is modal. Accessors for these fields are shown in Table 11-7.

TABLE 11-7. `NSPanel` Methods

<code>setFloatingPanel</code>	Set: flag (BOOL).
<code>isFloatingPanel</code>	Get: return a BOOL.
<code>setbecomesKeyOnlyIfNeeded</code>	
<code>becomesKeyOnlyIfNeeded</code>	
<code>setWorksWhenModal</code>	
<code>worksWhenModal</code>	

Controls

Controls are graphical elements that let users manipulate the interface. Buttons, sliders, and pop-ups are common controls: each has a distinctive form of representative, each responds to user actions, and each contains some data that can be modified by the user action.

NSControl

Controls are special-purpose views that in most frameworks are fairly complex objects. This is partly because controls have so much to do; it is also due to the fact that in most graphical user interfaces, standard controls are provided. A framework's control objects typically interact with and use the platform's control devices—which normally are not objects. This is one of the boundaries between the world of a framework's objects and the flat world of the operating system.

In Rhapsody, you don't have to worry about this: the platform's controls are indeed objects. Furthermore, control objects in the framework are just as complex as they are in other frameworks, but that complexity has been broken apart into manageable pieces. The cell object (see "NSCell" on page 216) is a lightweight object that can display images and text and that supports many of a view's methods. It does this without being a view but by relying on its having been placed inside a view that interacts intimately with it. The view into which most cells are placed is a control view—a descendant of NSControl.

Thus, the cell object can manage much of the interface as well as much of the drawing. Furthermore, since cells often contain image objects, the visualization of contents is moved to yet another player. This leaves the NSControl to worry about managing the cell(s) it contains, managing the editing of its content, and passing messages on to the cell(s) it contains.

In Table 11-8 you will find the basic methods of NSControl. This is an abstract class: you normally use NSSliders, NSBut-

tons, etc.—and you rarely instantiate or override them (the default classes that you program in Interface Builder are usually just fine). If you do need to modify the behavior of standard controls, you can do so by changing their cells or the cells' image objects.

The most common case for controls is that they contain a single cell of some sort. Some of the methods listed in Table 11-8 have meaning only for these single-cell controls.

Note also that `NSControl` manages editing of the contents of its cell(s) with three methods of its own as well as with a delegate that is specifically tailored to formatting and data validity checking. This delegate is another way that you can customize the behavior of the control without overriding the control, the embedded cell, or the cell's image. Remember also that an `NSText*` object is placed over active text cells to manage data entry; these objects are called field editors (see "Editing" on page 274).

TABLE 11-8. `NSControl` Methods

`initWithFrame`

Initializes an `NSControl` to `frameRect` (`NSRect`). If a default type of cell has been set for the control (using the following two methods), an instance of that cell is created and placed in the control.

`+setCellClass`
`+cellClass`

Accessors for the kind of cell to be placed in this control by default. For example, `NSButton` (a descendant of `NSControl`) sets its cell class to `NSButtonCell`.

Set: `class` (Class).
Get: returns a Class.

TABLE 11-8. NSControl Methods (Continued)

<code>setCell</code> <code>cell</code>	<p>Accessors for the instance of the cell in the control. Normally this cell is created by <code>initWithFrame</code>; if you set a cell, make certain that you properly dispose of the existing cell.</p> <p>Set: <code>aCell</code> (NSCell*). Get: returns an NSCell*.</p>
<code>selectedCell</code> <code>selectedTag</code>	<p>For controls with multiple cells, these methods return the id of the selected cell or the int of the tag of the selected cell. For controls without a selected cell, values of nil and minus 1 respectively are returned.</p>
<code>drawCell</code> <code>drawCellInside</code>	<p>If <code>aCell</code> (NSCell*) is the control's only cell, the control is drawn with or without its border. If there are multiple cells, <code>drawCell</code> draws the specified cell.</p>
<code>updateCell</code> <code>updateCellInside</code>	<p>Sets <code>aCell</code> (NSCell*) to be redisplayed (soon)—either in whole or just its interior.</p>
<code>sendAction</code>	<p>Asks the application object to send <code>theAction</code> (SEL) to <code>theTarget</code> (id). Returns a BOOL indicating success at sending the message (not necessarily success at completing it).</p>
<code>mouseDown</code>	<p>This is the standard method that processes <code>theEvent</code> (NSEvent*) for the control if that event is a mouse down.</p>

TABLE 11-8. NSControl Methods (Continued)

<code>abortEditing</code>	Returns a BOOL indicating if there was a field editor delegate associated with the control; aborts editing.
<code>currentEditor</code>	Returns the current field editor (if any) as an NSText*.
<code>validateEditing</code>	Validates user's changes to a text cell; if validation succeeds, the text of the field editor is moved to the text cell.
<code>calcSize</code>	Calls the cell's <code>calcDrawInfo</code> method and resizes the control accordingly.
<code>sizeToFit</code>	Resizes the control to be the size of its cell; undefined for controls with other than one cell.
<code>selectCell</code>	Selects aCell (NSCell*) within the control.
<code>setNeedsDisplay</code>	Causes the control and its associated cells to be redisplayed at the next occasion.
<code>setIgnoresMultiClick</code> <code>ignoresMultiClicks</code>	Accessors for the value that passes the second (third, etc.) click in a brief interval on to the control's super for processing. Normally NO. Set: flag (BOOL). Get: returns a BOOL.

NSControl provides the view functionality for cells: a number of view methods are passed on to the embedded cell. These

calls are listed in Table 11-9; they have the same meaning for NSControls as they have for NSCells.

TABLE 11-9. NSControl Methods That Are Described in NSCell

```
setEnabled
isEnabled

setDoubleValue
setFloatValue
setIntValue
setObjectValue
setStringValue

doubleValue
floatValue
intValue
objectValue
stringValue

takeDoubleValueFrom
takeFloatValueFrom
takeIntValueFrom
takeObjectValueFrom
takeStringValueFrom

setAlignment
setFont

alignment
font

setFloatingPointFormat

setAction
setTarget
setContinuous

action
target
isContinuous

sendActionOn
```

TABLE 11-9. NSControl Methods That Are Described in NSCell (Continued)

```
setAttributedStringValue
attributedStringValue

performClick
```

Buttons

A commonly used control is the button. As implemented with the architecture described here, it consists of two objects:

1. **NSButton** (a descendant of **NSControl**)
2. **NSButtonCell** (a descendant of **NSActionCell** and thus of **NSCell**)

(You can see how the architecture can be expanded by looking at **NSPopupButton** and **NSPopupButtonCell**, which implement pop-up menus. Their descriptions can be found in the Rhapsody developer documentation.)

NSButton The major methods of **NSButton** are shown in Table 11-10. (Of course, all of the methods of **NSControl** are also available to **NSButtons**.)

You can see that the major concerns of **NSButton** are involved with managing an alternate title and image for display in the button's alternate state. Interface Builder lets you use buttons of standard types. These types combine a number of features:

- **pushiness**—Does the button appear to be pushed in, using shadows to suggest a three-dimensional look, or does it appear flat?
- **perpetuity**—Does the click of a button change the button's state permanently or just momentarily do something to it. The distinction is best illustrated by the

difference between a button that sets auto-saving of files on and a button that saves a file now.

- special effects—Is this a radio button or some other special case?

Interface Builder supplies various combinations of these features.

TABLE 11-10. NSButton Methods

<code>†setCellClass</code> <code>†cellClass</code>	Accessors for cell class (by default <code>NSButtonCell</code>). Set: <code>class</code> (Class). Get: returns a Class.
<code>setButtonType</code>	The <code>NSButtonType</code> values are reflected in Interface Builder; consult the Rhapsody developer documentation for the latest values of button types (the interface is subject to change during the Rhapsody development cycle). Set: <code>aType</code> (<code>NSButtonType</code>).
<code>setState</code> <code>state</code>	Accessors for the button's state; it can have only two states—0 and something else. Set: <code>value</code> (int). Get: returns an int.

TABLE 11-10. NSButton Methods (Continued)

`setImagePosition`
`imagePosition`

Accessors for the position of the image (if any) relative to the title. Values for `NSCellImagePosition` are `NSNoImage`, `NSImageOnly`, `NSImageLeft`, `NSImageRight`, `NSImageBelow`, `NSImageAbove`, `NSImageOverlaps`.

Set: `aPosition` (`NSCellImagePosition`).

Get: returns an `NSCellImagePosition`.

`setTransparent`
`isTransparent`

Accessors for whether a button is transparent. Transparent buttons are hot and track the mouse but draw nothing. (In the early days of HyperCard, transparent buttons were placed on top of button images that were drawn as part of a background pattern.)

Set: flag (`BOOL`).

Get: returns a `BOOL`.

`setPeriodicDelay`
`getPeriodicDelay`

If the button is set for continuous actions (see `NSControl`'s `setContinuous` method), `setPeriodicDelay` lets you set the `delay` (float) after which messages are sent regularly at `interval` (float). The other method, `getPeriodicDelay`, takes the arguments `delay` (float*) and `interval` (float*).

TABLE 11-10. NSButton Methods (Continued)

<code>setAttributedTitle</code> <code>setAttributedAlternateTitle</code>	Accessors for primary and alternate button titles expressed as attributed strings (with styling information).
<code>attributedTitle</code> <code>attributedAlternateTitle</code>	Set: <code>aString</code> (<code>NSAttributedString*</code>). Get: returns an <code>NSAttributedString*</code> .
<code>setAlternateTitle</code> <code>alternateTitle</code>	Accessors for primary and alternate button titles as common strings. Set: <code>aString</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code> .
<code>setAlternateImage</code> <code>alternateImage</code>	Accessor for alternate image. Set: <code>image</code> (<code>NSImage*</code>). Get: returns an <code>NSImage*</code> .

In addition to the methods that `NSButton` inherits from `NSControl`, `NSButton` implements a number of methods that are similar to methods described in other objects—specifically in `NSCell`, `NSMenuItem`, and `NSView`. Those methods are listed in Table 11-11.

NSButtonCell `NSButtonCell` (the cell inside the `NSButton` object) has relatively few methods to implement. Many of its methods are inherited from `NSCell`; others are common methods that are implemented for other objects.

Table 11-12 lists the remaining `NSButtonCell` methods. They are concerned with implementing titles with a mnemonic and managing the visualization of the cell under various circumstances.

TABLE 11-11. NSButton Methods Described Elsewhere

<code>setKeyEquivalent</code>	See <code>NSMenuItem</code> .
<code>setKeyEquivalentModifierMask</code>	
<code>keyEquivalent</code>	
<code>keyEquivalentModifierMask</code>	
<code>setBordered</code>	See <code>NSCell</code> .
<code>setImage</code>	
<code>isBordered</code>	
<code>image</code>	
<code>highlight</code>	
<code>performClick</code>	
<code>performKeyEquivalent</code>	See <code>NSView</code> .

TABLE 11-12. NSButtonCell Methods

<code>setTitleWithMnemonic</code>	Sets the title, which is assumed to have a mnemonic as the first character following the & symbol. Set: <code>aString</code> (<code>NSString*</code>).
<code>setAlternateMnemonic</code>	Accessors for the location of the mnemonic within the alternate title.
<code>Location</code>	
<code>alternateMnemonicLocation</code>	Set: <code>location</code> (unsigned). Get: returns an unsigned.

TABLE 11-12. NSButtonCell Methods (Continued)

<code>setAlternateTitleWithMnemonic</code> <code>alternateMnemonic</code>	<p>Accessors for the alternate title, which is assumed to contain a mnemonic as the first character following the & symbol.</p> <p>Set: <code>aString</code> (NSString*). Get: returns an NSString*.</p>
<code>setKeyEquivalentFont</code> <code>keyEquivalentFont</code>	<p>Accessors for the font to be used to display the button cell's key equivalent (if any).</p> <p>Set: <code>fontObj</code> (NSFont*). Get: returns an NSFont*.</p>
<code>setKeyEquivalentFont</code>	<p>Accessor for the font (by name) and size for the key equivalent font. Arguments are <code>fontName</code> (NSString*) and <code>fontSize</code> (float).</p>
<code>setImageDimsWhenDisabled</code> <code>imageDimsWhenDisabled</code>	<p>Accessors for whether the cell's image is dimmed when the cell is disabled (default is YES).</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>isOpaque</code>	<p>Returns a BOOL indicating that every pixel of the button's content area and border is drawn.</p>
<code>setHighlightsBy</code> <code>setShowsStateBy</code> <code>highlightsBy</code> <code>showsStateBy</code>	<p>Accessors for the type that controls highlighted and alternate images, three-dimensional appearance, etc. Consult the latest Rhapsody developer documentation for values that reflect the evolving user interface.</p> <p>Set: <code>aType</code> (int). Get: returns an int.</p>

The methods of `NSButtonCell` that are described in other contexts are listed in Table 11-13.

TABLE 11-13. `NSButtonCell` Methods Described Elsewhere

<code>setAttributedTitle</code>	<code>NSButton</code>
<code>setAlternateTitle</code>	
<code>setAttributedAlternateTitle</code>	
<code>setAlternateImage</code>	
<code>setImagePosition</code>	
<code>setPeriodicDelay</code>	
<code>setTransparent</code>	
<code>attributedTitle</code>	
<code>alternateTitle</code>	
<code>attributedAlternateTitle</code>	
<code>alternateImage</code>	
<code>imagePosition</code>	
<code>getPeriodicDelay</code>	
<code>isTransparent</code>	
<code>setButtonType</code>	
<code>setTitle</code>	<code>NSMenuItem</code>
<code>setKeyEquivalent</code>	
<code>setKeyEquivalentModifierMask</code>	
<code>title</code>	
<code>keyEquivalent</code>	
<code>keyEquivalentModifierMask</code>	

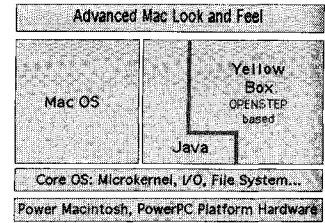
Summary

You can develop a pretty elegant interface with nothing more than the menus and buttons described in this chapter. The tools that make up those objects—the cells, for example, that are found in most `NSControl` objects—encapsulate further functionality and display features.

Rhapsody already combines these features in many ways for other types of interface elements (such as browsers); you can combine them in yet other ways for your own applications.

Chapter 12

Living in a Shared Environment



“Sharing” is used in its broadest sense in this chapter. It refers to the sharing of computer resources and of data over time and space. You may think of sharing as something that doesn’t affect you or your applications—you are resolutely single-user and single-machine in your design. But sharing affects almost every user and every application.

You share the processor and all of the computer resources with the operating system; when you copy data from one application to another you are sharing it via the clipboard; and when you open a document that you created yesterday, you are sharing the data that was stored then. Beyond that, if your computer has two or more processors, they share the responsibility for running your application (and

any other work that they may happen to be doing). Within your application, you may choose to use several threads that can process more or less simultaneously as the operating system optimizes the sharing of its resources; and within your application—even if it is single-threaded—your objects may share themselves and data elements.

OpenStep provides the tools to do all this at a very basic level—so basic that you may not even be aware of what it's doing all the time. Your awareness and implementation of sharing are limited to implementing a few class-specific methods. All of the classes and methods in this chapter are widely used: archiving and unarchiving objects are the essence of Interface Builder (you create the interface objects, it builds them, then archives them in a nib file); the coding methods and protocol are used both for archiving and for pasteboard support; the locking protocol and methods are used for synchronism throughout the system; and NSThread, of course, is the heart of multi-threaded applications, which are so efficient on the Rhapsody OS.

Archiving and Distribution

All object-oriented programming languages make it very easy to create objects at run time: nothing could be simpler. However, as soon as it is necessary to store that object on disk or to move it to a different address space or run-time environment, problems crop up.

The archiving and distribution architecture is described in this section; however, be aware that you can get away quite simply just by implementing the NSCoder protocol for your archivable objects and by calling the appropriate methods of NSCoder. The sophisticated handling of moving graphs of objects to and from memory is done for you by the framework; even the work of managing small graphs of objects (the views of a window, for instance) is done by standard objects (like `NSView`). You don't have to worry about how it happens.

Archiving is the process of taking objects from memory and storing them on disk; distribution is the process of taking objects from your application's address space and preparing them for transfer to another address space dynamically. Both archiving and distribution are based on the abstract superclass `NSCoder` and on the `NSCoding` protocol. `NSObject` adopts the `NSCoding` protocol and has specific methods to support `NSCoder` and the `NSCoding` protocol; as a result, all `NSObject`s have the basic ability to be archived and distributed (although not all implement the protocol).

Archiving is the general process of moving objects from memory to disk—and vice versa. Where a distinction needs to be made in the direction, unarchiving is used, and it is noted in the text that archiving refers only to the to-disk direction. Otherwise, archiving is used without regard to direction.

Coding

Coding is the process of converting an object to a format that can be shared—either on disk in a file or with another process on the same machine or elsewhere on the network. `NSCoder` objects are passed around in various `NSObject` methods as objects are being encoded and decoded; in practice, the objects that are passed around are descendants of `NSCoder`: `NSArchiver` and `NSUnarchiver` for files, and `NSPortCoder` for passing objects to another process or another machine on the network.

NSCoding [P] The two methods of the `NSCoding` protocol must be implemented for any of your objects that you want to be able to archive or distribute. The methods are responsible for calling the superclass's method to encode or decode all of the superclass's data (and that of its superclasses). You then call the method on your class's own data.

The `NSCoding` protocol is shown in Table 12-1

Even if your class has no instance variables, you still should consider implementing these methods as stubs with their calls

TABLE 12-1. NSCoder [P] Methods

<i>encodeWithCoder</i>	Uses aCoder (NSCoder*) to encode the object.
<i>initWithCoder</i>	Uses aDecoder (NSCoder*) to initialize an instance that is returned as type id.

to their superclass's methods; if you subsequently add instance variables, you will have the place to add their encoding and decoding processing. You should get in the habit of making certain that your instance variable declarations match your encoding and decoding methods as you go along: it makes debugging easier.

Clearly, objects that are purely run-time constructs (such as the NSArchiver itself) need not have these methods implemented. In addition, objects that can clearly be reconstituted from data stored in other objects may not need to be stored.

In order to implement archiving and distribution, you override the `encodeWithCoder` and `initWithCoder` methods of the objects you want to archive or distribute. These are methods of the NSCodering protocol that is adopted by NSObject, so any of your objects can override them.

The structure of each method is the same: a single parameter—coder, of type NSCoder*—is passed in. The first line of the method invokes the superclass's method, thus encoding or decoding all of this object's superclass(es). For encoding, the line is:

```
[super encodeWithCoder: coder]; // classic
super.encodeWithCoder (coder); //modern
```

The syntax varies slightly for decoding:

```
self = [super initWithCoder: coder]; //classic
self = super.initWithCoder (coder); //modern
```

The variation allows you to substitute another object in the middle of decoding. For example, you might implement `initWithCoder` and `encodeWithCoder` for an abstract superclass that you create—perhaps a `Customer` class. As you are decoding the object, you might find information that indicates that this particular instance is not an instance of the abstract superclass `Customer` but an instance of the concrete subclass `PayingCustomer` or `ProBonoCustomer`. At that moment, you can create the appropriate concrete class and return it from your `initWithCoder` method (which returns an id). In order for this architecture to work, each `initWithCoder` method returns either self or the new object that it has instantiated.

After you have called your superclass's `encodeWithCoder` or `initWithCoder` method, you then call the coder's appropriate methods to encode or decode your object's instance variables. These should be called in the same order so that the data is not corrupted as it is encoded and decoded. Remember also that not every instance variable needs to be initialized via decoding. Some have values that can be surmised from other values: in these cases, you need not encode or decode the values. In other cases, purely run-time values (mouse positions within views, for example) should be initialized to a default value by `initWithCoder`; normal practice is to remember window positions on a monitor (thus encoding and decoding that value) but not to remember mouse positions within a view.

If you are wondering about the apparent lack of symmetry between `encodeWithCoder` and `initWithCoder` (as opposed to a `decodeWithCoder` method), remember that on encoding you are starting from an existing object in memory and are writing it out. The reverse situation takes a data stream and must constitute an object from it. For decoding, you normally call an `alloc` routine to allocate an object and then you initialize it with `initWithCoder`. The structure of `initWithCoder` allows substitution if the object that you allocated was not the object that you wind up wanting. Thus, encoding is a one-step operation (since the object already exists) and decoding is a two step operation (creating the object, then initializing it with the decoded data).

NSCoder NSCoder has the basic routines to encode and decode basic types. You override NSCoder by overriding NSArchiver and NSUnarchiver or by overriding NSPortCoder. Obviously, you rarely override the methods that encode and decode standard types.

The encoding and decoding methods for standard data types and objects are shown in Table 12-2. These routines are called as necessary depending on your data types. You rarely override them—unless you want to store common data types in another format. You don’t even need to override or create new methods to archive your own objects—you call methods like `encodeObject`, which encodes a generic object and winds up calling that object’s encoding methods. So don’t think that you have to write an `encodeMyObject` method.

Arguments for encoding and decoding methods are the same unless otherwise shown in Table 12-2.

TABLE 12-2. NSCoder Coding Methods

<code>encodeValueOfObjCType</code> <code>decodeValueOfObjCType</code>	The primary methods take an argument of <code>type</code> (<code>const char *</code>), which describes the type of the data to be converted, which is passed in the second argument, <code>address</code> (<code>void*</code>).
<code>encodeValuesOfObjCTypes</code> <code>decodeValuesOfObjCTypes</code>	You can pass a variable-length list of <code>types</code> (<code>const char*</code>) to be converted (not available in Java or outside your own process).
<code>encodeArrayOfObjCType</code> <code>decodeArrayOfObjCType</code>	Methods for arrays of standard C types. The array consists of variables of <code>type</code> (<code>const char*</code>), has <code>count</code> (unsigned) elements, and is called <code>array</code> (<code>const void*</code>).

TABLE 12-2. NSCoder Coding Methods (Continued)

encodeNXObject decodeNXObject	<p>Coders for descendants of Object (provides backwards compatibility with pre-Rhapsody OpenStep).</p> <p>Encodes: object (Object*). Decodes: returns an Object*.</p>
encodeObject decodeObject	<p>Coders for Objective-C objects</p> <p>Encodes: anObject (id). Decodes: returns an id.</p>
encodeDataObject decodeDataObject	<p>Coders for NSData objects.</p> <p>Encodes: data (NSData*). Decodes: returns an NSData* object.</p>
encodePropertyList decodePropertyList	<p>Coders for property lists (which may be NSData, NSArray, NSDictionary, or NSStrings).</p> <p>Encodes: plist (id). Decodes: returns an id.</p>

The three methods shown in Table 12-3 all simply call `encodeObject` by default. You can override them if you want to exert finer control over the archiving process. You rarely do so, however, because `NSArchiver` (a descendant of `NSCoder`) does it for you.

TABLE 12-3. NSCoder Object Coding Methods

<code>encodeRootObject</code>	Encodes anObject (id) and all of the objects that are connected to it; for example you would use encodeRootObject to encode a window and all of its views.
<code>encodeByRefObject</code> <code>encodeBycopyObject</code>	Encodes anObject (id) as a copy of itself or as a reference to itself.
<code>encodeConditionalObject</code>	Encodes a reference to anObject (id) if it is encoded by the coder. If anObject has not been unconditionally encoded (for example, by encodeRootObject or encodeObject), anObject is not encoded.

Utility methods for NSCoder are shown in Table 12-4.

TABLE 12-4. NSCoder Utility Methods

<code>setObjectZone</code> <code>objectZone</code>	Accessors for the memory zone to be used by decoded objects. Set: zone (NSZone*). Get: returns an NSZone*.
<code>systemVersion</code>	Returns the system version as an unsigned int for the version in use when the object was archived.
<code>versionForClassName</code>	For className (NSString*), returns the version as an unsigned int for the class when it was archived.

Archiving

Archivers use `NSData` objects to store their data; those objects are then read to or written from disk. Both `NSArchiver` and `NSUnarchiver` are descendants of `NSCoder`; they override a number of methods there. You may never actually need to deal with `NSArchiver` objects: if you implement the `NSCoding` protocol and call the appropriate `NSCoder` methods to encode and decode your objects, it is likely that the framework and its standard objects will take care of everything for you.

NSArchiver `NSArchiver` is a subclass of `NSCoder` that deals specifically with writing objects to disk (using an `NSData` object and a file). Its methods are shown in Table 12-5; you rarely bother to call these methods unless you are starting the archive process by calling `encodeRootObject` or `archiveRootObject`.

TABLE 12-5. NSArchiver Methods

<code>initWithWritingWithMutableData</code>	Initializes an <code>NSArchiver</code> with <code>mdata</code> (<code>NSMutableData*</code>) as the data object to receive the data.
<code>archivedDataWithRootObject</code>	Creates an <code>NSArchiver</code> with <code>rootObject</code> (<code>id</code>) encoded; returns an <code>NSData*</code> , which is the archiver's data structure.
<code>archiveRootObject</code>	Instead of just returning an <code>NSData*</code> object, this method encodes <code>rootObject</code> (<code>id</code>) as the archiver's root object and writes it to <code>path</code> (<code>NSString*</code>) on disk.

Note that the process of encoding the root object may entail encoding an entire graph of related objects; this may be the only call you need to make.

TABLE 12-5. NSCoder Methods (Continued)

<code>encodeArrayOfObjCType</code>	This override of the <code>NSCoder</code> class takes the same arguments: <code>type</code> (<code>const char*</code>), <code>count</code> (<code>unsigned int</code>), and <code>array</code> (<code>const void*</code>) and encodes array.
<code>encodeConditionalObject</code>	This override of the <code>NSCoder</code> method encodes <code>object</code> (<code>id</code>) as a reference back to the archiver's root object (which must already have been encoded). <code>nil</code> is a valid argument for <code>object</code> .
<code>encodeRootObject</code>	Overrides <code>NSCoder</code> 's default implementation for <code>rootObject</code> (<code>id</code>). It is an error to encode a single object as a root twice within one archiver. Note that the process of encoding the root object may entail encoding an entire graph of related objects; this may be the only call you need to make.
<code>archiverData</code>	Returns the <code>NSData</code> object that this archiver is using.
<code>encodeClassName</code>	During encoding you can substitute a class (such as an abstract superclass) for another. Use this method to specify <code>trueName</code> (<code>NSString*</code>) and the name into which it should be encoded— <code>inArchiveName</code> (<code>NSString*</code>).
<code>†classNameEncodedForTrueClassName</code>	The reverse of <code>encodeClassName</code> , this method returns the <code>NSString*</code> that has been specified for <code>trueName</code> (<code>NSString*</code>).

NSUnarchiver The methods of NSUnarchiver (Table 12-6) mirror those of NSArchiver, as does its function. Also, just as with NSArchiver, you may never use an NSUnarchiver or call its methods directly: tend to your objects' data and implement `encodeWithCoder` and `initWithCoder`.

TABLE 12-6. NSUnarchiver Methods

<code>initWithReadingWithData</code>	Initializes an NSUnarchiver using <code>data</code> (NSData*) as its data object; returns the id of the NSUnarchiver.
<code>+unarchiveObjectWithData</code>	From <code>data</code> (NSData*), this method returns the id of an unarchived object.
<code>+unarchiveObjectWithFile</code>	From an NSData object located in the file at <code>path</code> (NSString*), returns the id of an unarchived object.
<code>decodeArrayOfObjCType</code>	The reverse of <code>encodeArrayOfObjCType</code> , this method takes the same arguments: <code>type</code> (const char*), <code>count</code> (unsigned int), and <code>array</code> (const void*) and encodes array.
<code>+classNameDecodedForArchiveClassName</code>	This method lets you find out the name under which a class was encoded. The argument <code>nameInArchive</code> (NSString*) is the class name; the method returns the name specified by <code>encodeClassName</code> in the <code>inArchiveName</code> argument.

TABLE 12-6. NSUnarchiver Methods (Continued)

<code>+decodeClassName</code>	Just as you could substitute class names during archiving, you can do so during unarchiving. This method substitutes <code>nameInArchive (NSString*)</code> with <code>trueName (NSString*)</code> during unarchiving.
<code>isAtEnd</code>	Returns a <code>BOOL</code> indicating if the end of the data stream has been reached.
<code>setObjectZone</code> <code>objectZone</code>	Accessors for the memory allocation zone to be used by the <code>NSUnarchiver</code> . Set: <code>zone (NSZone*)</code> . Get: returns an <code>NSZone*</code> .
<code>systemVersion</code>	Returns the data's system version as an unsigned int.

Distribution

Whereas archiving and unarchiving deal with moving objects to and from files, distribution deals with moving objects across address spaces. Again, you normally don't need to worry about creating these objects or even using them directly: the `NSCoder` that your `encodeWithCoder` and `initWithCoder` methods receive as an argument will happen to be an `NSCoder` that deals with distribution rather than with archiving.

Distribution relies on ports and connections between ports that are established with the help of the operating system. (For more on ports and connections, see "Tasks" on page 367.)

The primary class is `NSPortCoder`, a descendant of `NSCoder` that is used for distribution. Its methods are shown in Table 12-7; you might override `NSPortCoder` if you want to substi-

tute objects when they are transferred to or from your address space.

TABLE 12-7. NSPortCoder Methods

<code>connection</code>	Returns the <code>NSConnection*</code> object for this port coder.
<code>encodePortObject</code>	Encodes aPort (<code>NSPort*</code>). Use this only to encode an <code>NSPort</code> (not a regular object). In fact, you need never touch this (call it or override it): the <code>NSPortCoder</code> 's <code>encodeWithCoder</code> method calls this routine automatically.
<code>decodePortObject</code>	Called automatically by an <code>NSPortCoder</code> 's <code>initWithCoder</code> method, this method handles the appropriate negotiations with the Core OS to reestablish a port. You need never call or override it (unless you are modifying the microkernel...).
<code>isByCopy</code>	Returns a <code>BOOL</code> indicating whether it is a copy (YES) or a proxy (NO).

Controls

Other aspects of sharing in OpenStep include control structures to handle notifications as well as the current selection. (While selections are often used within a single application, they are the key to sharing information via the pasteboard.)

Notification

As objects within a single application (or several applications) interact, the ability for them to notify one another of specific events happening to specific objects makes for efficiencies both in coding and in running applications. For example, many interface elements reflect the states of the system and other applications (a font menu that contains the currently installed fonts is one such case). Without a notification structure, whenever you update your interface (redisplay your menus, redraw a panel, etc.) you need to go out and find what the current state is: have fonts been installed or uninstalled, has another application started or stopped, etc.

Using notifications, you can check the state of the system and other applications when you first present your user interface and register an interest in changes to those states. Then, instead of checking the states each time you re-present your interface, you can happily continue to display your initial states until notified otherwise.

Of course, notifications can be used for many other purposes than just keeping your interface up to date; the example given here just happens to be a common case.

Performance and Development Issues Objects can communicate with one another in various ways:

- An object can obtain a reference to another object and send messages to it, invoking its methods. This requires that the first object have a reference to the second and know what methods it has.
- A less tightly coupled structure lets the first object send messages to the second via an intermediary. The intermediary object (often a delegate of the first object) needs to know what messages the first object might send and what messages to pass on to the second object. The intermediary can function as a translator, converting one API to another.

- The least tightly coupled structure has no links between the objects. The first object posts notifications about possibly interesting events that it has processed; the second object responds to those notifications for those objects in which it is interested.

This last structure is the most sophisticated; it allows the greatest degree of independence among objects and therefore the maximum amount of object reuse and encapsulation (both of data and of functionality). Theoretically, you can construct a system in which objects know nothing about one another but simply post and respond to the appropriate notifications.

Be aware that this structure—which may allow significant savings in development time—may exert a toll at run time. In practice, the objects must know something about one another; if they don't, you are likely to wind up with a glut of notifications (most of which are not acted on). Even worse, objects that expect to receive notifications may not act on them because the notifications are not what they expected to receive.

A further difficulty with implementing notification-based architectures lies in the fact that many programmers remain uncomfortable with the idea. They simply don't trust the system to do the right thing: they are much more comfortable with using the fifty-year-old paradigm of procedure calls. However, to move into a robust, sophisticated, multiuser model (even the limited version of it that consists of multiple applications running simultaneously on a single computer), it is necessary to trust the framework that you use. The notification design, when properly used, can extend the economies of object-oriented programming as individual programmers become more and more expert in their own objects' functionalities and need to know less and less about other objects. Emotionally it may feel better to control all of your application's objects and their methods; unfortunately, that is a dead end, leading to closed systems, unmaintainable code, and to compromises with object-oriented design that will produce code that is just as spaghetti-like as the worst legacy Cobol

code (it just will take longer to reach that point because the object-oriented technologies will sustain you longer than Cobol did).

NSNotification An NSNotification object is the heart of the notification system. Although you can subclass NSNotification, doing so is likely to defeat the elegant simplicity of the design. A notification object has three fields:

1. The notification's name is an NSString that you select. Senders and receivers of notifications should agree on this string; it is usually much easier for development teams to agree on a string name for a notification than for a data structure or other shared information.
2. A notification contains a reference to an object. This is usually the object that posts the notification (that is, the object that causes the action that the notification reflects). This object may be nil; it also may be another object, but that design starts to muddy the clear design of the notification system.
3. The notification may also contain a dictionary that contains any additional information that the notification might need. For example, if several fields of a view are involved in a notification, the dictionary may contain keys such as "Field 1 view" and "Field 2 view" that are linked to specific objects. The dictionary is optional.

The flexibility of the notification lies in its simplicity. The dictionary (a very common OpenStep structure) lets you pass specific information in a common format along with the notification. (If you are familiar with Apple events, you will notice that this structure is quite reminiscent of the Apple event structure in which the common syntax allows an Apple event to contain very specific descriptors of meaning to its senders and recipients but which are transparent—but transmittable—to others.)

You can subclass `NSNotification`, but doing so immediately makes your events less general. If you need to pass additional information, pass it in the dictionary included with the notification. Don't subclass `NSNotification`.

Once you have decided that you have something to notify other objects about, you post the notification to the default notification center for your task. The notification center contains utilities for creating `NSNotification` objects. Table 12-8 lists the `NSNotification` methods: you usually create them via `NSNotificationCenter`s, but you may need the accessors in this table.

TABLE 12-8. `NSNotification` Methods

<code>notificationWithName</code>	Creates and returns a notification object (id) based on <code>aName</code> (<code>NSString*</code>) and <code>anObject</code> (id). The optional third argument, <code>userInfo</code> (<code>NSDictionary*</code>), can be used to pass additional information.
<code>name</code>	Returns the notification object's name as an <code>NSString*</code> .
<code>object</code>	Returns the notification object (often the originator of the notification) as an id. It may be nil.
<code>userInfo</code>	Returns the dictionary (if any) for the notification object as an <code>NSDictionary*</code> .

`NSNotificationCenter` The notification center for each task serves as the clearinghouse for notifications within that task. Notifications can be created by any object in the task and posted to the notification center. Independently, any object can register itself as an observer of notifications. As noted in the previous section, a notification has a name and optionally has

an object (usually the originator of the notification) and user information in a dictionary.

Observers can register to receive notifications based on name or originating object—or both, or neither. If the registration omits both the name and object, all notifications within the task are forwarded to it; if either or both are provided, only those notifications from the chosen object or with the given name are forwarded. When registering, an observer passes in a selector, which is a reference to one of its methods that will be invoked by the notification center. If a notification needs to be forwarded to an observer, that observer's method will be called and the `NSNotification` object passed as the single argument to it. You may name this method anything you want (but something like `ProcessNotification` wouldn't be such a bad idea). The `NSNotification` center makes very few demands on either notifiers or observers; the architecture is simple and robust.

Table 12-9 shows the `NSNotificationCenter` methods.

TABLE 12-9. `NSNotificationCenter` Methods

<code>+ defaultCenter</code>	Returns the default notification center for the task as an <code>NSNotificationCenter*</code> .
<code>+ addObserver</code>	Call this method to register an <code>Observer</code> (<code>id</code>); if notification needs to be forwarded, the observer's <code>aSelector</code> (<code>SEL</code>) method will be called, passing in the <code>NSNotification</code> . You may select notifications based on the <code>notificationName</code> (<code>NSString*</code>) or <code>anObject</code> (<code>id</code>) which is the originator of the notification. Either of these last two arguments may be <code>nil</code> .

TABLE 12-9. NSNotificationCenter Methods (Continued)

removeObserver	This is the complement to addObserver. Call it to remove anObserver (id) from the notification center.
	Using two additional arguments, you may remove it only for notificationName (NSString*) and anObject (id) which is the originator of the notification that you specified in addObserver.
removeObserver	notificationName object
postNotification	Use this method to post aNotification (NSNotification*).
postNotificationName	Use this method to create and post a notification of notificationName (NSString*) from anObject (id).

A notification center uses an **NSNotificationQueue** to manage the notifications; the notification queue normally functions as a first in–first out queue; however, the default **NSNotificationQueue** provides the ability to merge similar notifications (“coalescing”) and to remove events matching some criteria (name, object, etc.). Changing the way in which notifications are enqueued or dequeued can lead to problems later, so unless you are absolutely certain that you know what you are doing, stay out of the **NSNotificationQueue**. You are likely to optimize the performance of your application in a single-user, single-processor, nonnetworked environment... and lay the grounds for a disaster later on. Assume that the notification center uses a first in–first out queue and furthermore assume that it may function asynchronously (even though it does not in the simplest case).

Selection

Selection in a graphical user interface has two roles to play: it is used to identify data and information structure on which to perform actions (as in selecting text and then changing its font), and it is used as the first step in sharing data via the pasteboard. As with notifications, selections may need to be processed by objects that don't necessarily understand their content-specific structures. Just as the `NSNotification` object contains a dictionary (`userInfo`) that is capable of carrying information in a generalized format to and through objects that may not know about it, the `NSSelection` object contains an `NSData` object that wraps the content-specific selection information and that can be passed along by objects with no knowledge of the internals of the `NSData`.

Furthermore, although selections would appear to be highly content specific, many uses of selections are quite generalized. OpenStep declares three "well-known" selections:

1. `all`—everything in a document is selected
2. `empty`—nothing in a document is selected
3. `current`—the current selection (if other than all)

Selection objects are based on documents—a largely inferred part of the basic OpenStep structure. OpenStep deals largely with windows and views, and in its pre-Rhapsody architecture does not even include an `NSDocument`. By basing selections on a document, there is an assumption that multiple views of a single document cannot have individual selections.

NSSelection The methods of `NSSelection` are presented in Table 12-10. As with `NSNotification`, there is little reason to override the `NSSelection` object: put your application-specific information in the `NSSelection`'s `NSData` object.

TABLE 12-10. NSSelection Methods

<code>+selectionWithDescriptionData</code>	Creates and returns an <code>NSSelection*</code> based on <code>newData</code> (<code>NSData*</code>).
<code>initWithDescriptionData</code>	Initializes an <code>NSSelection*</code> based on <code>description</code> (<code>NSData*</code>) and returns it as an id.
<code>+allSelection</code> <code>+currentSelection</code> <code>+emptySelection</code>	These methods return shared instances of <code>NSSelection*</code> . They are not document specific.
<code>descriptionData</code>	Returns the <code>NSSelection</code> 's <code>NSData*</code> .
<code>isWellKnownASelection</code>	Returns a <code>BOOL</code> indicating if this selection is one of the three "well-known" selections.
<code>initWithPasteboard</code>	Initializes an <code>NSSelection*</code> based on the data on a <code>pasteboard</code> (<code>NSPasteboard*</code>). The data got there with <code>writeToPasteboard</code> (next).
<code>writeToPasteboard</code>	Writes the <code>NSSelection</code> to <code>pasteboard</code> (<code>NSPasteboard*</code>), whence a copy can be obtained by this task—or another—calling <code>initWithPasteboard</code> (previous).

Data Types

The cases of sharing discussed in this chapter so far have all been more or less mutual—data that is read and written, notifications that are posted and observed, selections that are read from and written to the pasteboard. Data linking is another case of sharing—but it is a case where one of the partners is dominant. The data link mechanism allows you (or a user) to

create a link from a selection in a source document to a selection in a destination document. The data link manager makes certain that those links are maintained and updated as necessary.

You have a number of tools that you can use to synchronize documents and windows within your application (and across several applications). As discussed in “Notification” on page 330, you can form tight links between two objects with strong mutual awareness as well as the very loose links among objects that communicate only through high-level notifications.

You can use data linking as yet another arrow in your synchronization and sharing quiver, relying on the data link manager to keep everything up to date. More often, though, data linking is a user-level feature; users decide if and when to use it. (The lack of enthusiasm for other incarnations of this functionality—publish/subscribe on Mac OS, parts of OLE/COM on Windows—is not necessarily an indication that people don’t want to do this; it very well may be an indication that the task as presented to users by developers remains too esoteric and computery.)

Data Links

The data link architecture may be familiar to you if you have used other mechanisms that provide for live updating of documents. Three primary objects implement the architecture:

1. `NSDataLink` is the data link object itself. A link may be complete (it has a source and destination) or it may be broken. There are an `NSDataLink` object for the source document and an `NSDataLink` object (possibly in another application) for the destination.
2. `NSDataLinkManager` is an object that must be instantiated for any document that supports linking.
3. `NSDataLinkPanel` is the panel that is used to control links (breaking, updating frequency, etc.).

The standard Paste and Link command is the user's entry into data linking. Note that while there are three objects and quite a few methods involved in supporting data linking, you rarely subclass these objects; you call relatively few of their methods. Linking is an easily-implemented feature of the OpenStep framework; the fact that its API is fairly hefty reflects only on its power and default functionality—on the amount of work that it does for you, and the relatively small amount of work that you have to do to implement data linking.

NSDataLink An `NSDataLink` (from either the source or destination document) is based on an `NSSelection` object. The user first selects a part of a document as a source and then selects a part of that or another document as the destination; at that point, the Paste and Link command is enabled, and the user can create the link.

The `NSDataLink` class contains the code necessary to manage links that are written to disk. Since data links are based on `NSSelection` objects, you can implement linking at a very high level: the document-specific aspects are encapsulated at the low level of the `NSData` object that is contained within the `NSSelection` objects for the source and destination.

Many of the methods shown in Table 12-11 are used by the `NSDataLinkManager` and the `NSDataLinkPanel`; you most likely will confine yourself to creating source links (`initWithSourceSelection`) and to notifying your source document that its links may have changed.

NSDataLinkManager A document that supports data linking must have a data link manager; data link managers communicate with one another to manage their various links. The implementation of data link managers relies on a delegate to do much of the document-specific processing. In this way, you can use the `NSDataLinkManager` class without subclassing it.

TABLE 12-11. NSDataLink Methods

<code>writeToPasteboard</code> <code>initWithPasteboard</code>	These methods write the link to or initialize if from pasteboard (NSPasteBoard*).
<code>initLinkedToSourceSelection</code>	This is the main routine that you use to create a source link. Its arguments are a description of the source data— <code>selection</code> (NSSelection*), <code>linkManager</code> (NSDataLinkManager*)—the source document's data link manager, and <code>newTypes</code> (NSArray*)—the types that the link manager can indirectly provide for the data in the link.
<code>initLinkedToFile</code>	Initializes and returns the id of a link encompassing <code>filename</code> (NSString*)—as opposed to a link stored in a file, as <code>initWithContentsOfFile</code> does.
<code>initWithContentsOfFile</code>	Returns the id of a link initialized from <code>filename</code> (NSString*). (See <code>saveLinkIn</code> and <code>writeToFile</code> .)
<code>saveLinkIn</code>	Returns a BOOL indicating success at setting <code>directoryName</code> (NSString*) for the NSDataLinkPanel to use in saving link files.
<code>writeToFile</code>	Returns a BOOL indicating success at saving the link in a file called <code>filename</code> (NSString*).
<code>openSource</code>	For a destination link, opens the appropriate source document; returns a BOOL indicating success.

TABLE 12-11. NSDataLink Methods (Continued)

<code>disposition</code>	Returns an <code>NSDataLinkDisposition</code> reflecting the link status. Possible values are <code>NSLinkInDestination</code> , <code>NSLinkInSource</code> , and <code>NSLinkBroken</code> .
<code>linkNumber</code>	Returns an <code>NSDataLinkNumber</code> identifying this link within the document. These numbers should not change and should be unique within each document.
<code>manager</code>	Returns the <code>NSDataLinkManager</code> ; broken links do not have an <code>NSDataLinkManager</code> and return <code>nil</code> for this method.
<code>lastUpdateTime</code>	Returns an <code>NSDate*</code> for the last update time.
<code>types</code>	Returns an <code>NSArray*</code> with the source document's types.
<code>break</code>	Breaks a link.
<code>noteSourceEdited</code>	Call this method of a source link to start the updating process (if necessary).
<code>updateDestination</code>	Updates the link; returns a <code>BOOL</code> indicating success.

TABLE 12-11. NSDataLink Methods (Continued)

setUpdateMode updateMode	A link's update mode is of type <code>NSDataLinkUpdate</code> mode and can have the following values: <code>NSUpdateContinuously</code> , <code>NSUpdateWhenSourceSaved</code> , <code>NSUpdateManually</code> , and <code>NSUpdateNever</code> . These are the accessors for the link. The <code>NSDataLinkPanel</code> lets users set these values. Set: <code>mode</code> (<code>NSDataLinkUpdateMode</code>). Get: returns <code>NSDataLinkUpdateMode</code> .
sourceApplicationName sourceFilename	Returns the source application or filename as an <code>NSString*</code> .
sourceSelection	Returns the source <code>NSSelection*</code> .
destinationApplicationName destinationFileName	Returns the destination application or filename as an <code>NSString*</code> .
destinationSelection	Returns the destination <code>NSSelection*</code> .

The methods of `NSDataLinkManager` are shown in Table 12-12. Remember that the source link and the destination link are two separate objects (possibly in two separate applications). When a link manager adds a link to a document (that is, converts a source link to a destination link), in fact it is not using the original object. The source link was copied to the pasteboard or a file; a new object was created and initialized from either the file or the pasteboard. It is this new object (which only indirectly is the source link object) that is used in the destination.

Each data link manager has a delegate; this delegate can be any type of object, but it must implement at least the methods

TABLE 12-12. NSDataLinkManager Methods

<code>initWithDelegate</code>	Initializes an <code>NSDataLinkManager</code> , setting its delegate to an object (<code>id</code>). The optional argument <code>path</code> (<code>NSString*</code>) specifies an existing file for the document.
<code>filename</code>	If a file was associated with the document (in <code>initWithDelegate</code>), this method returns it as an <code>NSString*</code> .
<code>addLink</code>	Adds <code>link</code> (<code>NSDataLink*</code>) at <code>selection</code> (<code>NSSelection*</code>) to the destination document. Note that this is a second link object constituted from the pasteboard or a file, not the source document's <code>NSDataLink</code> .
<code>writeLinksToPasteboard</code>	Writes all the document links to pasteboard (<code>NSPasteboard*</code>).
<code>addLinkPreviouslyAt</code>	Having written the links to the pasteboard with the previous method, you can copy or move an existing link with this method: pass it <code>oldSelection</code> (<code>NSSelection*</code>) for its current (old) location, <code>pasteboard</code> (<code>NSPasteboard*</code>) for its pasteboard, and <code>selection</code> (<code>NSSelection*</code>) for the new location.
<code>addLinkAsMarker</code>	Same as <code>addLink</code> , but this is for a link that contains no data (i.e., a reference back to another document or part of this document). Marker links are one way of implementing hypertext.

TABLE 12-12. NSDataLinkManager Methods (Continued)

<code>destinationLinkEnumerator</code> <code>sourceLinkEnumerator</code>	Return an <code>NSEnumerator*</code> for source or destination links managed by the link manager.
<code>destinationLinkWithSelection</code>	Returns the <code>NSDataLink*</code> associated with <code>destSel</code> (<code>NSSelection*</code>) in the document; may return <code>nil</code> .
<code>breakAllLinks</code>	Breaks each link in the link manager's document.
<code>noteDocumentClosed</code> <code>noteDocumentEdited</code> <code>noteDocumentReverted</code> <code>noteDocumentSaved</code>	These methods should be called by the application (or other controller object) to inform the link manager of these events.
<code>noteDocumentSavedAs</code> <code>noteDocumentSavedTo</code>	These methods should be called by the application (or other controller object) to inform the link manager of these events. They take a single argument <code>path</code> (<code>NSString*</code>) which is the file name to which the document has been saved (<code>as</code>).
<code>delegate</code>	Returns the link manager's delegate; should never return <code>nil</code> .
<code>setLinkOutlinesVisible</code> <code>areLinkOutlinesVisible</code>	Accessor for options controlling link display and user interaction.
<code>setInteractsWithUser</code> <code>interactsWithUser</code>	Set: flag (<code>BOOL</code>). Get: returns a <code>BOOL</code> .

TABLE 12-12. NSDataLinkManager Methods (Continued)

<code>setDelegateVerifiesLinks</code> <code>delegateVerifiesLinks</code>	Default data link manager functionality uses modification dates to synchronize links. In some cases (databases, remote systems, etc.), you may need to implement application-specific functionality in your delegate. Set: flag (BOOL). Get: returns a BOOL.
<code>isEdited</code>	Returns a BOOL indicating whether anything has happened since the last save; this relies on the <code>noteDocument...</code> methods.

shown in Table 12-13. This structure allows you to confine all of your document-specific processing to an object of your own choosing and to avoid subclassing `NSDataLinkManager`.

TABLE 12-13. NSDataLinkManager Delegate Methods

<code>dataLinkManager:</code> <code>didBreakLink</code>	This is a notification from sender (<code>NSDataLinkManager*</code>) that link (<code>NSDataLink*</code>) has been broken. Do whatever processing is necessary.
<code>dataLinkManager:</code> <code>isUpdateNeededForLink</code>	The delegate overrides this method to return a BOOL indicating to sender (<code>NSDataLinkManager*</code>) the update status of link (<code>NSDataLink*</code>).

TABLE 12-13. NSDataLinkManager Delegate Methods

<i>dataLinkManager TracksLinksIndividually</i>	You override this method and return a BOOL indicating if you are tracking links individually.
<i>dataLinkManager: startTrackingLink dataLinkManager: stopTrackingLink</i>	These methods are called by sender (NSDataLinkManager*) to inform you of the status of link (NSDataLink*).
<i>dataLinkManagerClose Document dataLinkManagerDid EditLinks dataLinkManagerRedraw LinkOutlines</i>	These methods are called by sender (NSDataLinkManager*) to provide you with information on what it is doing and what you may need to do.
<i>showSelection windowForSelection</i>	For selection (NSSelection*) in a link source, the delegate should show it on request, returning a BOOL indicating success; the second method should return the NSWindow* in which the selection is located.

TABLE 12-13. NSDataLinkManager Delegate Methods*copyToPasteboard*

After a source link has been created, the delegate is asked to actually move it to the pasteboard. Again, this is document-specific processing. The arguments are `pasteboard` (NSPasteboard*) and `selection` (NSSelection*). The final argument, `flag` (BOOL) indicates that you may not actually be writing anything to the pasteboard yet. See Table 12-16.

pasteFromPasteboard
importFile

These are the delegate's implementations of `addLink`; its arguments are `pasteboard` (NSPasteboard*) and `selection` (NSSelection*) or `filename` (NSString*) and `selection` (NSSelection*).

Here is where you do the document-specific processing (imaging, for example).

Modern syntax and Java syntax for some of the methods in Table 12-13 are not resolved; the methods shown here rely on Classic Objective-C's named arguments.

NSDataLinkPanel Users normally control data links using the standard data link panel. You should not create your own or override these methods. The NSDataLinkPanel methods are shown in Table 12-14.

TABLE 12-14. NSDataLinkPanel Methods

<code>†sharedDataLinkPanel</code>	Returns the <code>NSDataLinkPanel*</code> . Remember that like most system panels this may be shared, so don't muck around with its views! Use accessory views instead.
<code>setAccessoryView</code> <code>accessoryView</code>	Use an accessory view for document-specific controls. Set: <code>aView (NSView*)</code> . Get: returns an <code>NSView*</code> .
<code>†setLink</code> <code>†getLink</code>	Use these accessors to communicate with the <code>NSDataLinkPanel</code> ; you call <code>setLink</code> in response to document changes (new selection, for example) that would influence the data link panel's display. You call <code>getLink</code> to find out what the data link panel currently represents. The arguments are <code>link (NSDataLink*)</code> , <code>linkManager (NSDataLinkManager*)</code> , and <code>flag (BOOL)</code> —which indicates if multiple selections are allowed. These are both class and instance methods, and do the same thing.
<code>pickedBreakAllLinks</code> <code>pickedBreakLink</code> <code>pickedOpenSource</code> <code>pickedUpdateDestination</code> <code>pickedUpdateMode</code>	In response to user interaction with the data link panel, these messages are sent as needed; each takes the single argument <code>sender (id)</code> , which is the button that was clicked.

Display

Shared data is displayed in your documents and windows in whatever way is appropriate. The pasteboard, however, is a neutral middle ground that stores and sometimes displays data.

NSPasteBoard

The pasteboard is the simplest way to transfer data asynchronously between and among applications. Application A posts its data (in any of several formats) to the clipboard; application B then comes along, checks if any of the data on the clipboard is in formats that it recognizes, and if so enables its paste command.

Pasteboard (clipboards, for Mac OS devotees) have been around since the beginning of graphical user interfaces; they are an intrinsic part of the interface. OpenStep goes beyond the basics and provides five separate pasteboards:

1. `NSGeneralPboard` is the standard pasteboard in which text, graphics, etc. are stored.
2. `NSFontPboard` stores character styling information.
3. `NSRulerPboard` stores paragraph formatting information.
4. `NSFindPboard` allows you to cut, copy, and paste find criteria (such as search strings) without disturbing text that has been stored in a general pasteboard.
5. `NSDragPboard` stores drag-and-drop data.

This proliferation of pasteboards is one way of addressing the user interface problem in which a seemingly trivial action (such as copying a search string for use in a utility's search window) destroys a graphic that you carefully stored on the pasteboard—from an application that you have not quit.

This structure is extensible: you can create additional pasteboards with their own names. However, the benefit of pasteboards is their ability to let users easily share data between and among applications; idiosyncratic data formats as well as specialized pasteboards limit the reuse of the data.

The NSPasteboard methods are shown in Table 12-15.

TABLE 12-15. NSPasteboard Methods

<code>†pasteboardWithName</code> <code>†generalPasteboard</code>	Returns the pasteboard with <code>name</code> (NSString*). For NSGeneralIP-board, you can use <code>generalPasteBoard</code> , which takes no arguments. Both methods return an NSPasteboard*.
<code>†typesFilterableTo</code>	Normally filtering services are not invoked until you retrieve data from a pasteboard, this method returns an NSArray* listing all the types that <code>type</code> (NSString*) can be converted to given the installed filter services.
<code>availableTypeFromArray</code>	Given <code>types</code> (NSArray*), this routine returns an NSString* of the first type on the pasteboard that is found in the array.
<code>types</code>	Returns all the pasteboard's types in an NSArray*.
<code>†pasteboardWithUniqueName</code>	Creates and returns a pasteboard with a unique name; returns NSPasteboard*.

TABLE 12-15. NSPasteboard Methods (Continued)

<code>+pasteboardByFilteringData</code>	Creates and returns a pasteboard with a unique name, returning it as an <code>NSPasteboard*</code> . The pasteboard contains <code>data</code> (<code>NSData*</code>) of type (<code>NSString*</code>).
<code>+pasteboardByFilteringFile</code>	Creates and returns a pasteboard with a unique name, returning it as an <code>NSPasteboard*</code> . The pasteboard contains <code>filename</code> (<code>NSString*</code>).
<code>+pasteboardByFilteringTypesInPasteboard</code>	Creates and returns a pasteboard with a unique name, returning it as an <code>NSPasteboard*</code> . The pasteboard contains data from pasteboard (<code>NSPasteboard*</code>).
<code>declareTypes</code>	Sets the pasteboard to show that it contains <code>newTypes</code> (<code>NSArray*</code>) of data owned by <code>owner</code> (<code>id</code>). It returns an <code>int</code> which is the new <code>changeCount</code> . Follow a call to this method by calling <code>setData</code> .
<code>addTypes</code>	After <code>declareTypes</code> , you can call <code>addTypes</code> with <code>newTypes</code> (<code>NSArray*</code>) and <code>newOwner</code> (<code>id</code>). You get a new <code>changeCount</code> returned to you; instead of clearing the pasteboard, its ownership is transferred to <code>newOwner</code> and the types are added.

TABLE 12-15. NSPasteboard Methods (Continued)

setData	After declareTypes (and possibly addTypes), call this method with data (NSData*) and dataType (NSString*) indicating its format. Returns a BOOL indicating success of the operation. If you do not call setData for each type that you have promised, the pasteboard may call your pasteboard method to ask you to send a promised type.
setPropertyList setString	Specialized versions of setData take as arguments a propertyList (id) or string (NSString*) and dataType (NSString*). Returns a BOOL indicating success.
changeCount	Returns an int which is the pasteboard changeCount. This is maintained across applications; it is updated automatically every time the pasteboard's data changes. Having checked the pasteboard (for example, to see if certain types are available), you store the changeCount; the next time you need to access the pasteboard, if changeCount is the same, you know that the same data is there.
dataForType	Given a dataType (NSString*), this returns an NSData* object from the pasteboard. Note that nil may be returned if the pasteboard has changed since the last query; it can also return nil (not an exception) if there is no response from the pasteboard server before a time-out.

TABLE 12-15. NSPasteboard Methods (Continued)

<code>propertyListForType</code> <code>stringForType</code>	These methods return a property list (NSArray*, NSDictionary*, NSString*, or NSData* object) as an id—or an NSString* for dataType (NSString*).
<code>writeFileContents</code>	Writes filename (NSString*) contents to the pasteboard as NSFileContentsPboardType (or a more specific type based on the file's extension, creator, and type) and returns a BOOL indicating success. The utility NSCreateFileContentsPboardType is used to convert file extensions, creators, and types to pasteboard types.
<code>readFileContentsType</code>	Writes the pasteboard's contents of type (NSString*) to filename (NSString*). The filename used for writing is returned as an NSString*. The type here should match the type in writeFileContents.
<code>releaseGlobally</code>	Use this to release a private pasteboard; never release a standard pasteboard at all.
<code>name</code>	Returns the name of the pasteboard as an NSString*.

There is one case where specialized pasteboards may be quite useful: OpenStep provides routines that create pasteboards with guaranteed unique names. These pasteboards can be created empty or with the contents of other pasteboards, files, or NSData objects. You can then pass the name of such a pasteboard to another application (in a method call, via the dictionary of a notification object, or in a message) and it can

retrieve the information. This could save reading a file twice in two separate applications.

Data stored in the pasteboard can be stored in multiple formats (but it should always be the same data). When retrieved, filtering services can be invoked to convert data from one format to another. The `pasteboardByFiltering...` methods do not actually invoke the filtering services until data is removed from the pasteboard, so they have relatively little overhead.

Any object that owns a pasteboard at any time (by calling `declareTypes` and saying that it will place data in the pasteboard) must implement the two methods shown in Table 12-16.

TABLE 12-16. Pasteboard Owner's Methods

<i>pasteboardChangedOwner</i>	This method is called when <code>sender</code> (<code>NSPasteboard*</code>) has a new owner. The previous owner receives the message. You qualify if you have claimed the pasteboard with a call to <code>declareTypes</code> .
<i>pasteboard</i>	If you have not placed all the promised data types on the pasteboard, you may be asked to provide the data; your implementation of this method will be called by <code>sender</code> (<code>NSPasteboard*</code>) requesting that you place the data of <code>type</code> (<code>NSString*</code>) on that pasteboard. Implementing this method (and not calling <code>setData</code> for all data types) may provide significant run-time efficiencies.

There is rarely a need to subclass the `NSPasteboard` class. When an object places data in the pasteboard, it owns that pasteboard—until someone else claims it by placing data in it.

Programming Tools

Among the most basic programming tools needed to implement sharing is some ability to copy objects. Copying objects (cloning them, in some terminologies) is not quite so simple as you might imagine. Objects often contain instance variables that refer to other objects or to complex data structures. A copy of an object needs to handle these embedded objects appropriately; often they need to be copied themselves and references to the new objects stored in the copied object.

There is a distinction between shallow copying (cloning) and deep copying (cloning). A shallow copy is simply a copy of the object's bit structure in memory; references to objects in both the copy and the original point to the same objects. Deep copying makes copies of embedded objects where necessary. A shallow copy's references point back to the same objects as the original does, so a change to either the original or the copy has the same effect on the embedded objects. The `NSMutableCopying` protocol produces such shallow copies; `NSCopying` produces deep copies.

In some circumstances, you may decide that the single `NSCopying` protocol is appropriate to use and that it should produce shallow copies. Note also that the distinction between a shallow and a deep copy is clear conceptually, but may not be so hard and fast in reality. Nothing prevents a copy having some fields that are deep copies and others that are shallow copies. If this is the result of carelessness, it's obviously wrong; however, if the situation justifies it (and it is documented) there is nothing wrong with it.

Copying

Copying is implemented with protocols—`NSCopying` and `NSMutableCopying`. Objects are created by both protocols and may (or may not) be modified by others. These new objects are not autorelease objects: you must manage their disposal yourself.

Copying needs to be implemented based on your object's data structures. A reference to a font, a string, or a dictionary may well not need to be copied; references to subviews of a view usually do need to be reinstantiated and copied.

The challenge is actually not particularly daunting if your application is well structured; call the superclass's method first to allow it to do its own work. You probably will have relatively few instance variables that need to be copied.

On a case-by-case basis you will need to decide what to do about copying state variables. For example, is a copy of a view to have the same location and selection as the original? Should its location be slightly offset? Should its location be as it would be for a newly created view and should its selection be nil? These are your decisions to make.

NSCopying [P] The `NSCopying` protocol has two methods (Table 12-17). You call `copy`, which in turn calls `copyWithZone`. You override `copyWithZone`.

NSMutableCopying [P] If you need a mutable copy, you call the methods of the `NSMutableCopying` protocol to create mutable copies of objects, as shown in Table 12-18.

TABLE 12-17. NSCopying [P] Methods

<code>copy</code>	This is the method that you call to get a copy; if you want the copy to be allocated in a specific zone (other than the current zone) you must override it to specify the zone.
<code>copyWithZone</code>	Returns a new object allocated in zone (NSZone*). You override this method to do whatever re-instantiating is necessary.

TABLE 12-18. NSMutableCopying [P] Methods

<code>mutableCopy</code>	This is the method that you call to get a copy; if you want the copy to be allocated in a specific zone (other than the current zone) you must override it to specify the zone.
<code>mutableCopyWithZone</code>	Returns a new object allocated in zone (NSZone*). You override this method to do whatever re-instantiating is necessary.

Synchronism

You may never have given much thought to synchronism—the processing of several chores at once. Synchronism didn't exist on the early personal computers, and even today it is fairly rare. Multiprocessing, multitasking, and multithreading are all aspects of synchronism: they allow several things to go on at the same time. When a computer has more than one processor and has network connections and peripherals

that have their own processors, it makes little sense to hold everything hostage to a model that handles one thing at a time.

Kinds of Synchronism

Synchronism has four principal aspects:

- a hardware environment in which several things can be done at once
- an operating system that is designed to manage multiple processes
- application software that takes advantage of the hardware and operating system
- perceptual synchronism which refers to the structuring of an application's interface so that although nothing actually moves faster, it appears to do so for the user. (The display of progressive GIFs on Web pages is one such example.)

Rhapsody meets the first two conditions (Yellow Box on Mac OS or Intel meets the second and provides more limited synchronism). Your concern is with the third condition: software that takes advantage of all of this. Old-time programmers who wrestled with mainframes know the enormous performance improvements that you may be able to achieve when you use synchronism properly.

Hardware Synchronism Hardware-based synchronism takes advantage of the fact that a computer (even with a single processor) is often waiting for disk reads to complete, printing jobs to finish, network connections to work, etc. Having pieces of hardware that can work simultaneously is a necessary precondition for implementing any kind of synchronism. If you have only a single processor and no peripherals, nothing can be done on that computer that is not done by the processor: synchronism is impossible.

Today's computers, of course, present multiple opportunities for synchronism. Multiple processors within a single comput-

er (not to mention networked computers that can share processing chores) allow a high level of synchronism; peripherals with their own processors and buffers (such as modern printers) allow the main processor not to have to wait for slower devices.

Unless you are writing a very time-critical application that is intimately tied to a specific hardware configuration, as an application developer you are unlikely to be closely involved with hardware synchronism. However, the strategies that you take in implementing application synchronism rely on there being some form of hardware synchronism (you just don't know what specifically exists).

Core OS Synchronism Rhapsody's core OS (based on the Mach microkernel) implements as much synchronism as possible. Early tests with prototypes of the Blue Box (Mac OS) suggested that some applications actually ran faster on the same hardware when they ran in the Blue Box on the core OS than when they ran on the Mac OS. The reason for this is the microkernel's ability to schedule tasks—particularly input and output tasks—with a high degree of efficiency.

Application Synchronism You come in at this level. By doing nothing in your application, you can take advantage of the Rhapsody microkernel's efficiencies. However, a range of opportunities presents itself to you if you want to make your application as ready as possible to take advantage of these modern features.

Perceptual Synchronism There are many tricks to use to increase the perceived responsiveness of an application. One of the most common is embodied in progressive GIFs: start the response to a command as soon as possible, rather than waiting until all (invisible) processing is complete. Another technique—proven in much research—is to provide the user with the ability to control (and terminate) actions that take time. A user who feels in control of a process will swear that it takes less time than a process that cannot be controlled.

Opportunities for Synchronism

Here are some strategies for taking advantage of synchronism.

Take Advantage of Hardware and OS Synchronism Possibilities You can increase your application's throughput by interleaving communications calls with your application's code, rather than doing all of your processing and then transmitting everything. This means that your total elapsed time of using the network is longer than if you had waited to handle all of your communications at once, but as long as the protocol and networking software allows sharing, you don't degrade anyone else's performance. And since at least a certain amount of the network processing is dependent on remote events, your computer's processor is likely to be waiting for a significant amount of time during communications processing. This strategy is available to you on many operating systems, not just Rhapsody.

Use Multiple Threads Since Rhapsody provides support for threading, you can create multiple threads within your application. If each thread does standard (or at least similar) processing involving computation, memory accesses, disk accesses, etc., chances are that each thread will periodically wait for processor, memory, or disk. By having several threads, you increase the likelihood that one thread's wait state coincides with another thread's performance. (You can also give threads separate priorities so that a less important thread chugs along slowly but surely while your higher priority main thread does its critical processing as quickly as possible.)

Threads all share your task's globals; if you are going to be using threads, you will need to consider using locks to protect these globals from corruption (see "Locking" on page 361). Because they share your task's globals, the task cannot terminate until its last thread terminates.

Use Multiple Tasks You can spin off an additional task to do processing that is more independent than that of a thread.

Each task has its own memory space and its own globals: you don't have to worry about locking shared resources, but you also don't have the advantage of easily being able to share data. Tasks can be fired off to perform processes that the originating task no longer cares about (imaging to a file, communications, etc.).

If you are planning a system for an environment that may grow in the future, tasks can also position you very well to work in a multiprocessing environment. Because tasks are independent, you can distribute them across a network of computers. Even if this is not your environment today, the architecture presents few problems in the present—and may provide significant opportunities in the future.

Whatever you do—with tasks or threads—you have to banish the notion (if you still have it) that a single application program is in charge of everything.

Locking

As soon as you move out of the single-processing model, you need to manage interactions among processes. In multiuser databases on mainframes this has been an issue for decades, and you can take advantage of that experience in designing your application.

The simplest case is illustrated by what happens when you see an unusual lamp in an antique store that you cannot live without but which you suspect (fear? hope?) will not work its magic on your spouse, partner, or family. If you run home and drag the appropriate people back to the antique store, the lamp may have been sold to another sucker. You can prevent this by asking the dealer to hold the lamp—but that puts the dealer in the rather tricky position of possibly turning down a sale just so that you can come back and say no.

A lock is basically a hold on all processing—except for that of one thread. In the lamp example, if you could convince the dealer to place a lock on the lamp, no one else could buy it un-

til you released your lock. (Databases use sophisticated locks, as do antique dealers: your agreement may be that the dealer can't sell the lamp, but that it can be shown without commitment to prospective buyers.)

Locks are whatever you say they are. Although locks are often associated with resources (printers, etc.) and with data (a customer record, for example), the OpenStep lock mechanism has no intrinsic meaning. You can lock a method, for example, which you might do if you know that a call to that method will make heavy demands on the computer's resources. Although you allow users to have many windows open at the same time and to work with many images, perhaps you will lock the method that does heavy image manipulation so that if someone decided to reimagine several images at the same time, the imaging code would be locked to prevent crippling the system.

Note that locks are implemented in the next higher system level from their callers. Thus, if two threads within a task use a lock to manage shared resources, the lock (and presumably the shared resources) are owned by the task. Similarly, if two tasks within the OS share a locked resource, the lock (and the resource) reside in the OS.

NSLock The heart of the locking mechanism is the `NSLock` object. You create lock objects fairly early in your application—certainly before you spin off threads. (If you don't spin off threads, you have no reason to create locks.)

Locks should be given meaningful names that reflect what you are going to be locking. There is relatively little harm in creating an extra lock or two: until you actually lock one of them nothing happens.

The `NSLocking` protocol provides basic lock and unlock methods. Using the `NSLocking` protocol's lock method can produce problems, however. The thread that issues the lock call goes to sleep until the lock is granted. In most processing

TABLE 12-19. NSLock Method**lockBeforeDate**

Returns a BOOL indicating if the lock has been acquired before `limit` (NSDate*). Typically, `limit` is the current time plus a reasonable amount of time that varies with your environment. The timeout for a lock involved with Internet activities could well be five minutes; the timeout for a lock involving processing of several threads on a single computer might be less than a minute.

environments, it is better to attempt to lock something—and to regain control if the lock is not successful. Table 12-19 shows the NSLock method that does this.

NSConditionLock A descendant of NSLock is the NSConditionLock. This is a more sophisticated lock: it has a condition value (an int) that you can set. You can request that the lock be granted if that condition has a certain value. By defining constants with meaningful (to you) values, you can add a great deal of sophistication to your locks.

You normally set the lock's condition value using `initWithCondition` or using `unlockWithCondition`. If the condition value changes dynamically (between initialization and unlocking), you can override the condition method. The entire infrastructure is there, but you must implement it using the methods shown in Table 12-20.

Beware of getting too sophisticated with your conditions if you are not experienced in this area. In particular, conditions that themselves involve other locks can bring you to grief. A thread that requests a lock is blocked from further execution

TABLE 12-20. NSConditionLock Methods

initWithCondition	Initializes an NSConditionLock using <code>condition</code> (int).
<code>condition</code>	Returns the lock's condition as an int.
lockBeforeDate	Returns a BOOL indicating if the lock has been obtained before <code>limit</code> (NSDate*). [Same as NSLock method.]
<code>lockWhenCondition</code>	This method attempts to acquire the lock; it is successful if the lock is available and if its condition is equal to <code>condition</code> (int). This method is similar to the NSLocking protocol method: it blocks the thread's execution until the lock is granted.
lockWhenCondition	This method returns a BOOL indicating if the lock has been obtained with a value equal to <code>condition</code> (int) before <code>limit</code> (NSDate*).
unlockWithCondition	This releases the lock and sets its condition value to <code>condition</code> (int). This is a useful way for you to let the lock store information that might otherwise require a database access.

until that lock is granted: you can easily get yourself locked out from everything.

In the cases where you do need to request multiple locks within a single thread, use the NSRecursiveLock.

NSRecursiveLock A thread is blocked when it requests a lock until that lock is granted (unless it uses the timeout feature in

lockBeforeDate). If methods within your thread might attempt to acquire a lock that your thread has already acquired, you might be in the position of nesting your locks—and the second lock will never be granted and your thread will be blocked from further execution because the first lock cannot be released. In this case, use the NSRecursiveLock rather than the NSLock. This brings a little more overhead with it, but it prevents you from accidentally locking yourself out. (These rules about locks apply to individual locks: if your thread acquires five separate locks—five separate instantiations of NSLock—you don't have to worry about locking yourself out.)

Table 12-21 shows the NSRecursiveLock method.

TABLE 12-21. NSRecursiveLock Method

lockBeforeDate	Returns a BOOL indicating if the lock has been obtained before <i>limit</i> (NSDate*).
-----------------------	--

NSLocking [P] Finally, the NSLocking protocol defines lock and unlock methods that you use to manipulate NSLock objects. As noted previously, the lock method of the protocol blocks your thread from further execution until the lock is granted. You are almost always better off to use the NSLock lockBeforeDate method.

For unlocking, however, the unlock method is used in all cases (except for when you need to set a condition for an NSConditionalLock).

The methods of the NSLocking protocol are shown in Table 12-22.

TABLE 12-22. NSLocking [P] Methods

<code>lock</code>	Acquires a lock and blocks the thread's processing until the lock is acquired.
<code>unlock</code>	Unlocks a lock.

Threading

Each task starts off with a single thread. If you want to create additional threads, you use the `NSThread` object. Threads share your task's global variables. If you are going to be creating locks for data or processing, create them in your task's first thread before creating additional threads.

The methods of `NSThread` are shown in Table 12-23.

TABLE 12-23. NSThread Methods

<code>currentThread</code>	Returns an <code>NSThread*</code> which is the current thread.
<code>isMultiThreaded</code>	Returns a <code>BOOL</code> indicating if the task is now or ever has been multi-threaded.
<code>exit</code>	Terminates the thread. You normally do not terminate your own thread. You may terminate another thread either to abort its processing or because it is a background task that is no longer needed.

TABLE 12-23. NSThread Methods (Continued)

<code>detachNewThreadSelector</code>	This is the method you use to create another thread. Its arguments are <code>aSelector</code> (SEL), which is a method or message of <code>aTarget</code> (id), and which is invoked with a single argument <code>anArgument</code> (id).
<code>threadDictionary</code>	Returns an <code>NSMutableDictionary*</code> for the thread.

Tasks

Tasks have more independence than threads: they live in their own address space and function independently. You don't use locks if you are sharing resources among tasks: locks need to be in a common address space. A task runs in its own environment: a directory; standard input, output, and error files; and other implementation-specific variables. When you create a task, it inherits your own task's environment unless you explicitly change its environment in whole or in part. This must be done before the task is started; thereafter, it is responsible for making any changes in its environment.

The methods of `NSTask` are shown in Table 12-24.

TABLE 12-24. NSTask Methods

<code>†launchedTaskWithLaunchPath</code>	This method returns the <code>NSTask*</code> that is launched from the code at <code>path</code> (<code>NSString*</code>) using <code>arguments</code> (<code>NSArray*</code>).
<code>init</code>	This initializes an <code>NSTask</code> using the current task's environment. If you use <code>init</code> , you can use the accessors to reset the task's environment; when you are done, you can call its <code>launch</code> method.
<code>launch</code>	This method launches a task (usually one that you have used <code>init</code> and some accessors to customize; otherwise you can use <code>launchedTaskWithLaunchPath</code>).
<code>terminate</code>	Probably terminates the task. Tasks need not respond to the <code>terminate</code> method; if you need this degree of control, you may be better off with separate threads within your own task.
<code>isRunning</code>	Returns a <code>BOOL</code> indicating if the task is running. A <code>NO</code> may mean that the task hasn't started or that it has finished.
<code>waitUntilExit</code>	Suspends your task until the called task has exited. By suspending yourself, you may be mitigating the values of multiprocessing.
<code>terminationStatus</code>	Returns the task-defined termination code of the task as an <code>int</code> . Often, 0 means a successful termination.

TABLE 12-24. NSTask Methods

<code>setArguments</code> <code>arguments</code>	Accessors for the arguments set at launch; cannot be reset after launch. Set: <code>arguments</code> (NSArray*). Get: returns an NSArray*.
<code>setCurrentDirectoryPath</code> <code>currentDirectoryPath</code>	Accessors for the path for the task; cannot be reset after launch. Set: <code>path</code> (NSString*). Get: returns an NSString*.
<code>setLaunchPath</code> <code>launchPath</code>	Accessors for the path for the executable code of the task; cannot be reset after launch. Set: <code>path</code> (NSString*). Get: returns an NSString*.
<code>setEnvironment</code> <code>environment</code>	Accessors for the dictionary (user data) for the task; cannot be reset after launch. Set: <code>environmentDictionary</code> (NSDictionary*). Get: returns an NSDictionary.
<code>setStandardError</code> <code>standardError</code> <code>setStandardInput</code> <code>standardInput</code> <code>setStandardOutput</code> <code>standardOutput</code>	Accessors for the three standard files for the task; cannot be reset after launch. These can be NSFiles or NSPipes. Set: <code>file</code> (id). Get: returns an id.

Connections

The `NSConnection` (Table 12-25) lets you share objects between threads or tasks whether on the same machine or on different hosts.

A connection consists of two `NSConnection` objects, one in each thread that is communicating with the other. The connections each have a send and receive port; they mirror one another in the two threads (that is, the send port in the `NSConnection` of one thread is the receive port in the `NSConnection` of the other thread).

This is the implementation of the Mach messaging architecture described in “Communications Between and Among Tasks” on page 54. It is one of the most powerful features of Rhapsody. Sharing messages and data in a world of multiple processors and processes is not easily accomplished with traditional operating systems.

You can use `NSConnections` to share distributed objects between threads and to send messages back and forth. From your point of view, the connection’s root object is what is seen by others; the connection’s root proxy is what you see at the other end of the connection. For a real-life example of this, see “Using Distributed Objects” on page 449.

TABLE 12-25. `NSConnection` Methods

<code>+defaultConnection</code>	Returns an <code>NSConnection</code> with a single send/receive port.
<code>+connectionWithReceivePort</code>	Using <code>receivePort</code> (<code>NSPort*</code>) and <code>sendPort</code> (<code>NSPort*</code>) returns an <code>NSConnection*</code> .
<code>initWithReceivePort</code>	Initializes an <code>NSConnection</code> with <code>receivePort</code> (<code>NSPort*</code>) and <code>sendPort</code> (<code>NSPort*</code>).
<code>runInNewThread</code>	Creates and starts a new thread for the process, and runs the receiving connection in it.

TABLE 12-25. NSConnection Methods (Continued)

<code>registerName</code>	Names (or renames) the connection with <code>name</code> (<code>NSString*</code>). Returns a <code>BOOL</code> indicating success.
<code>setRootObject</code> <code>rootObject</code>	Accessors for the root object of the connection. Set: <code>anObject</code> (<code>id</code>). Get: returns an <code>id</code> .
<code>+connectionWithRegisteredName</code>	Returns an <code>NSConnection*</code> named <code>name</code> (<code>NSString*</code>) on <code>hostName</code> (<code>NSString*</code>). If <code>hostName</code> is <code>nil</code> , the local host is used.
<code>rootProxy</code>	Returns an <code>NSDistantObject*</code> .
<code>+rootProxyForConnectionWithRegisteredName</code>	Returns an <code>NSDistantObject*</code> for the connection <code>name</code> (<code>NSString*</code>) on <code>hostName</code> (<code>NSString*</code>).
<code>+allConnections</code>	Returns an <code>NSArray*</code> of all connections for the process.
<code>setRequestTimeout</code> <code>requestTimeout</code>	How long you have to respond to incoming messages over this connection. Set: <code>seconds</code> (<code>NSTimeInterval</code>). Get: returns an <code>NSTimeInterval</code> .
<code>setReplyTimeout</code> <code>replyTimeout</code>	How long you will wait for responses to messages sent (by you) over this connection. Set: <code>seconds</code> (<code>NSTimeInterval</code>). Get: returns an <code>NSTimeInterval</code> .

TABLE 12-25. NSConnection Methods (Continued)

<code>setIndependent- ConversationQueuing independent- ConversationQueuing</code>	These accessors control whether or not messages are handled atomically. Set: flag (BOOL). Get: returns a BOOL.
<code>addRequestMode removeRequestMode</code>	Adds or removes <i>mode</i> (NSString*) from the connection's run loop input modes.
<code>requestModes</code>	Returns an NSArray* of request modes for the receive port.
<code>invalidate</code>	Invalidates the NSConnection together with its proxy and exported object (if any).
<code>receivePort sendPort</code>	Returns the appropriate NSPort*.
<code>statistics</code>	Returns an NSDictionary* with debugging information for the connection.
<code>setDelegate delegate</code>	Set: anObject (id). Get: returns an id.

Summary

The shared environment of Rhapsody processing allows you to take full advantage of the modern OS and of sophisticated contemporary hardware with multiple processors and peripherals with their own processors and memory.

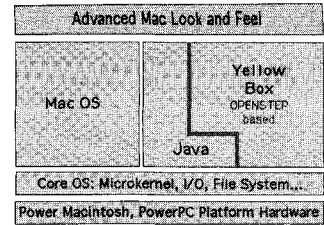
If you come from the hoary mainframe world, this is second nature to you; if you have lived all of your life on a personal computer, you may not realize the power of these features.

There is no doubt that the future will bring more processors, more connectivity—and more opportunities for applications that are prepared to live in this world.

Nothing requires you to use tasks and threads, but gaining a familiarity with them and using them where appropriate prepares you for the most sophisticated system designs of the future.

Chapter 13

Files and Documents



It is in the workhorse routine functions—file saving, printing, managing menus and mouse clicks—that a framework like OpenStep really shows its worth. In the world today there are some programmers who don't know what it is like to build a fully featured application from scratch. (Unfortunately, there are also a few who feel that if they don't reinvent their own particular wheels they're not earning their keep or proving their creativity.)

Behind the scenes, OpenStep does much of the work for you. In this chapter, you will find the routines that do that. Standard Open and Save panels let users select, open, and save files—you don't have to do anything to avail yourself of these services.

Beyond that, this chapter covers bundles and file wrappers—common objects that you may use and that you should know about.

Controls

OpenStep provides you and the user with default panels to select, open, and save files. You can override them—but why invent a new interface for you to maintain and for your users to learn?

Opening and Saving Files

The Open and Save panels are standard in the Rhapsody interface; as with other standard panels, you ask for one—and you get one. Of course, it may have been used, but it's returned to you with its default values intact. You may modify the values (and your users certainly will), but you shouldn't modify the panel itself: that is, don't add buttons and gratuitous (or any) graphics. If you do, the next person who gets that shared panel will get your handiwork. And probably not like it.

To accommodate application-specific needs, you can add an accessory view to the panel. The default panel has no accessory view, and whenever you get a panel (new or used) from the operating system it will have no accessory view. Put your graphics, special data entry needs, etc. into an accessory view and associate it with your panel.

While accessory views handily take care of specific data entry needs, you also often need to add functionality to the panel—a way of checking if a specific file really is going to be usable, for example. You can add a delegate object to the panel. If there is a delegate, `isValidFileName` is sent to the delegate at the last minute. Your delegate obviously should be able to respond to `isValidFileName`, and it can do any processing at all before returning a `BOOL` that will let the panel continue with its work or not.

On Windows, `NSSavePanel` is a direct descendant of `NSObject`. On Rhapsody, it is a descendant of `NSPanel` and thus `NSWindow`, `NSResponder`, and `NSObject`. On both platforms, `NSOpenPanel` is a descendant of `NSSavePanel`.

NSSavePanel The Save panel lets the user select a directory, enter a file name, and attempt to save a file. Your responsibility (through the delegate you attach to the panel) is to do any further checking; if you don't approve of the entries, your delegate's `isValidFileName` method returns `NO` and the panel stays up until the user either enters correct data or cancels out of it.

After this, it is your responsibility to actually save the file.

The `NSSavePanel`'s fabrication and accessor methods are shown in Table 13-1.

TABLE 13-1. NSSavePanel Fabrication and Accessor Methods

<code>+savePanel</code>	Creates and returns a Save panel with default settings.
<code>setAccessoryView</code> <code>accessoryView</code>	You add an accessory view to the Save panel to handle application-specific data fields. Set: <code>aView (NSView*)</code> . Get: returns an <code>NSView*</code> .

**TABLE 13-1. NSSavePanel Fabrication and Accessor Methods
(Continued)**

<code>setDelegate</code> <code>delegate</code>	<p>Use the delegate object to add functionality to the Save panel (as opposed to the data fields in an accessory view). The delegate will get the <code>isValidFileName</code> message and is expected to return a <code>BOOL</code>.</p> <p>Set: <code>anObject</code>. Get: returns an id.</p>
<code>setTitle</code> <code>title</code>	<p>This is the panel's title; the default is "Save," but you can change it as necessary.</p> <p>Set: <code>title</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code>.</p>
<code>setPrompt</code> <code>prompt</code>	<p>The default Save panel prompt is "Name:"; you can access it with these methods.</p> <p>Set: <code>prompt</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code>.</p>
<code>setDirectory</code> <code>directory</code>	<p>The default directory for the Save panel is the application's working directory. These accessors let you manage the directory shown in the panel. Note that when the panel is being shown modally, the directory is updated only at the end of the modal loop: <code>directory</code> will return incorrect results during (that is, before the end of) the modal loop.</p> <p>See <code>runModalForDirectory</code> for a shortcut to setting the directory explicitly.</p> <p>Set: <code>path</code> (<code>NSString*</code>). Get: returns an <code>NSString*</code>.</p>

**TABLE 13-1. NSSavePanel Fabrication and Accessor Methods
(Continued)**

<code>setRequiredFileType</code> <code>requiredFileType</code>	This limits the file to a certain type. Set: <code>type</code> (NSString*). Get: returns an NSString*.
<code>setTreatsFilePackagesAs</code> <code>Directories</code> <code>treatsFilePackagesAs</code> <code>Directories</code>	In addition to files and directories, Rhapsody supports the concept of file packages (as in applications that contain several files). Use this flag to determine whether such a package is treated as a file or as a directory. Set: <code>flag</code> (BOOL). Get: returns a BOOL.

In Table 13-2 you will find the methods that you use to actually control the Save panel.

TABLE 13-2. NSSavePanel Utility Methods

<code>runModal</code>	Starts the modal loop for the Save panel; it returns <code>NSOKButton</code> or <code>NSCancelButton</code> .
<code>runModalForDirectory</code>	Same as <code>runModal</code> , but you specify <code>path</code> (NSString*) and <code>fileName</code> (NSString*) as the initial values for the directory and file name.
<code>fileName</code>	Returns the file name entered by the user as an NSString*. You can use <code>directory</code> to get the chosen directory (see Table 13-1). Do not use either method until after the modal loop has terminated: until then, these values are undefined.

TABLE 13-2. NSSavePanel Utility Methods (Continued)

<code>ok</code> <code>cancel</code>	These methods, both of which take the single argument <code>sender</code> , are invoked after the corresponding button is clicked. You can override them to modify behavior.
<code>validateVisibleColumns</code>	This method asks the browser to update (if necessary) the visible files and directories it shows. Use it to cause an update after having changed parameters such as <code>requiredFileType</code> or <code>treatsFilePackagesAsDirectories</code> —anything that might make the browser’s display become out of date.
<code>selectText</code>	The <code>sender</code> is the object that is causing selected text to be changed (via mouse clicks or keyboard events like tabs). Internally used.

Delegates for NSSavePanel The delegate for an `NSSavePanel` has some specific requirements: it is used to filter files out of the list, for a final review, and on very rare occasions to reorder the files in the Save panel’s browser. These methods are shown in Table 13-3.

TABLE 13-3. NSSavePanel Delegate Methods

<code>panel:shouldShowFileName</code> <code>panel:isValidFileName</code>	<p>The sender (the panel) asks you to return a <code>BOOL</code> indicating whether <code>fileName</code> (<code>NSString*</code>) should be shown or is valid. <code>shouldShowFileName</code> is invoked as the panel is being drawn; <code>isValidFileName</code> is invoked as a last step after the user has made a choice.</p>
<code>panel:compareFileName:</code>	<p>On the extraordinarily rare occasions when you want to display files in an order other than the default order (which the user will see in every other browser on the system), you can use this method to resort the files. You'd sure better have a good reason to do this!</p> <p>Arguments are the sender (the panel), <code>fileName1</code> and <code>fileName2</code> (<code>NSString*</code>), and a <code>flag</code> (<code>BOOL</code>) indicating whether the comparison should be case sensitive or not. You return an <code>NSComparisonResult</code>.</p>

NSOpenPanel The theory behind `NSOpenPanel` is similar to that of `NSSavePanel`—the notion of reusing a shared panel, the use of a delegate, etc.

The methods of `NSOpenPanel` (other than those inherited from `NSSavePanel`) are shown in Table 13-4. You will note that the `runModal...` methods differ for `NSOpenPanel`: they add a `fileTypes` argument (an `NSArray*`) that you can use to screen the types of files that you want to be shown in the browser.

TABLE 13-4. NSOpenPanel Methods

<code>+openPanel</code>	Returns an Open panel.
<code>runModalForTypes</code>	Starts the modal loop for file-Types (NSArray*); returns NSOK-Button or NSCancelButton
<code>runModalForDirectory</code>	Same as <code>runModalForTypes</code> , but it takes three arguments: <code>path</code> (NSString*), <code>fileName</code> (NSString*), and <code>fileTypes</code> (NSArray*). <code>path</code> and <code>fileName</code> are used to seed the browser with a starting path and file name; either or both can be an empty string.
<code>filenames</code>	Returns an NSArray* with all the selected file names in it.
<code>setCanChooseFiles</code> <code>setCanChooseDirectories</code> <code>setAllowsMultipleSelection</code>	These accessors let you determine if files (default yes), directories (default no), and multiple selections (default no) are allowed.
<code>canChooseFiles</code> <code>canChooseDirectories</code> <code>allowsMultipleSelections</code>	Set: flag (BOOL). Get: returns a BOOL.

Programming Tools

Bundles and file wrappers are the most basic classes that help you manage files in general as well as bundles and text files.

NSBundle

The boundary line between the objects in frameworks and the outside, generally non-object-oriented world is always a tricky one. In Rhapsody, that boundary is pushed further and further into the operating system: bundles exist as directories in the file system as well as existing as objects in OpenStep. The `NSBundle` object deals with bundles intelligently—it

knows more than that they are files within a directory; it is sensitive to the special needs of resource files and to the handling of alternative language versions of the same file.

An `NSBundle` object lets you interact with the file system directories that are bundles; in practice, it lets you load classes from bundles other than your application's main bundle. Splitting the application into several bundles can make it more efficient and provide dynamic advantages.

The way in which you load classes dynamically from a bundle is simple.

1. You can create an `NSBundle` and set its path.
2. If you ask it for its principal class, all of its classes are loaded into memory at that moment.
3. Alternatively, you can ask it to return a class object using the `classNamed` method; it will return that class object (if it exists) and load all of its other classes into memory. Meanwhile, given the class object that you have just gotten, you can allocate an instance of it, initialize it, and proceed on your way.

Bundles also make it easy to localize software; they are sensitive to languages and can load alternative versions for different languages.

Table 13-5 lists the basic `NSBundle` methods.

TABLE 13-5. `NSBundle` Methods

`initWithPath`

Initializes an existing bundle with `fullPath` (`NSString*`).

TABLE 13-5. NSBundle Methods (Continued)

<code>†bundleForClass</code>	Returns an <code>NSBundle*</code> that contains <code>className</code> .
<code>†bundleWithPath</code>	Returns an <code>NSBundle*</code> with <code>path</code> (<code>NSString*</code>). It will create one if it doesn't already exist.
<code>†mainBundle</code>	Returns the application's main bundle.
<code>className</code>	Returns the <code>Class</code> object for <code>className</code> . The bundle is loaded into memory at this point.
<code>principalClass</code>	Returns the <code>Class</code> object for the first class in the bundle. Principal class should be (but need not be) the controlling or primary class in the bundle.
<code>†pathForResource</code>	Returns the <code>NSString*</code> for the path of the resource with name (<code>NSString*</code>) and extension (<code>NSString*</code>). The optional <code>directory</code> (<code>NSString*</code>) and <code>version</code> (<code>int</code>) arguments qualify the search. <code>pathForResource</code> is both a class and an instance method.
<code>bundlePath</code>	Returns the bundle's path in an <code>NSString*</code> .
<code>†stripAfterLoading</code>	If flag (<code>BOOL</code>) is <code>YES</code> , symbolics are stripped after the bundle is loaded. Obviously, you call this accessor after having created a bundle and before it is loaded; you leave this set to its default (<code>YES</code>) value for production rather than debugging code.

TABLE 13-5. NSBundle Methods (Continued)

<code>setBundleVersion</code> <code>bundleVersion</code>	<p>Accessors for the bundle version; default is 0.</p> <p>Set: <code>version</code> (unsigned). Get: returns an unsigned.</p>
<code>†setSystemLanguages</code>	<p>Lets the bundle know the user's languages (NSArray*) preferences. Used internally.</p>
<code>localizedStringForKey</code>	<p>Returns an NSString* containing the localized string for key (NSString*). The following two arguments—<code>value</code> (NSString*) and <code>comment</code> (NSString*)—are also returned.</p> <p>You can optionally add the <code>table</code> (NSString*) argument to specify the table to look in; otherwise the default table is used.</p> <p>Note that strings are identified by keys: in that sense, all values of the key are localized. The key might be "savestring," and its localized version might be "Save."</p>

NSFileWrapper

Like NSBundles, NSFileWrappers come close to the operating system. They do in fact wrap the non-object-oriented files that ultimately form the basis of Rhapsody.

File wrappers wrap three kinds of file system structures:

1. files
2. directories
3. links (or aliases)

Subclasses of `NSFileWrapper` exist for each of these structures, but you deal with `NSFileWrapper`'s interface.

The `NSFileWrapper` fabrication methods are shown in Table 13-6. Except for the first one (`initWithPath`), all of the methods in Table 13-6 create an unsaved file in memory; until `writeToFile` is called, they do not (necessarily) exist.

TABLE 13-6. `NSFileWrapper` Fabrication Methods

<code>initWithPath</code>	Initializes an already created <code>NSFileWrapper</code> from <code>path</code> (<code>NSString*</code>).
<code>initWithRegularFileWithContents</code>	Initializes a file <code>NSFileWrapper</code> with <code>contents</code> (<code>NSData*</code>).
<code>initWithDirectoryWithFileWrappers</code>	Initializes a directory <code>NSFileWrapper</code> with <code>wrappers</code> (<code>NSDictionary*</code>).
<code>initWithSymbolicLinkWithDestination</code>	Initializes an alias <code>NSFileWrapper</code> with <code>path</code> (<code>NSString*</code>).
<code>initWithSerializedRepresentation</code>	Initializes a file wrapper from <code>data</code> (<code>NSData*</code>) which is a serialized representation of a file's data; the appropriate type of wrapper is determined from the data and is created.

In Table 13-7 you will find the basic `NSFileWrapper` routines that you can use. The most frequently used methods are used to interact with disk.

Directory file wrapper methods are shown in Table 13-8.

TABLE 13-7. NSFileWrapper Utility Methods

<code>writeToFile</code>	Writes the NSFileWrapper's file to disk in <code>path</code> (NSString*); according to <code>atomicFlag</code> (BOOL). The final argument <code>updateNamesFlag</code> (BOOL) is used to control whether the file wrapper's filename field is updated to match what has happened on disk. Returns a BOOL indicating the success of the operation.
<code>serializedRepresentation</code>	Returns an NSData* object containing the file's data that can be used with <code>initWithSerializedRepresentation</code> or to copy to the pasteboard.
<code>isRegularFile</code> <code>isDirectory</code> <code>isSymbolicLink</code>	Returns a BOOL indicating the type of file wrapper this is.
<code>setFilename</code> <code>setPreferredFileName</code> <code>fileName</code> <code>preferredFileName</code>	<p>You can specify a file name as preferred, but since it may match a duplicate key in a directory's dictionary, it may not be used. These accessors let you manipulate both file names, although you normally manipulate only the preferred file name directly.</p> <p>Set: <code>fileName</code> (NSString*). Get: returns an NSString*.</p>
<code>setIcon</code> <code>icon</code>	<p>The icon is used to override the default icons based on file types.</p> <p>Set: <code>anImage</code> (NSImage*). Get: returns an NSImage*.</p>

TABLE 13-7. NSFileWrapper Utility Methods (Continued)

<code>setFileAttributes</code> <code>fileAttributes</code>	<p>Accessor for the file attributes (see <code>NSFileManager</code>).</p> <p>Set: <code>attributes</code> (<code>NSDictionary*</code>).</p> <p>Get: returns an <code>NSDictionary*</code>.</p>
<code>needsToBeUpdatedFromPath</code>	Returns a <code>BOOL</code> indicating if the <code>NSFileWrapper</code> needs to be refreshed from disk using <code>path</code> (<code>NSString*</code>).
<code>updateFromPath</code>	Refreshes the <code>NSFileWrapper</code> object from disk using <code>path</code> (<code>NSString*</code>). Returns a <code>BOOL</code> indicating success.
<code>regularFileContents</code>	Returns an <code>NSData*</code> object containing a regular file's data.
<code>symbolicLinkDestination</code>	Returns an <code>NSString*</code> with a link's actual file path.

TABLE 13-8. NSFileWrapper Methods for Directories

<code>addFileWrapper</code>	<p>Adds wrapper (<code>NSFileWrapper*</code>) to a directory file wrapper. This method is called automatically after specific types of file wrappers have been created (see the following methods).</p> <p>Returns the dictionary key for the new wrapper as an <code>NSString*</code>.</p>
-----------------------------	---

**TABLE 13-8. NSFileWrapper Methods for Directories
(Continued)**

<code>addFileWithPath</code>	<p>Creates a new <code>NSFileWrapper</code> based on the file with <code>path</code> (<code>NSString*</code>) and adds it (see <code>addFileWrapper</code>) to the directory file wrapper.</p> <p>Returns the dictionary key for the new wrapper as an <code>NSString*</code>.</p>
<code>addRegularFileWithContents</code>	<p>Creates a new <code>NSFileWrapper</code> from <code>contents</code> (<code>NSData*</code>) and <code>fileName</code> (<code>NSString*</code>), then adds it to this wrapper.</p> <p>Returns the dictionary key for the new wrapper as an <code>NSString*</code>.</p>
<code>addWSymbolicLinkWith Destination</code>	<p>Creates a new <code>NSFileWrapper</code> from <code>path</code> (<code>NSString*</code>) and <code>fileName</code> (<code>NSString*</code>), then adds it to this wrapper.</p> <p>Returns the dictionary key for the new wrapper as an <code>NSString*</code>.</p>
<code>removeFileWrapper</code>	<p>Removes <code>wrapper</code> (<code>NSFileWrapper*</code>) from the directory file wrapper object. This method also releases the removed file wrapper; it has no effect on files on disk.</p>
<code>keyForFileWrapper</code>	<p>Returns an <code>NSString*</code> with the key for <code>wrapper</code> (<code>NSFileWrapper*</code>) within the directory's dictionary.</p>
<code>fileWrappers</code>	<p>Returns an <code>NSDictionary*</code> with the file wrappers for this directory file wrapper.</p>

Summary

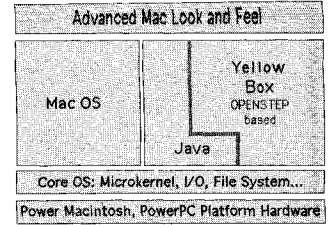
The classes in this chapter are workhorses; you rarely subclass them, but you use them constantly—either directly or through the inherited OpenStep classes that you are using.

The Save and Open panels allow customization by adding accessory views and delegates: for that reason, it is unlikely that you will directly subclass those objects.

Bundles and file wrappers serve as mediators between the object-oriented world and the Rhapsody operating system; subclassing them not only is unnecessary—it might cause serious problems.

Chapter 14

Interaction, Events, and Responders



Events are the heart of OpenStep—they are sent back and forth among objects (usually in response to user actions); many of the primary interface elements (NSApplication, NSView, NSWindow) are subclasses of the NSResponder class which handles events.

This chapter is about the programming tools that you use to manage events. The architecture does not differ greatly from other frameworks that you may have used; the most significant difference is the widespread use of delegates—objects that get a chance to respond to events on behalf of a primary object if the primary object can't respond. Delegates are a way in which you can add functionality to standard objects such as views and windows without overriding them. (Users of MacApp will recognize a somewhat comparable

structure in MacApp's behaviors; however, behaviors in MacApp normally come into play before the primary object; furthermore, behaviors are of the special class TBehavior, while most delegates can be of any class type.)

You never instantiate an NSResponder object directly; you use and sometimes override descendants of it as necessary. Structurally, you will notice that the NSResponder object deals with many more types of events and actions than comparable objects in other frameworks do (see the lists of events in the tables "NSResponder Mouse Event Methods" on page 402, "NSResponder Keyboard and Text-Related Event Methods" on page 403, and "NSResponder Other Event Methods" on page 406). Whereas in other frameworks you often have to construct such relatively sophisticated events (such as deleting the word surrounding the insertion point), OpenStep has done that work for you.

Programming Tools

The two principal elements of event management are the events themselves and responders. NSResponder is an abstract class from which NSView, NSWindow, and NSApplication descend.

NSEvent

Events are generated by the operating system; most of them are associated with a window. The event and its window are packaged together and delivered to the application which then processes them—normally by sending the event to the appropriate window's first responder.

Five primary event types are defined:

1. keyboard events
2. mouse events
3. tracking events

4. periodic events

5. other events

Keyboard and mouse events have counterparts in other frameworks and represent user actions involving the keyboard and mouse.

Tracking events handle events that occur repeatedly as the mouse is moved within a given area. In other frameworks, your mouse tracking is often built in your application using the mouse events that are provided. In OpenStep, this common code (usually centered around a tight loop that is called repeatedly while the mouse is being moved and until it is stopped or the mouse button is released) has been moved into the framework itself. It relies on a specific area (a tracking rectangle) that you can define; thereafter, mouse events within that tracking rectangle are treated specially. As you might expect, this structure is also used to manage the cursor—which can change as it moves from one tracking rectangle to another.

Periodic events are events that you can create and cause to be created repeatedly and sent to your application. Use periodic events to cause actions to occur even while no user action is occurring. For example, zooming in or out while holding down a plus or minus key occurs in response to a single event (the key down), but you want to process the zoom command repeatedly until the key is released.

All types of events share certain arguments in their creation methods:

Event type Within each of the five event types, a further categorization is provided with event types, which are of `NSEventType`. Valid event types for each kind of event are shown in Table 14-1.

TABLE 14-1. `NSEvent` Types and Flags Constants for `NSEvents`

	<i>NSEventType</i>	<i>Flags Constants</i>
key- board event	<code>NSKeyDown</code>	<code>NSAlphaShiftKeyMask</code>
	<code>NSKeyUp</code>	<code>NSShiftKeyMask</code>
	<code>NSFlagsChanged</code>	<code>NSControlKeyMask</code>
		<code>NSAlternateKeyMask</code>
		<code>NSCommandKeyMask</code>
		<code>NSNumericPadKeyMask</code>
		<code>NSHelpKeyMask</code>
mouse event	<code>NSMouseDown</code>	<code>NSLeftMouseDown</code>
	<code>NSLeftMouseDown</code>	<code>NSLeftMouseUp</code>
	<code>NSLeftMouseUp</code>	<code>NSRightMouseDown</code>
	<code>NSLeftMouseDragged</code>	<code>NSRightMouseUp</code>
	<code>NSRightMouseDown</code>	<code>NSMouseMoved</code>
	<code>NSRightMouseUp</code>	<code>NSLeftMouseDragged</code>
	<code>NSRightMouseDragged</code>	<code>NSRightMouseDragged</code>
track- ing event	<code>NSMouseEntered</code>	
	<code>NSMouseExited</code>	
	<code>NSCursorUpdate</code>	
other event	<code>NSAppKitDefined</code>	
	<code>NSSystemDefined</code>	
	<code>NSApplicationDefined</code>	
	<code>NSPeriodic</code>	

Modifier Flags The flags argument is an unsigned int; for each type of event you can construct a flags argument by using the C OR operator to combine as many of the specific flags as you want. The valid constants for each event type are shown in Table 14-1.

Location This is an `NSPoint` in the coordinates of the window for the event; it is the location of the mouse.

Timestamp This is an `NSTimeInterval` indicating the time since system start-up when the event occurred; it is a relative timestamp, not an absolute date and time.

Window Number The `windowNum` variable is an `int` that identifies the window associated with the event.

Context The Display PostScript (DPS) context for the event is an `NSDPSCContext*` object (context) for the event.

Table 14-2 lists the `NSEvent` fabrication methods. Keyboard, mouse, tracking, and periodic events are normally created by the system itself; you create other events as needed. In the case of periodic events, you tell the system to start or stop creating periodic events; you don't create them yourself.

TABLE 14-2. `NSEvent` Fabrication Methods

`enterExitEventWithType`

Creates and returns an `enterExitEvent` based on `type` (`NSEventType`), `location` (`NSPoint`), `flags` (`unsigned int`), `time` (`NSTimeInterval`), `windowNum` (`int`), and `context` (`NSDPSCContext*`).

Additional arguments are `eventNumber` (`int`)—a serial counter for mouse events during this execution of the application, as well as `trackingNumber` (`int`) and `userData` (`void*`)—values set in `NSView`'s `addTrackingRect` method.

TABLE 14-2. NSEvent Fabrication Methods (Continued)

<code>keyEventWithType</code>	<p>Creates and returns a <code>keyEvent</code> based on <code>type</code> (<code>NSEventType</code>), <code>location</code> (<code>NSPoint</code>), <code>flags</code> (unsigned int), <code>time</code> (<code>NSTimeInterval</code>), <code>windowNum</code> (int), and <code>context</code> (<code>NSDPSCContext *</code>).</p> <p>Additional arguments are the <code>character(s)</code> associated with the event—<code>characters</code> (<code>NSString *</code>), the characters without modifier (except shift)—<code>unmodCharacters</code> (<code>NSString *</code>), an indication of whether the key event was generated by a key having been held down—<code>repeatKey</code> (<code>BOOL</code>), and the keyboard key which generated the event—<code>code</code> (unsigned short int).</p>
<code>mouseEventWithType</code>	<p>Creates and returns a <code>mouseEvent</code> based on <code>type</code> (<code>NSEventType</code>), <code>location</code> (<code>NSPoint</code>), <code>flags</code> (unsigned int), <code>time</code> (<code>NSTimeInterval</code>), <code>windowNum</code> (int), and <code>context</code> (<code>NSDPSCContext *</code>).</p> <p>Additional argument are <code>eventNumber</code> (int)—a serial counter for mouse events during this execution of the application, <code>click</code> (int)—the number of mouse clicks, and <code>pressure</code> (float)—the pressure on the device (applicable for writing tablets, etc.).</p>

TABLE 14-2. NSEvent Fabrication Methods (Continued)

+otherEventWithType	Creates and returns an otherEvent based on type (NSEventType), location (NSPoint), flags (unsigned int), time (NSTimeInterval), windowNum (int), and context (NSDPSCContext *). Additional arguments subType (short int), data1 (int), and data2 (int) further define these events and are used internally.
+startPeriodicEvents AfterDelay	This thread starts to generate periodic events starting after delaySeconds (NSTimeInterval) and continuing at periodSeconds (NSTimeInterval).
+stopPeriodicEvents	Stops periodic events for this thread.

You can access any of the parameters of an NSEvent using the methods shown in Table 14-3. Note that these are accessors to the values set when the events were constructed using the methods in Table 14-2. They don't change during the life of the NSEvent.

TABLE 14-3. NSEvent Accessors

type locationInWindow modifierFlags timestamp windowNumber context	These accessors return the arguments of the NSEvent as they were created; the types are shown in Table 14-2.
---	--

TABLE 14-3. NSEvent Accessors (Continued)

<code>window</code>	Returns the <code>NSWindow*</code> associated with the <code>NSEvent</code> .
<code>characters</code> <code>charactersIgnoringModifiers</code> <code>isARepeat</code> <code>keyCode</code>	For keyEvents, returns these arguments as they were created; the types are shown in Table 14-2.
<code>clickCount</code> <code>pressure</code>	For mouseEvents, returns these arguments as they were created; the types are shown in Table 14-2.
<code>eventNumber</code>	For mouseEvent and enterExitEvents, returns the eventNumber as an int.
<code>trackingNumber</code> <code>userData</code>	For enterExitEvents, returns the arguments as they were created; the types are shown in Table 14-2.
<code>subtype</code> <code>data1</code> <code>data2</code>	For otherEvents, returns the arguments as they were created; the types are shown in Table 14-2.

NSResponder

NSResponders respond to events. Most interface elements (views, windows, and their descendants) as well as the `NSApplication` have `NSResponder` as a superclass.

The methods of `NSResponder` fall into three categories:

1. There are methods to manage the responder chain.
2. You can walk the responder chain to find if a method can be processed by specific objects.
3. Other methods handle specific events such as key down events and mouse events.

Managing the Responder Chain You call these methods for any responder object (such as a view within a window) if you want to alter its position in the responder chain.

Normally, OpenStep does this for you in response to user actions such as mouse clicks that activate windows. (It also automatically modifies the responder chain as windows are opened and closed.) However, there are a number of times when you want to change the target or focus of the user's attention and you modify the responder chain yourself. The only normal modification that you make is to call `NSWindow's makeFirstResponder` method, which calls the appropriate methods here.

Table 14-4 shows the responder chain management methods.

TABLE 14-4. NSResponder—First and Next Responder Methods

<i>acceptsFirstResponder</i>	Returns a <code>BOOL</code> indicating if it is willing to become a first responder. You must override this method if your view is willing to become a first responder.
<i>becomeFirstResponder</i>	Returns a <code>BOOL</code> indicating if it has successfully carried out this request to become the first responder. Override this method to do ancillary processing; do not call it—use <code>NSWindow's makeFirstResponder</code> .

TABLE 14-4. NSResponder—First and Next Responder Methods (Continued)

<i>resignFirstResponder</i>	Returns a BOOL indicating if it has successfully resigned first responder in response to this request. It is called automatically as a side effect of asking another object to become the first responder. Override it to do ancillary processing.
<i>setNextResponder</i> <i>nextResponder</i>	Accessors for the object's next responder in the linked list of the responder chain. You can modify the chain as needed; you rarely do so. OpenStep handles the layering of windows and modifications to the responder chain caused thereby. Modifying the responder chain within a single window can confuse the user. The only time when it is essential is when views are dynamically added to or removed from windows, and NSWindow appropriately calls these methods as needed. Set: aResponder (NSResponder*). Get: returns an NSResponder.

Messages and the Responder Chain You never override these methods, but you call them as necessary to find recipients of messages—objects that can perform the tasks at hand. In most cases, however, you use the NSApplication methods (which call these as needed) to do the work. The methods are shown in Table 14-5.

TABLE 14-5. NSResponder Message and Command Management Methods

<code>flagsChanged</code>	Modifier keys (shift, control, etc.) have changed and you should re-examine <code>theEvent</code> (NSEvent*).
<code>doCommandBySelector</code>	The responder attempts to perform <code>aCommand</code> (SEL). This is the heart of OpenStep method dispatching.
<code>tryToPerform</code>	Comparable to <code>doCommandBySelector</code> . The arguments are <code>anAction</code> (SEL) and <code>anObject</code> —which is passed to <code>anAction</code> if it can be performed. Returns a BOOL indicating success.
<code>validRequestorFor</code>	In order to build and update the Services menu, this method is repeatedly invoked for members of the responder chain with the arguments <code>sendType</code> (NSString*) and <code>receiveType</code> (NSString*). Return either nil or self—if your object can avail itself of such a service.
<code>noResponderFor</code>	This is called with <code>eventSelector</code> (SEL) when the appropriate method cannot be found in the object. This is your chance to redirect <code>eventSelector</code> to another object. It is also your opportunity to raise an exception if you feel that no further object in the responder chain should respond to the event.

Responding to Events You almost never override NSResponder directly; instead, you override one of its descendants, such as NSView, to implement specific functionality. The mouse events of NSResponder are shown in Table 14-6.

TABLE 14-6. NSResponder Mouse Event Methods

<i>mouseDown</i>	Each of these methods takes a single <i>theEvent</i> (NSEvent*) argument.
<i>mouseUp</i>	
<i>rightMouseDown</i>	
<i>rightMouseUp</i>	
<i>mouseDragged</i>	NSResponder's implementations do nothing; those of NSView, NSWindow, etc. often perform default and standard functions.
<i>rightMouseDragged</i>	
<i>mouseMoved</i>	Override them for your objects that implement additional or custom functionality.
<i>mouseEntered</i>	
<i>mouseExited</i>	
<i>keyUp</i>	
<i>keyDown</i>	

Keyboard and text-related events are shown in Table 14-7. As with the mouse events, the default NSResponder implementations do nothing. Subclasses such as NSViews and NSTextView in particular override these methods as needed.

The OpenStep keyboard and text-related events are extensive compared to other frameworks. In other environments you may have had to go from a mouse down to an insertion point and from there do manipulations to find the word or paragraph that you are dealing with. In OpenStep, the knowledge of the textual context—that is, words, lines, and paragraphs as opposed to basic mouse points—is available to NSResponder.

The methods shown in Table 14-7 let you function at a very high level. If you are using text, you probably will rarely use the mouse event methods shown in Table 14-6. Furthermore, review NSTextView itself carefully: you may never call or override these methods, since it provides basic implementations. Of course, if you need to provide alternative implementations or specialized functionality, you will override these methods.

What you will do frequently is to expose these actions in your user interface. Most of the methods in Table 14-6 are actions and as such can be wired to interface elements in your application. These in turn can be provided with keyboard equivalents for power users.

Most of these routines operate on the character immediately adjacent to the current insertion point—or on the word, line, or paragraph in which the insertion point is contained. Some routines also use the concept of a mark which can be set to indicate a point in the text stream. You use the combination of a mark (as a starting point) and the current insertion point (as an ending point) to select a range of text.

As with most text systems, insertion and deletion of characters occur at the insertion point; furthermore, if some text has been selected, insertion replaces the selected text with the inserted text.

TABLE 14-7. NSResponder Keyboard and Text-Related Event Methods

<code>deleteBackward</code>	Each of these methods takes a single sender argument.
<code>deleteForward</code>	
<code>deleteToBeginningOfLine</code>	They are implemented in descendants of <code>NSResponder</code> .
<code>deleteToBeginningOfParagraph</code>	
<code>deleteToEndOfLine</code>	
<code>deleteToEndOfParagraph</code>	
<code>deleteToMark</code>	
<code>deleteWordBackward</code>	
<code>deleteWordForward</code>	

TABLE 14-7. NSResponder Keyboard and Text-Related Event Methods (Continued)

insertBacktab	Each of these methods takes a single sender argument.
insertNewLine	
insertNewlineIgnoring FieldEditor	They are implemented in descendants of NSResponder.
insertParagraphSeparator	
insertTab	
insertTabIgnoringField Editor	
moveBackward	Each of these methods takes a single sender argument.
moveBackwardAndModify Selection	
moveDown	The insertion point is moved a single character, unless the method name suggests otherwise.
moveDownAndModifySelection	
moveForward	Many of these methods are in pairs: those that modify the selection extend (or shorten) the selection by the character or word that they have moved. Otherwise, the insertion point is moved and the selection (if any) is unchanged.
moveForwardAndModify Selection	
moveLeft	They are implemented in descendants of NSResponder.
moveRight	
moveToBeginningOfDocument	
moveToBeginningOfLine	
moveToBeginningOfParagraph	
moveToEndOfDocument	
moveToEndOfLine	
moveToEndOfParagraph	
moveUp	
moveUpAndModifySelection	
moveWordBackward	
moveWordBackwardAndModify Selection:	
moveWordForward	
moveWordForwardAnd ModifySelection	
changeCaseOfLetter	Each of these methods takes a single sender argument.
capitalizeWord	
lowercaseWord	They are implemented in descendants of NSResponder.
uppercaseWord	

TABLE 14-7. NSResponder Keyboard and Text-Related Event Methods (Continued)

<code>indent</code>	This method takes a single <code>sender</code> argument. It is implemented by subclasses which can determine what the meaning of <code>indent</code> is (no value is provided in the method call).
<code>transpose</code> <code>transposeWords</code>	Each of these methods takes a single <code>sender</code> argument. They are implemented in descendants of <code>NSResponder</code> . They transpose the characters or words adjacent to the insertion point; the insertion point should be moved beyond the transposed elements when the operation is completed.
<code>flushBufferedKeyEvents</code>	A stub to be implemented by subclasses as needed to get rid of processed (or to-be-ignored) key events. It takes no arguments.
<code>insertText</code>	Inserts a <code>String</code> (<code>NSString*</code>) at the insertion point. Implemented by subclasses.
<code>yank</code>	This method takes a single <code>sender</code> argument. It is implemented by subclasses to replace selected text at the insertion point by text that had been deleted (and hence automatically placed in a kill buffer).

Note that “forward” and “backward” are script dependent—they do not necessarily mean “right” and “left.” Also, some descendants of `NSResponder` have field editors that are called when text changes within their purview. Some of the methods

in `NSResponder` allow you to insert and delete characters and to prevent field editors of descendants from interfering; unless otherwise noted, the field editors in descendants (if they exist) are permitted to interfere after `NSResponder` text editing methods are called.

Other events that `NSResponder` handles are shown in Table 14-8. These events, which may come in response to keyboard commands from special keys (page up/down, etc.), may also be wired as actions to your interface elements as appropriate. Also in Table 14-8 you will find the balance of the `NSResponder` methods. These are methods that manage windows (page up/page down), selections that are not necessarily limited to text, as well as menus and events.

TABLE 14-8. `NSResponder` Other Event Methods

<code>pageDown</code>	Each of these methods takes a single sender argument.
<code>pageUp</code>	
<code>scrollLineDown</code>	
<code>scrollLineUp</code>	
<code>scrollPageDown</code>	
<code>scrollPageUp</code>	They are implemented in descendants of <code>NSResponder</code> .
<code>selectAll</code>	Each of these methods takes a single sender argument.
<code>selectLine</code>	
<code>selectParagraph</code>	
<code>selectToMark</code>	
<code>selectWord</code>	
<code>setMark</code>	They are implemented in descendants of <code>NSResponder</code> .
<code>swapWithmark</code>	As with other methods, they function relative to the current insertion point.
	(Note: <code>swapWithMark</code> reverses the insertion point and the mark: the beginning of the selection becomes the end and vice versa.)

TABLE 14-8. NSResponder Other Event Methods (Continued)

<code>centerSelectionInVisibleArea</code>	This method takes a single <code>sender</code> argument. It is implemented by subclasses.
<code>complete</code>	This method takes a single <code>sender</code> argument. It is implemented by subclasses to do any completion routines that are relevant.
<code>setMenu</code> <code>menu</code>	Accessors for the menu associated with this responder (the application, window, view, etc.). Set: <code>aMenu</code> (<code>NSMenu*</code>). Get: returns an <code>NSMenu</code> .
<code>interpretKeyEvents</code>	A utility function for keyboard events to send the characters in <code>eventArray</code> (<code>NSArray*</code>) to the system. Used internally.
<code>performKeyEquivalent</code>	If the character in <code>theEvent</code> (<code>NSEvent*</code>) is this event's key equivalent, attempts to perform it and returns a <code>BOOL</code> indicating success.
<code>performMnemonic</code>	Same as <code>performKeyEquivalent</code> , but the single argument is <code>aString</code> (<code>NSString*</code>) containing the key code.
<code>showContextHelp</code> <code>helpRequested</code>	Each of these methods takes a single <code>sender</code> argument. They are implemented in descendants of <code>NSResponder</code> . They differ in that <code>helpRequested</code> should be passed along to supervisors, windows, etc. if the receiving object cannot provide help.

NSServices-Request[P]

As noted previously, `validRequestForServiceType` allows each `NSResponder` to indicate whether it can process data of certain types; this enables the Service menu to be maintained.

If the user then in fact selects a service, the two methods of the `NSServicesRequest` protocol that are shown in Table 14-9 are invoked. `NSServiceRequest` is an informal protocol and is implemented as a category.

TABLE 14-9. NSServicesRequest Methods

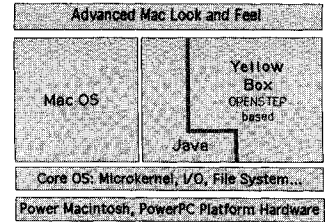
<i>writeSelectionToPasteboard</i>	<p>The <code>NSResponder</code> is directed to write its selection to <code>pboard</code> (<code>NSPasteboard *</code>) using one of types (<code>NSArray *</code>).</p> <p>The method returns a <code>BOOL</code> indicating success.</p>
<i>readSelectionFromPasteboard</i>	<p>After the service has performed its work, this method is sent to the original <code>NSResponder</code> and it retrieves the information <code>pBoard</code> (<code>NSPasteboard *</code>) and replaces the current selection with it.</p>

Summary

The responder chain is the heart of OpenStep (as it is of all such frameworks). It manages events and actions, menus, and messages. Compared to other frameworks, you will notice a much greater richness and more sophisticated set of events that are built into OpenStep. Most of the methods described in this chapter are stubs in `NSResponder`. They are overridden in its descendants—usually for you in `NSView`, `NSWindow`, etc.

Chapter 15

Printing



As with other frameworks, much of the work of printing is done for you automatically once you have the code to draw on the screen. With OpenStep, since the screen imaging uses PostScript, there is even less of a difference than on other systems between the screen image and output images (to printers, other imaging devices, plotters, and faxes).

The four classes that come into play specifically for printing are described in this chapter:

- *NSPrinter represents the physical printer device.*
- *NSPrint panel is the user interface element that lets people specify page numbers, specific printers, etc.*

- *NSPrintOperation* is an object encapsulating the print job—the combination of options from the Print panel and the specified printer.
- *NSPrintInfo* is the data structure (wrapped in an object) that is passed to the print operation.

The default *NSView* print method creates an *NSPrintOperation* in response to a print command; you don't have to do anything to support basic printing from your application. However, before you skip this chapter, you should at least skim through the architecture of printing in OpenStep. If you have your own documents (as opposed to simply view-based images) or if you want to do anything in addition to the defaults, you'll need to know where to add the code.

Controls

The sole control involved directly with printing is the Print panel, in which the user specifies page range, print quality, a specific printer, etc. The Print panel allows users to request a preview of the job as well.

NSPrintPanel

The Print panel is a standard interface element. You should use it as it is without modifying its structure or redesigning its buttons and fields. If you have specific needs for additional information for your application, create an accessory view and associate it with the Print panel. Users access your view by clicking the standard Print panel's Options button; if you have no accessory view, the Options button is disabled.

Before displaying the *NSPrintPanel*, you have created an *NSPrintOperation* object (which is done automatically by the print method of *NSView*). A single *NSPrintOperation* object is current for an application at a time. When it is created, either a default *NSPrintInfo* object is assigned to it or a specific one is created. *NSPrintPanel* assumes the existence of this

`NSPrintInfo` object: it reads starting values from it and writes values to it when OK is clicked.

(Long-time users of Mac OS can identify a number of products that did not use the standard print dialog but instead re-implemented it with their own fields. Developers who followed the rules on how to extend the standard print dialog had no problem during the redesigns of that dialog with the releases of Mac OS 7 and 7.5. Those who had re-implemented the whole dialog were up the creek. Learn.)

Since the Print panel (without its accessory views) is standard, the operating system can reuse the physical panel. Thus, when you request a new Print panel, you may get a new one—or one that happens to be hanging around. In either case, it is set to default values. However, if you insist on modifying the Print panel that you have been given, those modifications will be passed on to the next application that gets the panel—and they probably won't be welcome.

The `NSPrintPanel` methods are shown in Table 15-1.

TABLE 15-1. `NSPrintPanel` Methods

<code>+printPanel</code>	Returns an <code>NSPrintPanel*</code> for your use.
<code>setAccessoryView accessoryView</code>	<p>Accessor for your accessory view with application-specific fields and buttons; opened by clicking the Options button.</p> <p>Set: <code>aView</code> (<code>NSView*</code>). Get: returns an <code>NSView</code>.</p>

TABLE 15-1. NSPrintPanel Methods (Continued)

<code>runModal</code>	Displays the Print panel and runs a modal loop until either Cancel or OK is clicked. Returns an int with the constant value <code>NSCancelButton</code> or <code>NSOkButton</code> .
<code>updateFromPrintInfo</code> <code>finalWritePrintInfo</code>	Updates the panel with data from the <code>NSPrintInfo</code> object associated with the current <code>NSPrintOperation</code> ; the second method writes values from the Print panel to the <code>PSPrintInfo</code> object.
<code>pickedAllPages</code> <code>pickedButton</code> <code>pickedLayoutList</code>	These methods take a single sender argument. They are invoked when the user clicks the All Pages button; any of the Cancel, Fax, Preview, Print, or Save buttons; or a new layout. You can then specify action to modify values in the Print panel.

Programming Tools

Printing, whether actual printing to a printer or the generation of Encapsulated PostScript (EPS) calls, is managed by an interaction among the `NSPrintOperation` (which supervises the work); `NSView`, which does the actual imaging (as it does for visualization on the screen); and the `NSPrintInfo` object, which contains parameters and controls garnered from the `NSLayoutManager` object, `NSView`, and the `NSPrintPanel`'s user interface described previously in this chapter.

NSView—which is a superclass of NSWindow—often contains subviews; the interaction of NSView with printing is the interaction of all of those views as filtered through the view (or window) that is managing the printing and that initiates the print command.

You create an NSPrintOperation object and tell it to run the print job; you can use the default NSPrintInfo object. Likewise, if you are using the default printer, you need not worry about creating an NSPrinter (and NSPrintPanel manages the process of a user selecting a printer on the network).

This section describes the NSPrintInfo object and the NSPrintOperation object; it ends with a description of the last major player in the puzzle—the NSPrinter.

NSPrintInfo

An application uses a single NSPrintInfo object for all printing tasks and for its layout managers. This shared NSPrintInfo object can be modified and changed, but only one is the current shared object. If you have more than one document (and often if you have more than one document type), you associate an NSPrintInfo object with each document; when it comes time to print a document, you create an NSPrintOperation that includes that document's NSPrintInfo object.

The NSPrintInfo contains the parameters you would expect—margins, pagination controls, the disposition of the print job (to a printer, a file, a fax, or a preview), which printer to use, etc. NSPrintInfo itself doesn't do anything. The methods of NSPrintInfo (all of which are accessors except for an initialization method) are shown in Table 15-2.

TABLE 15-2. NSPrintInfo Methods

<code>initWithDictionary</code>	Initializes an <code>NSPrintInfo</code> object with a <code>Dictionary</code> (<code>NSDictionary*</code>).
<code>setSharedPrintInfo</code> <code>sharedPrintInfo</code>	Accessors for the application's single currently active shared <code>NSPrintInfo</code> object. (You don't have to set it; there's a default one already there for you.) Set: <code>printInfo</code> (<code>NSPrintInfo*</code>). Get: returns an <code>NSPrintInfo*</code> .
<code>setTopMargin</code> <code>setLeftMargin</code> <code>setBottomMargin</code> <code>setRightMargin</code>	Accessors for page margins. The units are points. Set: <code>margin</code> (<code>float</code>). Get: returns a <code>float</code> .
<code>topMargin</code> <code>leftMargin</code> <code>bottomMargin</code> <code>rightMargin</code>	
<code>setOrientation</code> <code>orientation</code>	The values of <code>NSPrintOrientation</code> are <code>NSPortraitOrientation</code> and <code>NSLandscapeOrientation</code> . Set: <code>orientation</code> (<code>NSPrintOrientation</code>). Get: returns an <code>NSPrintOrientation</code> .
<code>sizeForPaperName</code>	Returns an <code>NSSize</code> for <code>name</code> (<code>NSString*</code>).
<code>setPaperName</code> <code>paperName</code>	You can name paper sizes (Letter, A5, etc.); the dictionary contains the standard paper names and their sizes. Set: <code>name</code> (<code>NSString*</code>). Get: returns an <code>NSString</code> .

TABLE 15-2. NSPrintInfo Methods (Continued)

<code>setPaperSize</code> <code>paperSize</code>	<p>Accessor for the current paper's size.</p> <p>Set: <code>aSize</code> (NSSize). Get: returns an NSSize.</p>
<code>setHorizontalPagination</code> <code>setVerticalPagination</code>	<p>Values for <code>NSPrintingPagination</code> are:</p> <p><code>NSAutoPagination</code>—splits the image into page-size sections along the horizontal or vertical dimension.</p> <p><code>NSFitPagination</code>—scales the image (after any view scaling that has already been done) to fit the width or height of a single page.</p> <p><code>NSClipPagination</code>—instead of scaling, this mode clips the image to the width or height of a single page.</p> <p>Set: <code>mode</code> (<code>NSPrintingPaginationMode</code>). Get: returns an <code>NSPrintingPaginationMode</code>.</p>
<code>horizontalPagination</code> <code>verticalPagination</code>	
<code>setHorizontallyCentered</code> <code>setVerticallyCentered</code>	<p>Determine whether the image will be centered on the page.</p> <p>Set: <code>flag</code> (BOOL). Get: returns a BOOL.</p>
<code>isHorizontallyCentered</code> <code>isVerticallyCentered</code>	
<code>setDefaultPrinter</code> <code>setPrinter</code>	<p>Accessors for the printer to use for this job as well as for the application's default printer.</p> <p>Set: <code>aPrinter</code> (NSPrinter*). Get: returns an NSPrinter*.</p>
<code>defaultPrinter</code> <code>printer</code>	

TABLE 15-2. NSPrintInfo Methods (Continued)

<code>setJobDisposition</code> <code>jobDisposition</code>	The possible values for <code>NSPrintSpoolJob</code> , <code>NSPrintFaxJob</code> , <code>NSPrintPreviewJob</code> , <code>NSSaveJob</code> , and <code>NSPrintCancelJob</code> . Set: <code>disposition</code> (<code>NSString*</code>). Get: returns an <code>NSString</code> .
<code>setUpPrintOperation</code> <code>DefaultValues</code>	You override this method if you subclass <code>NSPrintInfo</code> to perform initialization.
<code>dictionary</code>	Returns the <code>NSDictionary*</code> associated with this <code>NSPrintInfo</code> object.

NSPrintOperation

The `NSPrintOperation` (Table 15-3) oversees the work: posing the `NSPrintPanel`, updating the `NSPrintInfo`, running the print job.

TABLE 15-3. NSPrintOperation Methods

<code>runOperation</code>	Call this method from the print method of your document; if you have no document, call it from the print method of your view (the default method does that). Also call it in response to a Copy command. This method returns a <code>BOOL</code> indicating the success of the operation.
---------------------------	--

TABLE 15-3. NSPrintOperation Methods (Continued)

<code>+printOperationWithView</code>	Creates and returns an <code>NSPrintOperation*</code> for <code>view</code> (<code>NSView*</code>). The optional <code>printInfo</code> (<code>NSPrintInfo*</code>) parameter is used if you want to use your document's <code>NSPrintInfo</code> object (which you normally do if you have a document).
<code>initWithView</code>	Initializes an already created <code>NSPrintOperation</code> with <code>view</code> (<code>NSView*</code>).
<code>+EPSOperationWithView</code>	<p>This method creates and returns an <code>NSPrintOperation*</code> for <code>view</code> (<code>NSView*</code>); the operation will convert the area of the view specified in <code>rect</code> (<code>NSRect</code>) to EPS. The EPS calls are stored in <code>toData</code> (<code>NSMutableData*</code>). You may optionally add a final <code>aPrintInfo</code> (<code>NSPrintInfo*</code>) argument if you don't want to use the current defaults.</p> <p>An alternative allows you to specify a file with <code>toPath</code> (<code>NSString*</code>) instead of <code>toData</code>.</p>
<code>initEPSOperationWithView</code>	Initializes an already created <code>NSPrintOperation*</code> for EPS; the parameters specify the <code>rect</code> (<code>NSRect</code>) to be printed, the destination of the calls— <code>data</code> (<code>NSMutableData*</code>), and the <code>aPrintInfo</code> (<code>NSPrintInfo*</code>) parameters to use.

TABLE 15-3. NSPrintOperation Methods (Continued)

<pre> +setCurrentOperation +currentOperation </pre>	<p>Only one print operation can be associated with a thread at a time; this accessor manages that for you on the rare occasions when you need to interfere.</p> <p>Set: <code>operation</code> (NSPrintOperation*).</p> <p>Get: returns an NSPrintOperation*.</p>
<pre> isEPSOperation </pre>	<p>Returns a BOOL indicating if this is an EPS operation (data copy) or a printing operation.</p>
<pre> setPrintInfo printInfo </pre>	<p>You can use a print info object other than the default object and other than the one that you used in creating or initializing this object.</p> <p>Set: <code>aPrintInfo</code> (NSPrintInfo*).</p> <p>Get: returns an NSPrintInfo*.</p>
<pre> view </pre>	<p>Returns the view that is doing the printing as an NSView*.</p>
<pre> cleanUpOperation deliverResult </pre>	<p>These methods are called by runOperation at its end. If you must know, <code>cleanUpOperation</code> takes no arguments and returns nothing; <code>deliverResult</code> returns a BOOL—which you get as the result of runOperation.</p>
<pre> setShowPanels showPanels </pre>	<p>By default, the Print panel is shown to control the print operation; this parameter lets you use noninteractive printing (with no user dialog).</p> <p>Set: <code>flag</code> (BOOL).</p> <p>Get: returns a BOOL.</p>

TABLE 15-3. NSPrintOperation Methods (Continued)

<code>setAccessoryView</code> <code>accessoryView</code>	<p>You can set the accessory view for the Print panel here without making any calls directly to NSPrintPanel.</p> <p>Set: <code>view</code> (NSView*).</p> <p>Get: returns an NSView*.</p>
<code>setPrintPanel</code> <code>printPanel</code>	<p>NSPrintingOperation uses the default Print panel, but you can change that.</p> <p>Set: <code>panel</code> (NSPrintPanel*).</p> <p>Get: returns an NSPrintPanel*.</p>
<code>createContext</code> <code>context</code> <code>destroyContext</code>	<p>These methods are used internally to create, access, and then destroy the NSDPSCContext* used in the printing process. Stay out.</p>
<code>currentPage</code>	<p>Returns the page currently being printed as an int. As with all such routines, you should remember that many printers have buffers: this is the page that the application thinks the printer is printing—it may well be several pages behind.</p>
<code>setpageOrder</code> <code>pageOrder</code>	<p>NSPrintingPageOrder can have any of the following values: NSAscendingPageOrder, NSDescendingPageOrder, NSSpecialPageOrder, and NSUnknownPageOrder.</p> <p>Set: <code>order</code> (NSPrintingPageOrder).</p> <p>Get: returns an NSPrintingPageOrder.</p>

The same code that is used for printing to a PostScript output device is also used to copy images to the pasteboard; hence,

all of the code in this section can be invoked either from a user's choice of the Print command or from a choice of the Copy command.

In the case of the Copy command, output is not to a physical device; instead it is to a file or data structure in which the EPS calls are captured. The NSPrintOperation makes this distinction.

NSPrinter

Finally, the NSPrinter object represents a printer (actual or virtual) that is to be used in the printing process.

PostScript printers are described in documentation from Adobe as well as from Apple. For most cases, you never need to move beyond creating a printer with a specific name (if that—the default methods of NSPrintOperation and NSPrintInfo usually do that for you). The information is stored in PostScript Printer Description (PPD) files.

The distinction between printer types and printer names is important to you: printer names are the names of specific printers on the network ("Printer Outside Kennel"), while printer types are the names of printer models ("LaserWriter 16/600").

The accessors for printer PostScript variables are shown in Table 15-4 (along with the other NSPrinter methods), but they are not described.

TABLE 15-4. NSPrinter Methods

<code>+printerWithName</code>	Creates and returns an NSPrinter* with name (NSString*). The printer should exist—you can find the list of printers using printerNames.
-------------------------------	---

TABLE 15-4. NSPrinter Methods (Continued)

tpriinterWithName	This variation lets you specify a printer by name (NSString*), domain (NSString*), and whether it is currently available—flag (BOOL).
tpriinterWithType	Creates and returns an NSPrinter of type (NSString*). Create an abstract printer when you need to find out the attributes of a type of printer rather than a specifically-named one.
tpriinterNames tpriinterTypes	Return the system's recognized printer names and types in an NSArray*.
domain host name note type	Accessors for printer attributes each of which returns an NSString*.
acceptsBinary isColor isOutputStackInReverse Order	Accessors for printer attributes each of which returns an NSString*.
imageRectForPaper	Returns an NSRect for paperName (NSString*).
pageSizeForPaper	Returns an NSSize for paperName (NSString*).
isFontAvailable	Returns a BOOL indicating if faceName (NSString*) is available on the printer.
languageLevel	Returns an int representing the PostScript language level available on the printer.

TABLE 15-4. NSPrinter Methods (Continued)

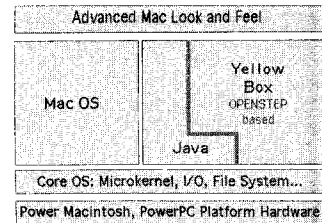
<code>statusForTable</code>	Returns <code>NSPrinterTableStatus</code> for table (<code>NSString*</code>). Possible return values are <code>NSPrinterTableOK</code> , <code>NSPrinterTableNotFound</code> , <code>NSPrinterTableError</code> .
	Use this routine before calling the methods in the following section.
<code>isKey</code> <code>stringForKey</code> <code>stringListForKey</code> <code>booleanForKey</code> <code>floatForKey</code> <code>intForKey</code> <code>rectForKey</code> <code>sizeForKey</code>	These routines return the type indicated for <code>key</code> (<code>NSString*</code>) and <code>inTable</code> (<code>NSString*</code>)—the PPD table.
<code>deviceDescription</code>	Returns an <code>NSDictionary*</code> for this device.

Summary

The basic printing architecture of OpenStep is not complicated. To expand and customize it, you often add an accessory view to the `NSPrintPanel` (for your application-specific printing parameters).

Because so much of the printing operation involves objects that are shared by all printing clients in your application, make sure you play by the rules. Even if your application today has only one document that can be printed and even if it can be printed in only one way, you can't tell what tomorrow may bring. Furthermore, by constructing an application that doesn't play by the rules and is idiosyncratic, you limit its re-use by users, other developers—and even yourself.

Part III: Using Rhapsody



The last part of this book shows Rhapsody in action. These five applications that have already shipped under OpenStep are among the first applications to ship on Rhapsody. The following chapters describe these applications and highlight their use of some of Rhapsody's most interesting features. You will find screen shots, descriptions of the architecture, and critical sections of code that implement these features.

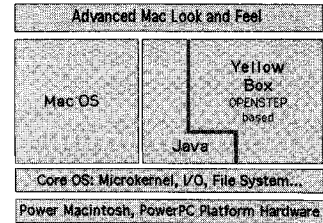
Rhapsody incorporates the tools and techniques that are needed to move into the new world of software—a world that is both simpler and more complex than the world most people know today. It is simpler because it does more with less effort on the parts of developers and users. It is more complex because it is a world of multiple processors, networked computers, and applications that constantly communicate with one another.

The opportunities for developers are vast—both among the small number of the world's population who already use computers and among the vast number who currently don't. Rhapsody provides the infrastructure that lets you build the simple and reliable applications that people need; your imagination lets you envision the sorts of things that people want to do with these robust applications.

This part of the book is far from finished. The applications of tomorrow (whether powerful stand-alone applications or sophisticated and elegant Rhapsody services) are just starting to be written. Maybe yours will be in the next edition of this book, illustrating yet another innovative feature of Rhapsody. (Contact <http://www.philmontmill.com/rhapsody> for more information.)

Chapter 16

Dynamic Binding and Loading Resources with TIFFany



TIFFany™ is an image processing application developed originally on OpenStep for Mach and now available for Rhapsody. Its power derives from its use of the native framework TIFF image type and of Display PostScript commands—as well as from the design and implementation by the developer, Stan Jirman.

From the point of view of the Rhapsody developer, TIFFany offers two interesting case studies:

- *The implementation of ruler views allows a simple customized form of feedback for embedded objects. This demonstrates not only the ease of using the framework's suite of views (including rulers) but also the way in which you can*

easily load resources (such as the arrow markers in rulers) at run time.

- *The implementation of the arrows in the ruler view relies on dynamic binding to a method called boundingBox.*

Image Processing

Figure 16-1 shows a TIFFany window in action.

Image processing is a highly specialized area of the graphics world. It's not drawing or painting: it starts from an image and proceeds to modify that image in various ways. Common modifications include wholesale darkening or lightening of all or part of the image, color changes, removal of scratches on the image caused by physical problems (camera lens damage, for instance), and combining images from several sources.

The basic image is that of a bison (which is the preferred name for the American buffalo among lexicographers—the animals don't care).

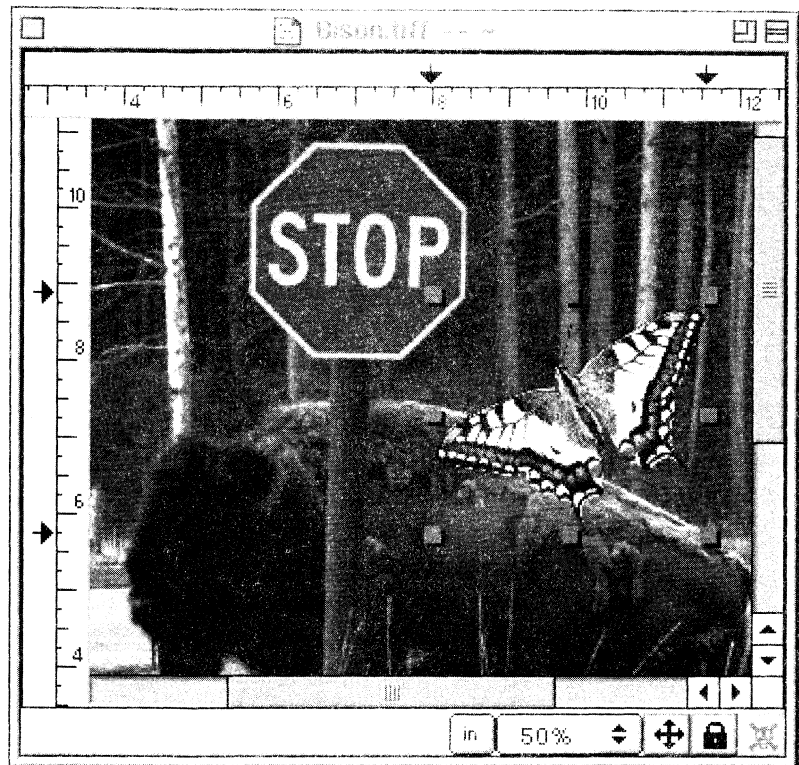
The window displays the OpenStep interface elements; all are present in Rhapsody and in Yellow Box applications running on Mac OS and Windows, but some are in different locations.

The scroll bars, for example, group the up and down arrows at one end of the scroll bar (you can determine which end in your code). Instead of a scroll button to move up and down, the scroll bar presents a bar that represents the size of the image relative to the scrollable size. In this case, the image of the bison is almost the size of the scroller.

The image of the butterfly (with a transparent background) has been pasted in.

You can see the eight resize handles of the butterfly view; you can also see in each ruler the two arrows indicating the edges of the butterfly's view. It is the placing of those markers in the standard `NSRulerView` objects that is the subject of the code excerpt that follows.

FIGURE 16-1. TIFFany in Action



At the bottom of the window are five standard controls. From left, they are:

- a button that lets you choose the units for rulers (inches, centimeters, picas, points)
- an `NSPopUpButton` object that lets the user specify the amount of zoom for the image

- an icon that lets you choose whether to resize the window's view when the window is resized
- a lock to prevent the layer of the image from being moved
- a button to abort the current action (if any)

Placing Markers in Ruler Views

The code to place the markers in the ruler views is an example of dynamic binding. The object for which marker should be placed must implement a method called `boundingBox`, which returns its bounding rectangle in its superview's coordinates.

This may seem old-hat and simple to you, but if you are coming from a framework based on a static language, you may not immediately appreciate the flexibility. The object to be marked must implement a method called `boundingBox`, but the object can be a view, a collection of views, an image, or any other object that implements `boundingBox`. Even if that method is not part of the base object's methods, you can dynamically load it at run time. Furthermore, in the case of objects that have delegates, you can implement the method in a delegate, further reducing the problems of the fragile base class. All that matters is the name of the method and that it returns an `NSRect`. The implementation and the object of which the method is a part are irrelevant.

Setting the Markers

This is the method that actually sets the markers.

```
- (void) addMarkersFor:(GroupObject *)go
{
    static NSPoint hp = {6, 0}, vp = {11, 5};
    NSRulerMarker *m;
    NSRect b;
```

The `NSRulerMarker` class has an image for each marker as well as a single point within that image that defines the location for the marker. The static `NSPoints` identify the location (the arrow tip) within the horizontal and vertical marker images. (Remember that the coordinates run increasingly on the vertical scale: (6,0) is six pixels in and on the base—not top—coordinate line.)

The view object contains two static `NSImages` (`horMark` and `vertMark`) that are initially set to `nil`; they will contain the actual marker images. The next section of code loads them if necessary, using a `define` (which is shown after this method).

```
if (!horMark)
{
    horMark    = IMAGENAMED(@"horMark");
    vertMark   = IMAGENAMED(@"vertMark");
}
```

This is the critical line of code: it calls the embedded object's `boundingBox` method and sets the local variable `b` to its value.

```
b = [go boundingBox];
```

Having gotten the marker locations (and loaded the marker images if necessary), all that's left is to create the marker objects and locate them accordingly.

The `NSRulerMarker`'s class method `alloc` is called to allocate a new instance of `NSRulerMarker`; that instance's `initWithRulerView` method is then called. The arguments for the instance's method set up the marker with the ruler view (`horRuler`—an instance of the view), the left side of the bounding box (`NSMinX(b)`), the image that was loaded if necessary in this method (`vertMark`), and the location for the tip of the marker's arrow (`hp`). Markers are normally not removable by users from their ruler views; the `TIFFany` interface allows them to be removable, so the `setRemovable` method is called with `YES`. Finally, the newly constructed marker (`m`) is added to an array of markers (`hMarkers`).

Here is the code for the left edge in classic Objective-C syntax:

```
m = [[NSRulerMarker alloc] initWithRulerView:horRuler
                                markerLocation:NSMinX (b)
                                image:vertMark
                                imageOrigin:hp];

[m setRemovable:YES];
[hMarkers addObject:m];
```

Here is the same code in modern syntax:

```
m = (NSRulerMarker.alloc).
    initWithRulerView (NSMinX (b), vertMark, hp);
m.setRemovable (YES);
hMarkers.addObject (m);
```

The code for the other edges is comparable; it is shown below only in classic syntax.

```
/* right edge */
m = [[NSRulerMarker alloc] initWithRulerView:horRuler
                                markerLocation:NSMaxX (b)
                                image:vertMark
                                imageOrigin:hp];

[m setRemovable:YES];
[hMarkers addObject:m];

/* bottom edge */
m = [[NSRulerMarker alloc] initWithRulerView:vertRuler
                                markerLocation:NSMinY (b)
                                image:horMark
                                imageOrigin:vp];

[m setRemovable:YES];
[vMarkers addObject:m];

/* top edge */
m = [[NSRulerMarker alloc] initWithRulerView:vertRuler
                                markerLocation:NSMaxY (b)
                                image:horMark
                                imageOrigin:vp];

[m setRemovable:YES];
[vMarkers addObject:m];
}
```

The View's Declarations

The main view in the window (to which the rulers are added) has a few declarations that play a role in the marker management.

```
static NSImage      *horMark = nil, *vertMark;
NSMutableArray      *hMarkers, *vMarkers;
```

The NSImages horMark and vertMark are the marker images that are loaded (if needed) in the addMarkersFor method. An NSRulerView expects to have an array of markers associated with it. Pointers to those arrays are declared here.

For easy of reading, a macro IMAGENAMED(x) is declared and invoked from addMarkersFor. The macro is shown here first in classic and then in modern syntax.

```
#define IMAGENAMED(x)
[[NSImage alloc]
 initWithContentsOfFile:
 [[NSBundle bundleForClass:[self class]] pathForResource:
 x ofType:@"tiff"]]

#define IMAGENAMED(x)
(NSImage.alloc).initWithContentsOfFile
(NSBundle.bundleForClass(self.class)).
 pathForResource(x, "tiff"));
```

This is a very common section of code (in fact, it is similar to the code that creates the markers in addMarkersFor). The steps in the process are:

1. Call the class (static) method of NSImage to create and return a new NSImage instance.
2. Call the NSImage instance's method initWithContentsOfFile. There is only one argument to this method (the file's path), although it takes three calls to get it.
3. Call the class method of self to return the Class object for self.
4. Call the NSBundle class (static) method bundleForClass using this Class object. This returns the bundle in which the current object is placed.
5. Finally, call that bundle's pathForResource with x—

the macro's variable—and the string “tiff” to locate a TIFF resource named *x*.

This code may appear dense, but it quickly gets into your fingers. You can split it into several separate lines, but it really is only one operation—locate and load something from the current class's bundle.

Note also how easy it is to organize your application and its files. Nothing requires you to load the images from the bundle in which *self* is contained: you could load the images from the *NSApp* bundle—or from any other bundle that you can identify either directly with a path or indirectly by passing in an object and finding its class bundle.

For Mac OS programmers, this is the code that you use when you want to do the equivalent of loading resources. On Mac OS the possibility of using multiple resource files is powerful, but often confusing. Originally, applications had a single resource file. As it became possible to use multiple resource files, some of the toolbox routines that relied on a single resource file being open at a time broke (or at least became somewhat fragile in actual use). The use of bundles in specific paths makes the management of external resources much easier in *Rhapsody*.

Caffeine Software

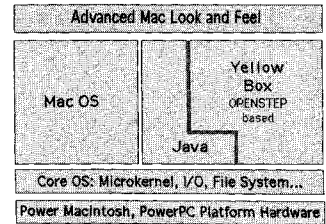
TIFFany is a product of Caffeine Software™. You can reach them at:

Caffeine Software

caffeine@cs.stanford.ed

Chapter 17


Dynamic Loading with Create



Create's sister product, TextArt, was the first independent software product to ship on the original NeXT platform. Create shipped in 1990 for OpenStep, Windows, Solaris, and Mach. Now it is available for Rhapsody.

A product that evolves over time can easily become bloated with features, slow to launch, and clumsy to use. One way to manage this problem is to add features and functionality to the product that are used only when necessary. Create's animation module is loaded—data, classes, and all—only when the user selects an animation command. Dynamic loading is one of the most powerful ways to keep Rhapsody applications efficient.

In Figure 17-1 you see a flier for Create. Note the extensive use of text placed along geometric shapes (arcs, irregular paths, etc.): these effects are notoriously difficult to produce reliably on screens and then on output media. The use of PostScript on both the printer and the display device makes this a much easier task.



Create™

Professional Desktop Publishing
for the Macintosh

Full-page and half-page spreads
with bleed-through printing
and full-page and half-page
additions and deletions for
revisions



Spell-check, grammar
check, and thesaurus
for the text

Create™
features the power of
FontShop and the power of
ColorShop to create
the most beautiful
desktop publishing
materials

ColorShop includes a
color calibration system
and automatic color
management

FontShop includes a
font management system
and automatic font
management

The Premier Drawing App for Macintosh

- Beam, Rotate, Warp
- Scale, Rotate, Warp
- Bevels & PS Patterns
- Fillers, G-Map Guides
- Unfilled & Zoned
- Guide & B-Sides
- Overlays & Layers
- Automatic Images
- Postscript 1.0
- AirMail graphics
- Flyer, Other & Pages
- Plus... it's a list itself

Full-page and half-page spreads with bleed-through printing and full-page and half-page additions and deletions for revisions



Spell-check, grammar check, and thesaurus for the text

Create™ features the power of FontShop and the power of ColorShop to create the most beautiful desktop publishing materials

ColorShop includes a color calibration system and automatic color management

FontShop includes a font management system and automatic font management

The Premier Drawing App for Macintosh

- Beam, Rotate, Warp
- Scale, Rotate, Warp
- Bevels & PS Patterns
- Fillers, G-Map Guides
- Unfilled & Zoned
- Guide & B-Sides
- Overlays & Layers
- Automatic Images
- Postscript 1.0
- AirMail graphics
- Flyer, Other & Pages
- Plus... it's a list itself

Full-page and half-page spreads with bleed-through printing and full-page and half-page additions and deletions for revisions



Spell-check, grammar check, and thesaurus for the text

Create™ features the power of FontShop and the power of ColorShop to create the most beautiful desktop publishing materials

ColorShop includes a color calibration system and automatic color management

FontShop includes a font management system and automatic font management

The Premier Drawing App for Macintosh

- Beam, Rotate, Warp
- Scale, Rotate, Warp
- Bevels & PS Patterns
- Fillers, G-Map Guides
- Unfilled & Zoned
- Guide & B-Sides
- Overlays & Layers
- Automatic Images
- Postscript 1.0
- AirMail graphics
- Flyer, Other & Pages
- Plus... it's a list itself

Full-page and half-page spreads with bleed-through printing and full-page and half-page additions and deletions for revisions



Spell-check, grammar check, and thesaurus for the text

Create™ features the power of FontShop and the power of ColorShop to create the most beautiful desktop publishing materials

ColorShop includes a color calibration system and automatic color management

FontShop includes a font management system and automatic font management

The Premier Drawing App for Macintosh

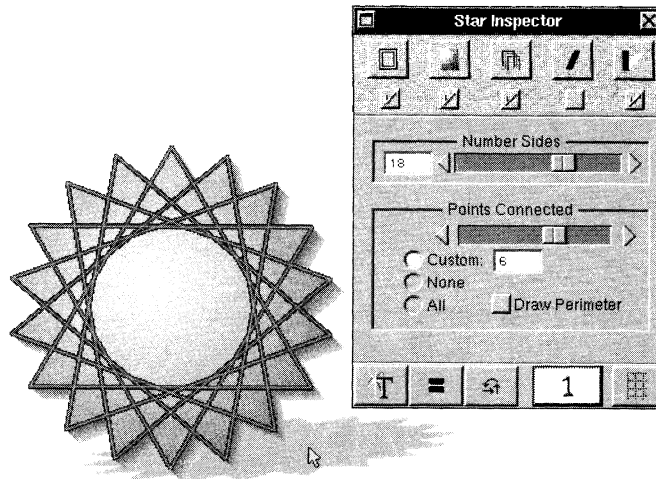



- Beam, Rotate, Warp
- Scale, Rotate, Warp
- Bevels & PS Patterns
- Fillers, G-Map Guides
- Unfilled & Zoned
- Guide & B-Sides
- Overlays & Layers
- Automatic Images
- Postscript 1.0
- AirMail graphics
- Flyer, Other & Pages
- Plus... it's a list itself

To create an animated star, first you select the Star tool from the tool bar and draw the rectangle in which you want the star

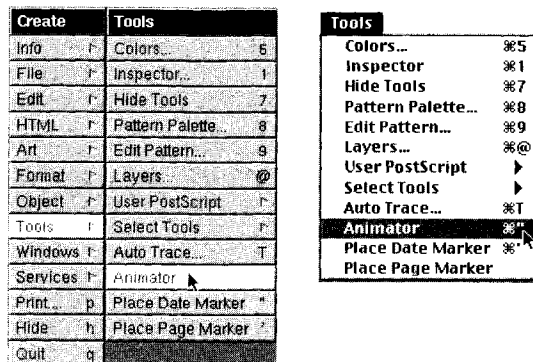
to appear. The Star Inspector panel opens as shown in Figure 17-2 to let you set the number of points, etc.

FIGURE 17-2. Star and Inspector Panel



You can go beyond this and animate the star: you choose Animator tool from the Tools menu as shown in Figure 17-3.

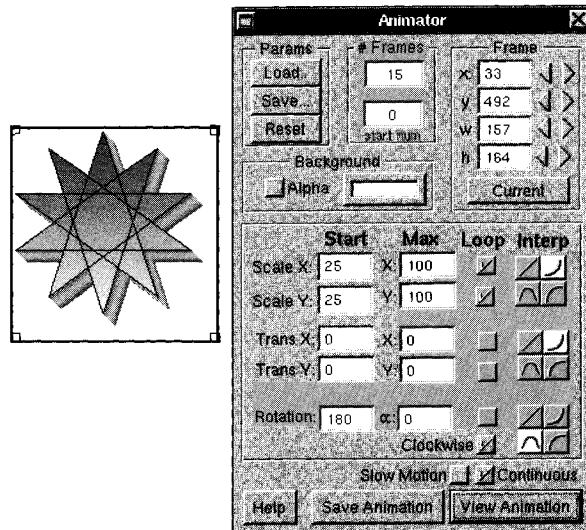
FIGURE 17-3. Tools Menu



(The menus in Figure 17-3 are shown both in classic OpenStep and in Advanced Mac Look and Feel versions.)

When you choose Animator from the Tools menu, the Animator panel opens as shown in Figure 17-4. You can specify how you want the animation to be generated, view it, and save it as you see fit.

FIGURE 17-4. Animator Panel



From the user's point of view, opening the Animator panel is no different from opening the Star panel. However, from the application's point of view, it is very different. The Animator code is loaded dynamically—as are the Animator panel and the rest of the interface.

Loading an Object and Its Code at Run Time

The procedure for loading code, objects, and resources at run time is the same no matter what is being loaded.

1. Find the path for what is being loaded.
2. Load it—as a resource (image, sound, etc.), object, or code. A bundle, which can contain objects and their associated resources, is the most complex object that you can load.

Here is the code that actually loads the Animator in Create.

First, find the path:

```
NSString *path = [[NSBundle mainBundle]
    pathForResource:@"Animator" ofType:@"bundle"]; // classic

NSString *path = (NSBundle.mainBundle).
    pathForResource ("Animator", "bundle"); //modern
```

You use the mainBundle method of the NSBundle class to return the bundle for the application that is running. Having gotten it, you then call that bundle's pathForResource method with the title of the bundle ("Animator") and the type of resource ("bundle") that you're interested in. As long as you know the name and type of what you're loading, you're OK).

Next, you use the path to load whatever you're interested in:

```
NSBundle *b = [[NSBundle alloc] initWithPath:path]; //classic

NSBundle *b = (NSBundle.alloc).initWithPath (path); //modern
```

The NSBundle class method alloc creates and returns a new bundle object which you then initialize using the path that you created.

Bundles are complex structures, containing resources and objects. Most bundles have a principal class which you can ex-

tract using the `principalClass` accessor. In order to convert the class from the bundle into a fully instantiated object, you call `sharedInstance` to reconstitute it:

```
animator = [[b principalClass] sharedInstance]; //classic  
  
animator = (b.principalClass).sharedInstance); //modern
```

You now have a real animator object and can call its methods; in this case, you want to pull the Animator panel forward:

```
[animator orderFront:self]; //classic  
  
animator.orderFront (self); //modern
```

(The Animator panel is an `NSWindow`; `orderFront` is an inherited method of `NSWindow`.)

When to Load the Animator

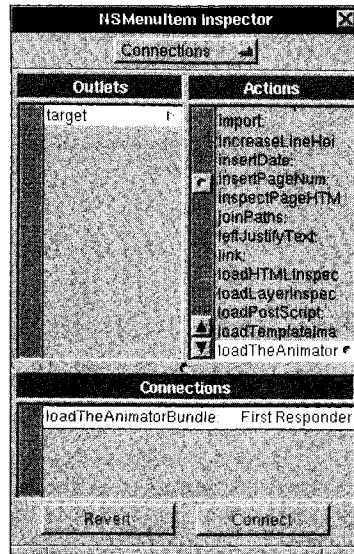
The remaining piece of the puzzle is deciding when to load the Animator. There are two considerations here:

1. Do it when the user wants it.
2. If it has already been loaded, don't bother.

In Create, the `NSApplication` object is subclassed as an `App-TextArt` object. It has a single method that manages both of these chores: the method is called `loadTheAnimatorBundle`.

Letting the user control the Animator panel is as simple as wiring the Animator menu command to this method. When the user chooses Animator from the Tools menu, the `App-TextArt` object's `loadTheAnimatorBundle` method is called.

Using Interface Builder, you program this by connecting the target of the menu command to the `loadTheAnimatorBundle` action, as shown in Figure 17-5.

FIGURE 17-5. Interface Builder for Create

Within `loadTheAnimatorBundle`, a static animator instance variable is declared as `nil` (in the italicized line of code). Static variables such as `animator` are set only if they have not been set before; the second time this method is called, `animator` will have been set (in the boldface line of code), and it is not reset to `nil`.

The `loadTheAnimatorBundle` method checks to see if `animator` is `nil`; if it is, it is loaded using the code described above. After loading, the animator is then pulled to the front. The full code (in classic syntax) is presented here.

```
- loadTheAnimatorBundle:sender
{
    static id animator = nil;
    // If the bundle is not loaded, go find it!
    if (!animator)
    {
        NSString *path = [[NSBundle mainBundle]
```

```
        pathForResource:@"Animator" ofType:@"bundle"];
    if (path)
    {
        NSBundle *b = [[NSBundle alloc] initWithPath:path];

        // if the bundle is valid, find the principal class
        // and create the single instance of the Animator...

        if ((b != NULL) && ([b principalClass] != NULL))
        {
            animator = [[b principalClass] sharedInstance];
            NSLog(@"Now loading %@...\n", path);
        }
        else
            NSLog(@"Can't Load %@!\n", path);
    }
    else
        NSLog(@"Couldn't find Animator bundle!\n");
}

// Order the animator's panel to the front:
if (animator)
    [animator orderFront:self];
else
    NSLog(@"Couldn't load Animator!\n");

return self;
}
```

Stone Design

Create is a product of Stone Design. You can reach them at:

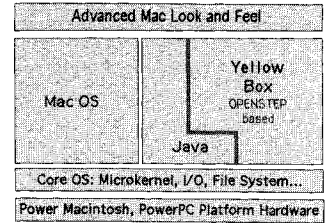
Stone Design
2400 Rio Grande Boulevard, Suite 116
Albuquerque, NM 87104

(505) 345-4800
(505) 345-3424 (fax)

<http://www.stone.com>
info@stone.com

Chapter 18

Providing a Service and Using Distributed Objects with CHaRTSMITH



Mac OS supports the clipboard, which lets users put data of various types into its shared area so that they can use the same or different applications to remove it and act on it in different contexts. Rhapsody's services use its pasteboard to transfer data back and forth between a service and a client application (the transfer is often bi-directional, rather than uni-directional as on the Mac OS). The Services menu is maintained by the system and enables its various items depending on what is selected in a user's document. The data type(s) selected and the data type(s) recognized by each service are compared to determine what services are available at that moment. (All of this is done automatically, as is the start-up processing that takes an inventory of the available services in the user's environment.)

Services are often two-way streets: a spell-checking service takes text, spell checks it, and then returns corrected text. The services API defines both sending and receiving types, although often the receiving type is ignored.

Opportunities of two sorts are provided by this architecture. On the one hand, developers can provide services—typically lightweight, specialized applications that do one or two things very well. On the other hand, developers of applications can eliminate certain functions that they either don't do well or don't choose to do and they can rely on services to provide these functions.

In this chapter, you will see how a service is provided to other applications. You will also see how a closer tie can be forged using distributed objects: the service can be incorporated into the main application with the user's intervention.

CHaRTSMITH

CHaRTSMITH is a perfect example of a service. It does one thing—charts and graphs—very well. Furthermore that one thing is something that many applications ranging from spreadsheets to word processors to e-mail applications and to accounting software sometimes need to do. For some users.

In the world of traditional applications, this functionality is often built into specialized or “professional” versions of common applications. This makes a certain amount of sense (why should every word processor user have sophisticated charting if only a few users need the feature?), but the logic quickly falls apart as soon as you realize that just about every user needs one of the “professional” features at some time or another. So the historian who vitally needs the ability to put video interviews into scholarly documents is stuck with the cockamamie charting nonsense—and the economist who can't live without the sophistication of bubble, pie, high-low, and pie charts is equally stuck with the irrelevant (economically) video editing features.

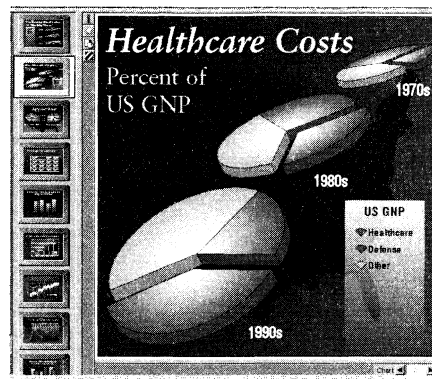
When charting is a service, it is available to any application that can export strings—the application need not know that it is exporting data to be charted. All the application need do is register its ability to export strings and the Services menu will enable the CHaRTSMITH submenu. It is the user who decides to use charting. The application's hook works equally well with other services that handle strings. This is exactly the type of structure that provides users with the power and functionality that they want (without buying enormous, expensive, and duplicative software products).

Of course, for some applications, charting is not a rare occurrence for many users. In these cases, a tighter integration is available in which the application does know about the service and integrates its objects directly into its own application.

CHaRTSMITH at Work

Figure 18-1 shows the main CHaRTSMITH window.

FIGURE 18-1. CHaRTSMITH Screen Shot



It is charting data as exploded pie charts—the other charts in this document are shown in a scrolling list at the left, and a user can quickly switch from one chart type to another. CHaRTSMITH allows you to annotate charts with text, lines, comment balloons, and other objects. It allows you to create

special effects (such as the three-dimensional images shown in the figure) as well as to create and use templates for a standard look to a series of charts.

CHaRTSMITH also exports its finished results as images that can be pasted back in the original application or that can be used in another environment.

Providing a Service

There are three steps to providing a service:

1. Setting up the Services menu
2. Registering the service when the service application starts up
3. Implementing the functionality

Setting up the Services Menu

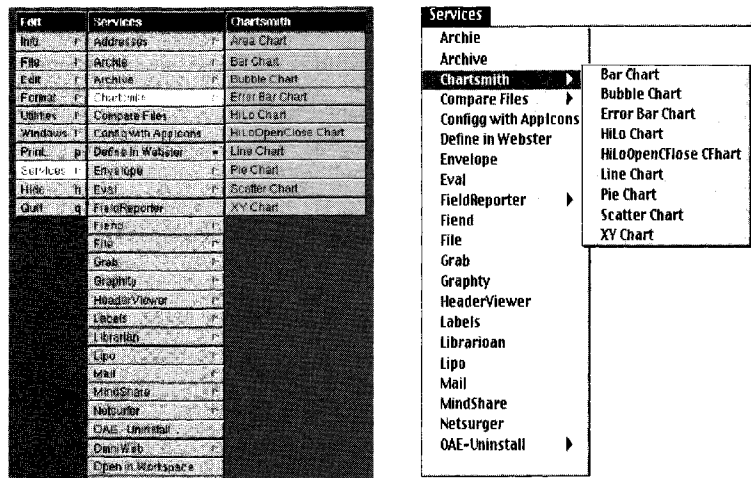
CHaRTSMITH is installed automatically as part of the Services menu when it is found at system start-up. When you select text that can be charted, its submenu is enabled, as shown in Figure 18-2. The Services menu varies from one user's environment to another; this user has a number of services installed. You can see the CHaRTSMITH services that are available when text is selected in a document from any application. On the left of Figure 18-2 is the OpenStep look of the Services menu; on the right is one preliminary version of the Services menu in Rhapsody's user experience.

A user can choose the chart to be created; CHaRTSMITH will be launched (if necessary), will open a window, and will prepare the chart.

The Services menu is constructed from each service's property list. One of the properties is the NSServices property; it is assembled from the ASCII text into a dictionary. (In fact, the

presence of an NSServices property is what makes an application eligible to be included in the Services menu.)

FIGURE 18-2. Services Menu Showing Enabled CHARTSMITH Submenu



NSServices Like all dictionaries, it has pairs of keys and values. The keys and their values along with their meanings are:

- `NSMessage` = `methodName`; This is the name of the method that is to be invoked in the service application.
- `NSPortName` = `programName`; The name of the service application.
- `NSSendTypes` = (`type1` [, `type2`] ...); The framework type constant(s) that the service will receive (that an application will send—the naming is application- not service-centric).
- `NSReturnTypes` = (`type1` [, `type2`] ...); The framework type constant(s) that the service will return.
- `NSMenuItem` = `default` = `item`; The menu item to appear in the Services menu; if a submenu is used, the

commands are separated by a slash—see the example that follows.

- `NSKeyEquivalent` = default = character; Key equivalent for the menu item.
- `NSUserData` = string; A string that is passed to `NSMessage` (the method invoked) when this Services menu command is selected. This is a constant string; the selected data in the pasteboard is variable.
- `NSTimeOut` = milliseconds; How long the application should wait for a response from the service (default is 3000 milliseconds).
- `NSHost` = hostname; If provided, the name of a host on the network where the service should be launched.
- `NSExecutable` = path; An optional entry describing the path for the service to use.

CHaRTSMITH's NSServices Property List Here is the start of the property list for CHaRTSMITH. The first three Chartsmith Service menu commands are shown here (Area Chart, Bar Chart, and Bubble Chart). In each case, if the user selects the command, the `chartFromService` method will be invoked; it will be passed a string (Area, Bar, or Bubble) depending on which command was chosen. In all cases, the menu commands will be enabled when string text is selected, and in all cases the CHaRTSMITH application will be used.

```
NSServices = (  
    NSMessage = chartFromService;  
    NSPortName = Chartsmith;  
    NSSendTypes = (NSStringPboardType);  
    NSMenuItem = default = "Chartsmith/Area Chart";  
    NSUserData = "Area";  
    ,  
    NSMessage = chartFromService;  
    NSPortName = Chartsmith;  
    NSSendTypes = (NSStringPboardType);  
    NSMenuItem = default = "Chartsmith/Bar Chart";  
    NSUserData = "Bar";  
    ,  
    NSMessage = chartFromService;
```

```

NSPortName = Chartsmith;
NSSendTypes = (NSStringPboardType);
NSMenuItem = default = "Chartsmith/Bubble Chart";
NSUserData = "Bubble";
...

```

Registering the Service at Start-up

The method that is to be called from the Services menu is a method of an object that you select as a service provider within your application. At start-up, you call `NSApp`'s `setServicesProvider` method and pass in that object. CHARTSMITH does that with the following code (in classic and modern syntax):

```

-(void) applicationDidFinishLaunching:
    (NSNotification *)notification
{
    ServiceProvider *servicesProvider =
        [[ServiceProvider alloc]]init;
    [NSApp setServicesProvider:servicesProvider];
}

(void) applicationDidFinishLaunching
    (NSNotification* notification)
{
    ServiceProvider *servicesProvider =
        (ServiceProvider.alloc()).init();
    NSApp.setServicesProvider (servicesProvider);
}

```

The `ServiceProvider` class may implement a class method `alloc`, and an instance method `init` (both of which are used to allocate and initialize the object); if it doesn't have local variables, it can use its superclass's `alloc` and `init` methods. It must also implement the method `chartFromService`, since that is the method that will be called when a Services menu command is chosen (as specified in the `NSServices` property list).

Implementing the Functionality

The method that you provide must have three arguments:

1. `pasteBoard` (`NSPasteboard*`) is the pasteboard containing the data selected by the user.
2. `data` (`NSString*`) is the string that you specified in the `NSServices` property list key `NSUserData`. In the case

shown here, it is used to distinguish among several different commands that use the same dispatching method.

3. `error (NSString*)` is a string that you return.

Note that while CHaRTSMITH uses one method (`chartFromService`) to dispatch several menu commands, it is equally possible to have several methods within your services provider object and let each method handle a single Services menu command. The name of the method must be that of the NS-Message property for the appropriate NSMenuItem.

Here is the `chartFromService` method:

Classic Syntax

```
-(void) chartFromService: (NSPasteboard*) pasteboard
    userData: (NSString*)data
    error: (NSString*) error
{
    NSArray *types;
    types = [pasteboard types];

    /* confirm that correct data type was sent on
    Service pasteboard */
    if (![types containsObject:NSStringPboardType])
        *error = NSLocalizedString (@Pasteboard Error:
            Chartsmith is unable to read Services
            pasteboard data", @"pasteboard not string type");
    return;

    /* request document handler to create new chart using
    Service pasteboard data */
    [documentHandler addChartOfType: data
        fromServicePasteboard: pasteboard];
}
```

Modern Syntax

```
-(void) chartFromService: (NSPasteboard* pasteboard,
    NSString* data, NSString* error)
{
    NSArray *types;
    types = pasteboard.types ();
}
```

```

/* confirm that correct data type was sent on
Service pasteboard */
if (!(types.containsObject(NSStringPboardType))
    *error = NSLocalizedString (@Pasteboard Error:
        Chartsmith is unable to read Services
        pasteboard data", @"pasteboard not string type");
    return;

/* request document handler to create new chart using
Service pasteboard data */
documentHandler.addChartOfType (data, pasteboard);
}

```

All that it does is to retrieve the types from the pasteboard, check that one of them is an `NSStringPboardType`, and then (in the boldface line of code) ask the `CHARTSMITH` documentHandler (an internal method) to add a chart of type data (the constant string set in the `NSServices` property list) using the data on the pasteboard.

That's all it takes to provide a service.

Using Distributed Objects

Rather than going through the Services menu, with distributed objects, you can treat the other application's objects as if they were your own. This allows tighter integration and less user involvement—but of course it also means that the primary application needs to know about the other application's objects and data structures.

Exposing your application's functionality as Services (almost always a good idea and a source of added revenues as people use your work for things you've never dreamed of) requires you to factor your application so that the interface (commands and preparation of data) is separate from the actions taken on the data. This structure serves you in good stead if you decide to implement distributed objects. `CHARTSMITH` does both.

Whereas the objective of the services implementation was to get the method `chartFromService` to be called, the objective of the distributed objects implementation is to let a client application directly call CHaRTSMITH's `newChart` method. (There are other aspects to its implementation of distributed objects, but this is the one discussed here.) The `newChart` method is a method of a CHaRTSMITH document object, and CHaRTSMITH has a method called `currentDocument` that returns such a document.

Specifically, a client application can write the following method. The two key lines are highlighted: in *italics*, you see the line of code that gets the current document; in **boldface**, that document's `newChart` method is called. (`connectionToChartsmith` is described in the following section; for now, just think of it as CHaRTSMITH.)

Classic Syntax

```
- (void) createNewChartWithData: (NSString*)dataString and-
StoreInFile: (NSFileHandle*)fileHandle
{
    id aDocument;
    id aChart;
    NSData* chartPostScriptData;

    /* grab the current document in the Chartsmith application
    and create a chart using the dataString data*/
    aDocument = [connectionToChartsmith currentDocument];
    aChart = [aDocument newChart:BAR
        fromData: dataString
        hasCollLabels: NO
        hasRowLabels: NO
        useDefaultTemplate: YES];
    /* request the PostScript for the chart,
    then save it to the file */
    [aChart getPSData: chartPostScriptData];
    NS_DURING
        [fileHandle writeData:chartPostScriptData];
    NS_HANDLER
        NSLog(@"Failed to save chart to file.");
    NS_ENDHANDLER
}
```


Modern Syntax

```

- (void) createNewChartWithData: (NSString* dataString ,
    NSFileHandle* fileHandle)
{
    id aDocument;
    id aChart;
    NSData* chartPostScriptData;

    /* grab the current document in the Chartsmith application
    and create a chart using the dataString data*/
    aDocument = connectionToChartsmith.currentDocument ();
    aChart = aDocument.newChart (BAR, dataString, NO, NO, YES);

    /* request the PostScript for the chart,
    then save it to the file */
    aChart.getPSData (chartPostScriptData);
    NS_DURING
        fileHandle.writeData (chartPostScriptData);
    NS_HANDLER
        NSLog (@@"Failed to save chart to file.");
    NS_ENDHANDLER
}

```

Note the use of the error handler at the end of this method: writing to a file is an operation that typically can fail and that should be protected in this way.

Making a Connection (from the Client)

In the code shown previously, the `currentDocument` method in `CHARTSMITH` is invoked as a method of `connectionToChartsmith`. In fact, `connectionToChartsmith` is an instance of `NSConnection` that is created when the client starts up. It is the connection to a distributed object in `CHARTSMITH` (an object of which `currentDocument` is a method).

In order to make a connection to a distributed object, you need to know some information about the object to which you are connecting. This information is provided by the developer of that object as part of the API. The information consists of:

- The name under which the connection will be registered (in this case it will be `ChartsmithAPI`).

- The methods of the object that you can call (sort of necessary if you're going to be using them!).
- A protocol that the distributed object implements. This last is not essential, but it provides efficiency improvements. Before actually dispatching method calls to distributed objects, the system confirms that the method can be handled. This makes for an extra round-trip before each call; if you can group all of the methods into a protocol, you can tell your end of the connection (the client end) that that protocol is supported, and for method calls within that protocol, this extra round-trip is eliminated.

Here is the code for a client to establish a connection to CHaRTSMITH. It is called once (usually at start up), and establishes the `connectionToChartsmith` object which is then used throughout the application.

Classic Syntax

```
-(void)connectToChartsmith: sender
{
    NSConnection *connectionToChartsmith;

    /* get the root object for the Chartsmith server */
    connectionToChartsmith = [NSConnection
        rootProxyForConnectionWithRegisteredName:
        [NSString stringWithCString:"ChartsmithAPI"
        host:[NSString stringWithCString:""]];

    if (!connectionToChartsmith)
        /* error handling */

    /* set the protocol for the root object to reduce
    confirmation overhead */
    [connectionToChartsmith setProtocolForProxy:
        @protocol(CSAppAPI)];
}
```

Modern Syntax

```
-(void)connectToChartsmith (id sender)
{
    NSConnection *connectionToChartsmith;
```

```

/* get the root object for the Chartsmith server */
connectionToChartsmith = NSConnection
    .rootProxyForConnectionWithRegisteredName
    (NSString stringWithCString("ChartsmithAPI"),
     NSString stringWithCString(""));

if (!connectionToChartsmith)
    /* error handling */

/* set the protocol for the root object to reduce
confirmation overhead */
connectionToChartsmith.setProtocolForProxy(
    @protocol(CSAppAPI));
}

```

The `connectionToChartsmith` now contains a connection to the application that was registered with the name `ChartsmithAPI`; furthermore, it has a reference to a root object that is a proxy for an object within `CHaRTSMITH`. (The proxy is within the client, the real object is within `CHaRTSMITH`; messages sent to the proxy are in effect sent to the real object through the connection.)

Allowing a Connection (from the Server)

In order to enable this from the server side, you must create your own connection (the counterpart to the connection that will be established from the client), register a name (the same name that will be used by the client—here `ChartsmithAPI`), and set a root object (which will be the real object that the client connection's proxy refers to).

The `DOHandler` is a `CHaRTSMITH` object that handles the distributed object calls (such as `currentDocument` in the previous code). It can be any object that you want and must implement the methods that you will allow others to use. As noted previously, it should group those methods (or most of them) into a protocol that can be set by clients to minimize error checking.

All of this is done in `CHaRTSMITH` with the following code:

Classic Syntax

```
NSConnection* chartsmithConnection; /* server connection */

/* Chartsmith root object for DO messaging */
DOHandler *dOHandler = [[DOHandler alloc] init];

/* get application's NSConnection instance, set handler
as root object, and register connection name */
chartsmithConnection = [NSConnection defaultConnection];
[chartsmithConnection setRootObject:dOHandler];
[chartsmithConnection setDelegate:dOHandler];
if (![chartsmithConnection registerName :
    [NSString stringWithCString:"ChartsmithAPI"]])
    /* error handling */
```

Modern Syntax

```
NSConnection* chartsmithConnection; /* server connection */

/* Chartsmith root object for DO messaging */
DOHandler *dOHandler = (DOHandler.alloc).init;

/* get application's NSConnection instance, set handler
as root object, and register connection name */
chartsmithConnection = NSConnection.defaultConnection ();
chartsmithConnection.setRootObject (dOHandler);
chartsmithConnection.setDelegate (dOHandler);
if (![chartsmithConnection.registerName (
    NSString.stringWithCString("ChartsmithAPI"))])
    /* error handling */
```

Symmetry of Distributed Objects

In order to use distributed objects, both client and server must have an NSConnection. These are two separate objects (one in the client's thread, one in the server's), but they are united by the use of a common name.

Each connection object has a root object. On the server side, this is a real object (the DOHandler object in the code previously shown); on the client side, the root object in its connection is a proxy for the real object. Once the connection is established, you can call methods of the real object by calling "methods" of your connection.

CHaRTSMITH

CHaRTSMITH is a product of BlackSmith, which has also developed QUeRYSMITH, SLiDESMITH, and DaTASCRIBE. They can be reached at:

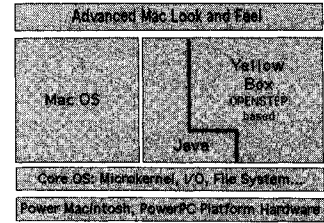
BlackSmith
1651 Old Meadow Road, Suite 325
McLean, VA 22102

(888) BLKSMTH
(703) 847-8180
(703) 847-5767 (fax)

<http://www.blacksmith.com>
info@blacksmith.com

Chapter 19

Using Services with OmniWeb



OmniWeb2 is a World Wide Web browser written originally for NextStep and OpenStep; it is now available on Rhapsody. The Internet and its various protocols (Web, e-mail, Telnet, FTP, etc.) are made available to people using three types of architectures:

- *You can use separate application programs for each Internet protocol. This means that you have to maintain and synchronize a number of separate programs; it also means that the Internet experience is fragmented into separate operations.*
- *You can use a single “monster” Internet application that can handle all protocols. Like any large application, this is*

likely to be constantly in a state of flux; furthermore, while some parts of the application (typically the Web browser) may be quite good, other parts (e-mail, for example) are not so good as specially dedicated products.

- *You can use a suite of cooperating Internet applications that are tied together with a common architecture. (This is the approach that Cyberdog uses on Mac OS.) OmniWeb, though a single application, allows you to replace its parts with cooperating applications; furthermore, it implements some functionalities using loadable bundles for Gopher, GIFs, and other features. In this chapter, you will see how a loadable bundle lets OmniWeb use an external application for news reading.*

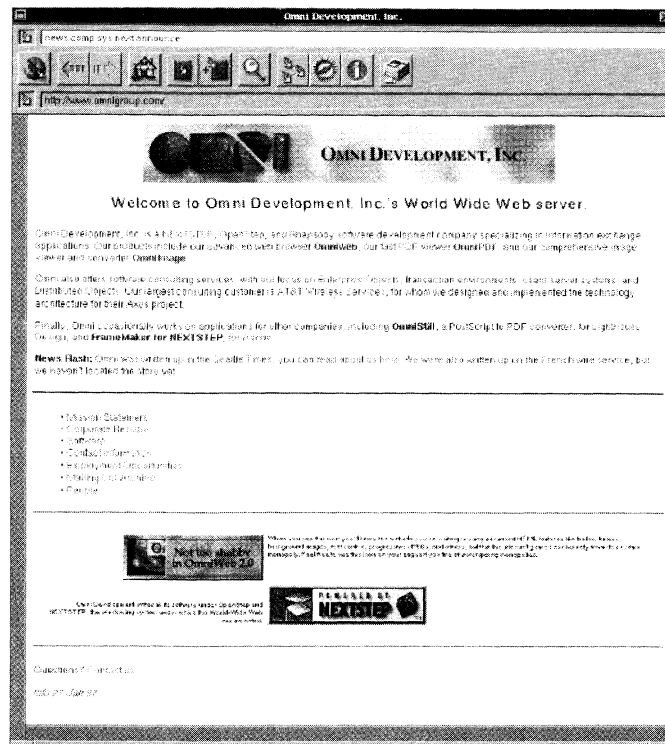
OmniWeb does not implement all of the Internet protocols. It implements two (http and ftp) and does one other thing very well. The other thing that it does is to identify URLs based on their scheme and to dispatch them to appropriate other cooperating applications for Telnet, news, and mail. This is done with the Services architecture and is applicable to many other cases. Whether the dispatching via the Services menu is done in response to user requests (such as when a user selects some text and spell checks it using a service) or automatically (as when OmniWeb opens a news reader), the structure is the same.

Whereas the previous chapter demonstrated how an application can provide a service, here you see OmniWeb function as the user of a service. You will also see how an application can use a customized property list to store preferences and parameters for itself, much as the standard NSServices property was used in the last chapter.

OmniWeb

In Figure 19-1, you see the OmniWeb browser. It displays part of the Omni Development home page.

FIGURE 19-1. OmniWeb Browser



OmniWeb's architecture relies on pipeline objects. Its pipelines are composed of a series of processors—each of which can be substituted at run time. A series of three processors displays a Web page, for example:

1. The protocol processor converts the URL into a stream of bytes using the HTTP network protocol.

2. A second processor converts the raw stream of bytes to structured HTML objects (tags, text, and embedded images).
3. A third processor converts HTML to the layout displayed in a window.

This architecture is easily maintainable in the world of objects: the first process (the stream of bytes processor) interacts with the flat, nonobject world of the Internet. Everything beyond that is object oriented.

Furthermore, it is easy to replace any component of the system without doing harm to any other. Adding new HTML tags should not affect the byte stream; a change to the byte stream (possibly to reflect efficiencies in low-level communication or buffering) shouldn't affect the final imaging of the data. Anyone who has looked at the source code of one of the major (non-object-oriented) commercial browsers will appreciate this simplicity. The well-documented difficulties in modifying those browsers attest to the advantages of this type of architecture.

Still, it is necessary to provide access to any number of Internet protocols from within a Web browser. In the URL entry field, the user can type the URL of a news group; as soon as return is pressed, somehow OmniWeb must display that news group—something that it doesn't do on its own.

OmniWeb Services

Services are used in two ways in Rhapsody. In the first model, two applications are involved: one is the main application in which the user is working, and the second is the service application. Their communication is under direct control of the user. When data is selected in a view or window, that view is responsible for registering the type of data with the Services menu. Meanwhile, each application in the Services menu has registered the types of data that it can send and receive.

Where there's a match, the relevant Services menu commands are enabled.

This model is based on the main application putting the selected data on a pasteboard, passing it (in response to the user command) to the service, and then receiving it back from the service on the pasteboard and restoring it to its place in the document. Services that check spelling, for example, send and receive strings—which may differ in that the received string contains correctly spelled words.

OmniWeb uses the basic service architecture, but it is customized for its purposes. In particular, OmniWeb does not export and import data that has been modified by a service: OmniWeb uses the service architecture to launch specific helper applications to handle protocols that it does not provide. There is no return of data from the service to the main application. (Since it is not returning data, the service can in fact be a standard application—as it is here.)

Adding a News Reader Service to OmniWeb

In the previous chapters, you have seen examples of loading resources and objects from bundles—in particular, from an application's main bundle, from the bundle in which a particular class object is located, and from a bundle with a specific name. OmniWeb uses bundles in yet another way: it checks the property lists of all the application's bundles and stores the properties it cares about (called OmniRegistration) so that it can later load the bundles as needed.

In OmniWeb, one such bundle contains the OmniServicesMenuProcessor, a subclass of OmniProcessor which is the basic component of the pipeline of processors outlined previously in this chapter.

OmniServicesMenuProcessor does the following:

- It initializes and deallocates itself.
- It overrides the OmniProcessor process method to do the work. In this case, it will delegate the work to an external application as a service.
- A final method does the dispatching of the service.

The code is general: although the example given here is for a news reader, it can equally well be used for other Internet services not supported by OmniWeb. Furthermore, these values can be changed by users.

Registering New Functionality

Each added bundle has a property list in CustomInfo.plist. This is a structure controlled by the developer and which users can modify at will. A single OmniRegistration for the News Reader service looks like this:

```
{
OmniRegistrations=
{
    OmniProcessor=
    {
        converts =
        (
            {
                input = "url/news";
                output = "omni/source";
            }
        );
    };
};

OmniPreferenceController=
{
    OmniServicesMenuPreferences=
    {
        defaultsDictionary=
        {
            ServiceForNews = "Alexandra/OpenURL";
        };
    };
};
}
```

There are several sections to an OmniRegistration; two relevant ones are shown here. The OmniProcessor identifies the data conversion that this class can handle (here it is from a news URL to standard Omni format); the OmniPreferenceController identifies the external application that does the work. In this case, Alexandra is a news reader application. (Note that the structure of the OmniProcessor allows both input and output to be specified; in the news reader example shown here, the output is displayed by the news reader application and the output in OmniProcessor is ignored.)

The boldface line with the keyword `ServiceForNews` can be changed by the user to select another news reader.

The Code

There are four methods to `OmniServicesMenuProcessor`:

1. `initWithPipeline`
2. `dealloc`
3. `process`
4. `processService`

The code to load the bundle containing the `OmniServicesMenuProcessor` and the `OmniRegistrations` property list for that bundle is in the main application. Bundle-loading is no different in this case than in the previous examples.

Within this code, you need assume only two external architectural features:

1. There is a class called `OWURL` (OmniWeb URL) that contains a URL and is capable of presenting its URL and its scheme on request. (The scheme is the part of the URL to the left of the colon: `http`, `mailto`, `ftp`, etc.)
2. Processes are part of pipelines (as described previously in this chapter).

initWithPipeline This method is implemented in its own way for every processor. A pipeline object is passed into the `init` method; it is placed in the instance variable `pipeline`, from whence it is used later in the `initWithPipeline` method. The code follows in both classic and modern syntax:

```
/* Every processor is instantiated with the initWithPipeline:
call. The pipeline object contains the series of processors and
content which is being generated from the initial type to get
to the final content type. In this case, we save the URL which
is the last content in the pipeline, and then add nothing to
the pipeline and tell it to start processing. This informs the
pipeline that this processor doesn't have any output to be dis-
played or processed further. */
```

Classic Syntax

```
- initWithPipeline:(OmniPipeline *)aPipeline
{
    if (![super initWithPipeline:aPipeline])
        return nil;
    /* Remember the URL we're handling */
    dstUrl = [(OWAddress *) [pipeline lastContent] url];
    [dstUrl retain];

    /* We don't produce any real information (inside OmniWeb,
    anyway) */

    [pipeline addContent:nil];
    [pipeline startProcessingContent];

    return self;
}
```

Modern Syntax

```
initWithPipeline:(OmniPipeline *)aPipeline
{
    if (![super initWithPipeline:aPipeline])
        return nil;
    /* Remember the URL we're handling */
    dstUrl = (OWAddress *) (pipeline.lastContent.url);
    [dstUrl retain];

    /* We don't produce any real information (inside OmniWeb,
    anyway) */
```

```

pipeline.addContent (nil);
pipeline.startProcessingContent ();

return self;
}

```

This is a common initialization routine; in cases such as this in which subclasses implement a common initialization routine, the first line is almost always a call to the superclass's initialization routine. Following that, the URL is obtained from the pipeline object. Note the boldface line that retains the URL. This is also a very common case: a reference to an object created elsewhere is important to this object. Since there is no guarantee that the pipeline object won't deallocate the URL object, it is essential to retain it here so that it remains in existence as long as this object needs it.

Which is why a dealloc method is necessary.

dealloc The dealloc method releases the object that was retained in the initialization routine and calls the superclass's dealloc method. If nothing had been retained in the initialization method, there would be no reason to implement the dealloc method since the superclass's method would be called by default. (The superclass itself in this case should retain and release the pipeline object that it copied into its local pipeline variable.)

Classic Syntax

```

- (void)dealloc
{
    [dstUrl release];
    [super dealloc];
}

```

Modern Syntax

```

void (dealloc)
{
    dstUrl.release ();
    super dealloc ();
}

```

process Each processor object has a process method, which is called as the pipeline processes data. All that this method does is to construct the appropriate key in the defaultsDictionary in the OmniPreferenceController and then retrieve the associated value. In following the code, the following may be helpful (refer back to the OmniRegistrations property list):

- The boldface line of code is attempting to construct the string `ServiceForNews` based on the constant `ServiceFor` and the scheme extracted from the URL (News). The scheme is first translated to lowercase and then capitalized—news, then News. Applied to HTTP, the conversion would be to http and then Http. Since dictionary keys are case sensitive, this is necessary.
- The italicized line of code is attempting to use the key to obtain the value `Alexandra/Open URL`.

Finally, process calls the perform service method, which is shown next.

```
/* Process is the method which needs to actually do the conversion between content types. In this case, we look up the name of the application that this particular user wants to use for news urls as a user default named "ServiceForNews". Note that this code is generic in that you could add a "ServiceForFoo" default that would cause foo: URLs to be passed to the application of your choice - no code change is required, just additions to the information file shown below. Note that each processor performs its work in a separate thread, so once we look up the service name to invoke from the user's defaults, we need to send a message to be performed in the main thread to do the actual communication with the other application. (At the moment, the OpenStep AppKit is not thread-safe.) */
```

Classic Syntax

```
- (void)process
{
    NSString *defaultKey, *serviceName;

    /* Figure out the name of the service to invoke */

    defaultKey = [@"ServiceFor" stringByAppendingString:
        [[[dstUrl scheme] lowercaseString]
```



```

        capitalizedString]];
serviceName = [[OmniUserDefaults sharedUserDefaults]
               stringForKey:defaultKey];

[self setStatusFormat:@"Invoking \"%@\\" ...", serviceName];
[pipeline processorStatusChanged:self];

/* Appkit stuff is not generally thread-safe. */
[self mainThreadPerformSelector:@selector
    (performService:) withObject:serviceName];
}

```

Modern Syntax

```

(void)process
{
    NSString *defaultKey, *serviceName;

    /* Figure out the name of the service to invoke */

    defaultKey = "ServiceFor";
    defaultKey.stringByAppendingString
        ((dstUrl.scheme()).lowercaseString).capitalizedString);
    serviceName = (mniUserDefaults.sharedUserDefaults()).
        (stringForKey (defaultKey));

    self.setStatusFormat (@"Invoking \"%@\\" ...", serviceName);
    pipeline.processorStatusChanged (self);

    /* Appkit stuff is not generally thread-safe. */
    self.mainThreadPerformSelect (:@selector
        (performService:) , serviceName);
}

```

performService Communication with services is via a pasteboard. The `OmniServicesMenuProcessor` declares a static (shared) pasteboard for all of its requests and explains the rationale:

```

/* Unfortunately, it's not possible to simply create a paste-
board for each request, since with an synchronous request the
appkit might use the pasteboard long after we return from the
NSPerformService() call (eg. if it has to launch the application
before sending the message). To avoid leaking pasteboards we
just use one shared pasteboard. */

```

```

static Pasteboard *sharedPb = nil;

```

When it comes time to actually perform the service, that pasteboard is created (if necessary) and is populated with the custom `W3URIPboardType` and the `NSStringPboardType` and with the URL's string representation. (The `compositeString` method of the `OWURL` returns that string.) The italicized lines are these standard pasteboard calls: initialization, setting of types, and setting of values.

The boldface line dispatches the service (passed into the method as an argument); the pasteboard is then read by the service.

Here is the code:

Classic Syntax

```
/* This is the method invoked in the main thread by the
process call. It gets a globally accessible pasteboard and
pasteboard types: we prefer other applications to treat
the pasteboard data as a WWW URI, but it can also be treated
as a simple NSString. The NSPerformService() call does
the real work. It starts up the destination application, if
necessary, and then calls that application, passing it
the pasteboard with our data. */

- (void)performService:(NSString *)serviceName
{
    NSPasteboard          *pb;
    NSArray               *pbTypes;
    NSString              *W3URIPboardType;
    BOOL                  success;

    /* Shared pasteboard (sigh!) */
    if (!sharedPb)
        sharedPb = [Pasteboard newUnique];
    pb = sharedPb;

    W3URIPboardType = [OWAddress W3URIPboardType];

    /* Pasteboard types, in decreasing order of spiffiness */

    pbTypes = [NSArray arrayWithObjects:W3URIPboardType,
        NSStringPboardType, nil];

    [pb declareTypes:pbTypes owner:nil];
    [pb setString:[dstUrl compositeString]
```

```

        forType:W3URIPboardType];
[pb setString:[dstUrl compositeString]
 forType:NSStringPboardType];

/* O, Mighty Service of the Appkit, I Invoke Thee! */
success = NSPerformService(serviceName, pb);

if (!success) [pipeline processor:self hasError:
 [NSString stringWithFormat:@"Error invoking service\"
 %@\".", serviceName]];
}

```

Modern Syntax

```

/* This is the method invoked in the main thread by the
process call. It gets a globally accessible pasteboard and
pasteboard types: we prefer other applications to treat
the pasteboard data as a WWW URI, but it can also be treated
as a simple NSString. The NSPerformService() call does
the real work. It starts up the destination application, if
necessary, and then calls that application, passing it
the pasteboard with our data. */

(void)performService:(NSString *)serviceName
{
    NSPasteboard          *pb;
    NSArray               *pbTypes;
    NSString              *W3URIPboardType;
    BOOL                  success;

    /* Shared pasteboard (sigh!) */
    if (!sharedPb)
        sharedPb = Pasteboard.newUnique ();
    pb = sharedPb;

    W3URIPboardType = OWAddress (W3URIPboardType);

    /* Pasteboard types, in decreasing order of spiffiness */

    pbTypes = NSArray.arrayWithObjects (W3URIPboardType,
        NSStringPboardType, nil);

    pb.declareTypes (pbTypes, nil);
    pb.setString
        (dstUrl.compositeString(), W3URIPboardType);
    pb.setString
        (dstUrl.compositeString(), NSStringPboardType);

    /* O, Mighty Service of the Appkit, I Invoke Thee! */
    success = NSPerformService(serviceName, pb);
}

```

```
if (!success)
    pipeline.processor (
        self ,
        NSString stringWithFormat(@"Error invoking service\"
            %@\".", serviceName);
    )
}
```

Using Services

Whether you use a standard service (either implementing or invoking it) or a custom service such as is described here, the process is the same: data is placed on a pasteboard and sent to the service. In the case of standard services, it is usually returned as the same or as a different type; here the data (the URL) is not returned.

This example also shows how you can use bundles and property lists to control applications and choose helper applications at run time.

Omni Development

The OmniWeb browser is available directly to users; developers can license parts of it for use in their own applications. The object-oriented architecture lends itself particularly easily to extensions.

You can contact Omni Development at

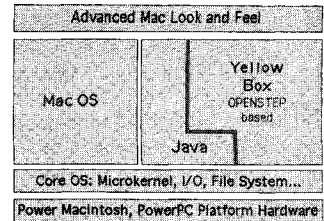
Omni Development
2707 NE Blakeley Street
Seattle, WA 98105-3118

(800) 315-OMNI
(206) 523-4152
(206) 523-5896 (fax)

<http://www.omnigroup.com>

Chapter 20

Notifications with OpenBase



In the last chapter, a service was used under the control of the application (rather than being chosen by the user from the Services menu) to handle some of the application's processing. Although the service is another application and can run on its own, services as shown so far are basically synchronous operations: at a certain point control is transferred to the service, which does its work and then transfers control back to the main application.

This is not necessarily always the case. Services—and other cooperating applications—are often able to go off and start processing on their own in their own address spaces and on their own host computers. Remember that Rhapsody's core OS was designed from the beginning for this type of processing—a type of processing that can

take advantage of the computing environment that is coming into existence even on the most humble desktop. This computing environment is composed of multiple fast processors (often within a single computer box) and of a network that allows fast, broadband data transfers.

Making it possible is Rhapsody's notification mechanism which allows processes to communicate easily with one another. The notification mechanism is demonstrated in this chapter using the OpenBase SQL database engine from OpenBase International.

Using Databases

Modern computing environments permit (and really require) the use of sophisticated data managers to store, retrieve, and maintain data. The volumes of data used in a sophisticated environment can be huge, and when numerous applications run together their needs to share data quickly become evident. Applications can share data among themselves through the pasteboard, distant and proxy objects, and through messaging; but the constant transfer of large amounts of data back and forth quickly eats up system resources unnecessarily. Further, requiring each application to store, retrieve, and maintain its own data is a very expensive design choice (even with the use of frameworks that can generalize much of the work).

The world into which Rhapsody is being launched requires the use of databases for any kind of serious data storage that does anything other than use the disk as a temporary cache for transient information from one execution of the program to the next. There are two complementary strategies with regard to databases in the first Rhapsody releases:

1. The Enterprise Objects Framework (which is part of Rhapsody) allows you to form a simple bridge to existing databases (Oracle, Sybase, Informix, for exam-

ple). It enables you to develop Rhapsody applications around an organization's existing standards and existing data management systems.

2. OpenBase, a third-party database engine, ships with the developer version of Rhapsody and allows developers who do not have an existing data management infrastructure to use a database in support of their Rhapsody applications.

Note that these strategies are not mutually exclusive. You can use Enterprise Objects Framework with the OpenBase database, thus writing everything using EOF. If you're developing a new application and are not working in an environment where there is a database standard, OpenBase is a very good choice for your data manager.

If you're writing a single-user application or even a multiuser application with minimal data storage requirements, you may assume that you don't need any data manager. The moment that your application touches disk storage for any reason, you really should provide the full range of support that users expect. Using a data manager is the simplest way to do this—not to mention the fact that it allows other applications (and the user!) access to the data. In most cases, the data belongs to the user, and locking it up inside your proprietary file formats isn't very nice. And one day, users are going to notice this and rebel.

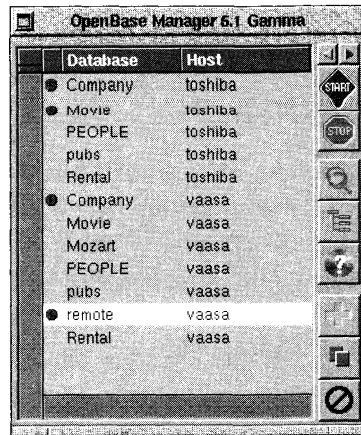
Starting a Database with the Database Manager

A database is the combination of data storage and data retrieval that lets applications and users use data. Databases are usually visible on a system as one or more files; at run time, a database manager accesses the data and serves it up to one or more clients. (In the case of single-user systems, sometimes the database manager is a library that is compiled into individual applications.)

The OpenBase database manager is like most other managers: it lets you identify databases by name and by host (the machine on which the database is stored) and lets you perform

routine maintenance on the databases. The OpenBase Database Manager main window is shown in Figure 20-1.

FIGURE 20-1. OpenBase Database Manager Main Window

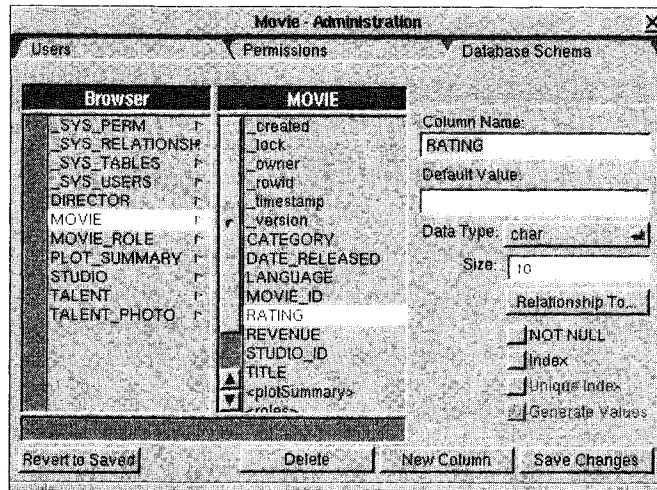


By selecting a database and clicking the Start button, the database is started and is made available to all authorized users.

The Database Manager itself lets you identify users and set their levels of authorization; you can also use it to create new databases and to reorganize them. Figure 20-2 shows the administration window of the OpenBase Database Manager. You first select a database in the OpenBase Database Manager window (Figure 20-1) and click the Admin and Schema Design button (fourth from the top of the palette). Then, for a given table (in this case Movie), you can edit each column (in

this case the column Rating is selected) as shown in Figure 20-2.

FIGURE 20-2. OpenBase Database Manager Admin Window



Connecting to a Database from a Client

Once the database has been started, any authorized users may connect to it. In OpenBase, this is done by linking in the OpenBase libraries (provided with developer Rhapsody) and creating an OpenBase object at run time. The OpenBase object in the client can then execute OpenBase calls, the first of which is almost always a `connectToDatabase` call, which forges the link between the client and the database.

Here is the code to connect to the database remote on host `vaasa` that was selected in Figure 20-1. The database object that will be created is an instance variable called `database`. You can make it an instance of your application object (the most common choice), but it could be located within another object. A single database object represents a connection to an individual database: you can then access all of its tables (subject to authorization permissions). To access another database, you create another database object. Thus, there is one

database object for each database, and many database objects may communicate with a single database manager.

Classic Syntax

```
database = [[OpenBase alloc] init];
if (![database connectToDatabase: "remote"
    onHost: "vaasa"
    login:"root"
    password:""
    return &returnCode)
{
    //handle error
}
```

Modern Syntax

```
database = (OpenBase.alloc()).init();
if (!(database.connectToDatabase ("remote", "vaasa", "root",
    "", &returnCode)
{
    //handle error
}
```

Note that the code to connect to the database uses a login and a password. Depending on the application, you may hard-code an application-specific login (as shown here) or you may query the user for a login and password to be passed to the database manager. In the second case, your error-handling code would have to handle the case of invalid login and/or password.

With these two simple steps—starting the database with the Database Manager and connecting to it with a database object in your client—you have all of the data management tools that you normally need. You send an SQL message to the database object and receive data back from it. Far from being more complicated than using flat files, it is much, much easier.

And, OpenBase provides you with a notification feature that is built on Rhapsody's notification architecture.

Notifications of Database Changes

You don't need to use OpenBase's notification architecture, but it is very simple to use and can provide significant advantages to your application.

The heart of the notification structure is that whenever a change is made to the database, OpenBase tells you: it calls a method of your application and reports what has happened. When dealing with data, this is a frequent need and many applications are written around the absence of notifications.

Although the actual code that implements notifications is very complex (since it optimizes possibly hundreds of clients on a network), at its core, the architecture is very simple. Relying on distant objects, the OpenBase database manager creates a proxy object for your object and then calls a method of your object that it defines in its API. To implement notifications, you therefore need to do two things:

1. Tell OpenBase which of your objects cares about notifications; it will then create a proxy (in the database manager application) for your object.
2. Implement the notification method in your object.

The first step is one line of code:

```
[database startNotificationFor (yourObject)]; //classic  
database.startNotificationFor (yourObject); //modern
```

(database is assumed to be the OpenBase database object that you created at initialization.)

In your object, you implement the `notifyChange` method. Here is a sample of that method:

Classic Syntax

```
- (void)notifyChange:(const char *)action
```

```
database:(const char *)database
table:(const char *)table
rowid:(const char *)rowid
{
    // check to make sure the notification pertains to
    // the right table
    if (strcmp(table, [tableName cString]) != 0) {
        return;
    }

    // if another user unlocks a record
    if (rowidValue && !lockedRowid // is there a read-only
                                     // record displayed
        && (strcmp(action, "(unlock)") == 0) // was it unlocked?
        && [rowidValue isEqualToString:
            [NSString stringWithCString:rowid]])
    {

        // try to lock record and enable user editing
        [self lockRecord];
        return;
    }

    // if another user updates a record
    if (rowidValue // is a record selected?
        && [rowidValue isEqualToString:
            [NSString stringWithCString:rowid]]
        // does the rowid match?
        && (!lockedRowid || !isLockingEnabled) // is the record
        // displayed read-only?
        && (strcmp(action, "(update)") == 0))
    { // was it updated?

        // refresh the display with the changes
        [self setStringValue:rowidValue];
    }

    return;
}
```

Modern Syntax

```
(void)notifyChange:(const char *)action
(const char *)database
(const char *)table
(const char *)rowid)
{
    // check to make sure the notification pertains to
    // the right table
    if (strcmp(table, tableName.cString()) != 0) {
```

```

        return;
    }

    // if another user unlocks a record
    if (rowidValue && !lockedRowid // is there a read-only
        // record displayed
        && (strcmp(action, "(unlock)") == 0) // was it unlocked?
        && rowidValue.isEqualToString (
            NSString stringWithCString(rowid)))
    {
        // try to lock record and enable user editing
        self.lockRecord ();
        return;
    }

    // if another user updates a record
    if (rowidValue // is a record selected?
        && rowidValue.isEqualToString (
            NSString stringWithCString (rowid))
        // does the rowid match?
        && (!lockedRowid || !isLockingEnabled) // is the record
        // displayed read-only?
        && (strcmp(action, "(update)") == 0))
    { // was it updated?

        // refresh the display with the changes
        self.setStringValue (rowidValue);
    }

    return;
}

```

Two local variables are used to process notifications:

- `rowIDValue` is the value of the currently selected row in the user's display.
- `lockedRowID` is a `BOOL` indicating if that row has been locked by another user.

When the `notifyChange` method is called, it first checks to see if the notification pertains to the table that is being displayed. Then, if a row is selected and if it is locked by another user (the italicized line of code), if the notification indicates that the action was an unlock, this object attempts to lock it (`lockRecord`

is the application's method that marks database records for updating).

Similarly, in the boldface lines of code, you can see that if a row has been selected and if the action was update the row will be redisplayed.

Notice how simply this application can respond to events that are taking place in other applications using these Rhapsody features:

- Distant and proxy objects allow the database manager to call a method of your application.
- Dynamic binding lets the database manager call the `notifyChange` method of your object without imposing any other restrictions on you. It doesn't have to be an object of any particular type (it can be your application object, a view, etc.). All it must do is implement the `notifyChange` method.

Whether or not you use databases in your applications (and you should), and whether or not you use OpenBase, the Enterprise Objects Framework, or both, the simplicity with which you can let two applications communicate should impress even the most jaded developer. And for those developers who think that they don't need this sort of frill, careers in telephone sales may provide more job security.

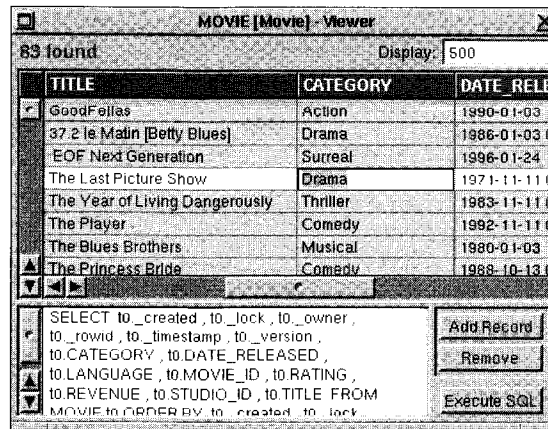
OpenBase

OpenBase is a product of OpenBase International. It has been bundled with Rhapsody to allow developers to use it directly and to have a database product to use with Enterprise Objects.

In addition to the features described here, it provides a data browser for users, as shown in Figure 20-3. Users can enter

SQL directly or can construct it graphically using the database manager.

FIGURE 20-3. Data Viewer



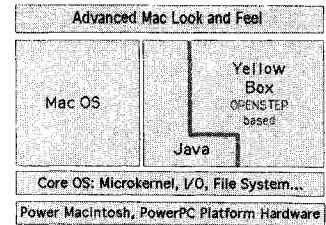
You can reach OpenBase at

OpenBase International, Ltd.
58 Greenfield Road
Francestown, NH 03043

(603) 547-8404
(603) 547-2423 (fax)

<http://www.openbase.com>
scott@openbase.com

Index



A

accessors **66**
 in semantics tables 122
actions **128**, 132
Algol 3
Apple Guide **268**
archiveRootObject 325
archiving 325–328
archiving and distribution **119**
 archiving 325–328

 coding 319–324
 distribution 328–329
 NSArchiver 325
 NSCoder 322
 NSCoding [P] 319
 NSPortCoder 328
 NSUnarchiver 327
arrays and enumerators 172
assertions **170**
autorelease pools 141

Principal entries and definitions are shown in bold.

B

bison 426
Blue Box 31
box 133
browsers 133
buffalo, American 426
bundle 116
bundles 382–385

C

C with Classes 98
C++ 75
 delegation 104
 fragile base class problem 79, 102
 multiple inheritance 103, 104
 static typing 99
 vtables 102
Caffeine Software 432
cells 132
CHaRTSMITH 441–455
class methods 123
classNameed 383
coach mark 268
coalescing (of notifications) 335
Cobol 3
coding 319–324
color 119
 NSColor 209
 NSColorList 213
commands 119
 drag-and-drop 296–301
 NSDraggingDestination [P] 299
 NSDraggingInfo [P] 300
 NSDraggingSource [P] 298
 menus 284–296
 NSMenu 288
 NSMenuItem [P] 293
 NSMenuValidation [P] 292
 panels 301
connections 126, 369–372

connections (between outlets and interface objects) 131
content view 239
controls 119
 notification 330–335
 NSNotification 332
 NSNotificationCenter 333
 NSBox 215
 NSButton 308–311
 NSButtonCell 311
 NSCell 216
 NSControl 303
 NSOpenPanel 381
 NSPrintPanel 410
 NSSavePanel 377
 NSSelection 336
coordinate system 228
Create 433–440
cursor handling 251

D

data hiding. See encapsulation
data links 338–348
data types 119
 data links 338–348
 dates and times 160
 NSDate 160
 NSTimeZone 165
 NSTimeZoneDetail 167
 NSData 151
 NSDataLink 339
 NSDataLinkManager 339
 NSDataLinkPanel 347
 NSDecimalNumber 158
 NSMutableData 154
 NSNumber 157
 NSValue 155
databases 472
delegate 130
 for NSControl 304
 for NSPanel 376

- for NSSavePanel 380
 - for NSWindow 280
- delegation
 - in C++ 104
 - in Rhapsody 106
- development tools 114
- dictionaries 177
- display **119**, 245
 - NSPasteboard 349–354
- Display PostScript 117, 249, 395
- distant objects 480
- distributed objects 449–454
 - allowing a connection (from the server) 453
 - making a connection from the client 451
- distribution 328–329
- doesNotRecognizeSelector 144
- drag-and-drop 296
- dragImage 297
- Dylan 74, 75
- dynamic binding 126, 480
 - TIFFany example 425
- dynamic linking 7
 - in Objective-C 80
- dynamic loading 437–440
 - Create example 437
- dynamic typing 126
 - in Objective-C 76
- dynamism 70, 126
 - in Objective-C 76

E

- Encapsulated PostScript 412
- encapsulation 65
- encodeObject 322
- encodeRootObject 325
- encodeWithCoder 320, 321, 328
- Enterprise Objects Framework 472
- error handling **119**, 167
 - NSAssertionHandler 171
 - NSException 169
- events **127**

- exception handling 46, **51**
- exceptions 168

F

- fabrication 121
- failure handling **51**
- fault handling. *See* failure handling
- fax 413
- field editors 406
- files
 - opening and saving 376–382
- Fortran 3
- forwardInvocation 83, 146
- fragile base class problem (in C++) 79, 102

H

- high-level language 3

I

- images **120**
 - NSImage 229
 - NSImageRep 235
- inheritance 64
- initWithCoder 320, 321, 328
- initWithCondition 363
- Interface Builder 114, 115, 116
- isValidFileName 376, 377

J

- Java
 - Object class 90

packages 94

K

key window 129, 130

L

Latitude 109, 110

Lisp 74, 75

loadTheAnimatorBundle 438

localization 117

locking 361–366

lproj 117

M

MacApp 128

Mach 53–55, 359

- memory management 54

- messaging 54

- tasks and threads 53

main window 129, 130

mainBundle 437

makeFirstResponder 399

memory management 46, 50, 54

messaging 46, 50, 54

methodSignatureForSelector 146

Metrowerks 109

microkernel architecture 52

modern OS

- exception and fault handling 46

- memory management 46

- messaging 46

- microkernel architecture 52

- process management 45

- security 45

modern syntax (Objective-C) 76

multiple inheritance 103, 104

N

nibs 115

notification 245, 330–335

NS_DURING 168

NS_ENDHANDLER 168

NS_HANDLER 168

NSApp 117

- setServiceProvider 447

NSApplication 117, 192–203

- Events and Actions 197

- Management 194

- Managing Menus 204

- Modality 201

- Panels 202

- Window Management 199

NSArchiver

- Methods 325

NSArray 172–173

- Fabrication Methods 172

- Utility Methods 173

NSAssert 170

NSAssertionHandler 171

- Methods 171

NSBackingStoreBuffered 260

NSBackingStoreNonretained 260

NSBackingStoreRetained 260

NSBackingStoreType 260

NSBorderType 215

NSBox 215

- Methods 215

NSBundle

- mainBundle 437

- Methods 383

- pathForResource 431, 437

- principalClass 438

NSBundleAdditions

- Methods 205

NSButton 303, 308

- Methods 309

-
- Methods Described Elsewhere **312**
 - NSButtonCell 311–314
 - Methods **312**
 - Methods Described Elsewhere **314**
 - NSCalendarDate **164**
 - Methods **164**
 - NSCAssert 170
 - NSCell 133, 216–228
 - Accessors for Text **224**
 - Action Accessors **228**
 - Action and Responder Methods **218**
 - coordinates 221
 - Fabrication Methods **218**
 - General Accessors **221**
 - Image Accessors **227**
 - represented object 217
 - NSCharacterSet **181**
 - Methods **182**
 - NSClipView 254
 - NSClosableWindowMask 260
 - NSCoder 143
 - Coding Methods **322**
 - Object Coding Methods **324**
 - Utility Methods **324**
 - NSCoding [P]
 - Methods **320**
 - NSColor 209–213
 - Methods **210**
 - NSColorList 213
 - Methods **213**
 - NSComparisonResult **134**
 - NSConditionLock
 - Methods **364**
 - NSConnection
 - example in CHARTSMITH 454
 - Methods **370**
 - NSControl 133, 303–308
 - delegate 304
 - Methods **304**
 - Methods That Are Described in NSCell **307**
 - NSCopying [P]
 - Methods **357**
 - NSData 151–153
 - Accessor Methods **152**
 - Fabrication Methods **151**
 - Utility Methods **153**
 - NSDataLink
 - Methods **340**
 - NSDataLinkManager
 - Delegate Methods **345**
 - Methods **343**
 - NSDataLinkPanel
 - Methods **348**
 - NSDate **161**
 - Fabrication Methods **161**
 - Utility Methods **163**
 - NSDecimalNumber **158**
 - Methods **159**
 - NSDecimalNumberBehavior [P] 158
 - NSDecimalNumberHandler 158
 - NSDeviceBitsPerSample **271**
 - NSDeviceColorSpaceName **271**
 - NSDevicesScreen **271**
 - NSDeviceResolution **271**
 - NSDeviceSize **271**
 - NSDictionary 178–180, **271**
 - Fabrication Methods **178**
 - Utility Methods **179**
 - NSDPSContext 395
 - NSDraggingDestination [P] **299**
 - Methods **299**
 - NSDraggingInfo [P] **300**
 - Methods **301**
 - NSDraggingSource [P] **298**
 - Methods **298**
 - NSDragOperationAll 297
 - NSDragOperationCopy 297
 - NSDragOperationGeneric 297
 - NSDragOperationLink 297
 - NSDragOperationNone 297
 - NSDragOperationPrivate 297
 - NSEnumerator **177**
 - Method **177**
 - NSEvent 392–398
 - Accessors **397**
 - Fabrication Methods **395**
 - NSEventTypes and Flags Constants **394**
 - NSException **169**
 - Methods **170**
 - NSFileWrapper
 - Fabrication Methods **386**

- Methods for Directories **388**
- Utility Methods **387**
- NSFloatingWindowLevel 265
- NSImage 229–234
 - delegate 229
 - Methods **229**
- NSImageRep **235**
 - Methods **235**
 - registry of image types 235
- NSInternalInconsistencyException 170, 171
- NSInvocation 138, **145**
 - Methods **145**
- NSLock
 - Method **363**
- NSLocking [P]
 - Methods **366**
- NSMenu 288–292
 - Methods **289**
- NSMenuItem [P] **293**
 - Methods **294**
- NSMenuValidation [P] **292**
 - Method **293**
 - validateMenuItem 293
- NSMiniturizableWindowMask 260
- NSMutableArray 175
 - Methods **176**
- NSMutableCharacterSet **181**
 - Methods **183**
- NSMutableCopying [P]
 - Methods **357**
- NSMutableData **154**
 - Fabrication Methods **154**
 - Utility Methods **155**
- NSMutableDictionary **180**
 - Methods **180**
- NSMutableString **188**
 - Methods **189**
- NSNibAwaking
 - Method **206**
- NSNormalWindowLevel 265
- NSNotification
 - Methods **333**
- NSNotificationCenter
 - Methods **334**
- NSNotificationQueue 335
- NSNumber **157**
 - Methods **157**
- NSObject 135–145
 - archiving 143
 - Archiving Methods **143**
 - doesNotRecognizeSelector 144
 - dynamism 144
 - Dynamism Methods **145**
 - error handling 144
 - Error Handling Method **144**
 - fabrication 137
 - Fabrication Methods **138**
 - forwardInvocation 83
 - functionalities 136
 - garbage collection and memory management 141
 - Garbage Collection and Memory Management Methods **142**
 - identification 138
 - Identification Methods **140**
 - introspection 138
 - Introspection Methods **139**
 - message handling 138
 - Messaging Methods **140**
 - respondsToSelector 83, 129, 138
- NSOpenPanel
 - Methods **382**
- NSOrderedAscending 134
- NSOrderedDescending 134
- NSOrderedSame 134
- NSPanel **301**
 - Methods **302**
- NSPasteboard 349–354
 - Methods **350**
- NSPopUpButton 427
- NSPopupButton 308
- NSPopupButtonCell 308
- NSPortCoder
 - Methods **329**
- NSPrinter
 - Methods **420**
- NSPrintInfo
 - Methods **414**
- NSPrintingPageOrder **419**
- NSPrintOperation
 - Methods **416**

- NSPrintPanel
 - Methods 411
- NSProxy 135, 146
 - example in CHARTSMITH 454
 - forwardInvocation 146
 - Methods 146
 - methodSignatureForSelector 146
- NSRange 134
- NSRecursiveLock
 - Method 365
- NSResizableWindowMask 260
- NSResponder 128, 398–407
 - First and Next Responder Methods 399
 - Keyboard Event Methods 403
 - Message and Command Management Methods 401
 - Mouse Event Methods 402
 - Other Event Methods 406
- NSRulerMarker 429
- NSSavePanel
 - Delegate Methods 381
 - Fabrication and Accessor Methods 377
 - Utility Methods 379
- NSScanner
 - Utility Methods 190
- NSScreen 259
 - Methods 259
- NSScrollView 254
- NSSelection
 - Methods 337
- NSSelectionDirection 274
- NSServices
 - NSExecutable 446
 - NSHost 446
 - NSKeyEquivalent 446
 - NSMenuItem 445
 - NSMessage 445
 - NSPortName 445
 - NSReturnTypes 445
 - NSSendTypes 445
 - NSTimeOut 446
 - NSUserData 446
- NSServices property 444
- NSServicesRequest
 - Methods 408
- NSSlider 303
- NSString 183–188
 - Encoding Methods 185
 - File Path Name Methods 189
 - Methods 183
 - String Manipulation Methods 185
- NSTask
 - Methods 368
- NSText 304
- NSTextView 402
- NSThread
 - Methods 366
- NSTimeZone 165
 - Methods 166
- NSTimeZoneDetail 167
 - Methods 167
- NSTitledWindowMask 260
- NSTitlePosition 216
- NSUnarchiver
 - Methods 327
- NSValue 156
 - Methods 156
- NSView 237–258
 - cursor handling 251
 - display 245
 - dragImage 297
 - Drawing Methods 250
 - Event and Mouse Handling Methods 252
 - Fabrication Method 239
 - Geometry Methods 242
 - Hierarchy Methods 240
 - Menus and Scrolling Methods 254
 - notification 245
 - Notification Methods 245
 - Paper Handling Methods 256
 - setNeedsDisplay 245
 - Size, Focus, and Display Methods 247
- NSViewMaxXMargin 246
- NSViewMinXMargin 246
- NSViewWidthSizable 246
- NSWindow 259–280
 - Delegate Methods 281
 - delegate methods 280
 - Drawing Methods 269
 - Editing Methods 275
 - Event and Key View Methods 273
 - Fabrication Methods 261

Final Methods 279
Geometry Methods 261
makeFirstResponder 399
Menus, Cursor, and Dragging Methods 278
More Window Management Methods 276
Window Management Methods 266
Window Server and Screen Methods 271
NSWindowAbove 266
NSWindowBelow 266
NSWindowOrderingMode 266
NSWindowOut 266

O

Object class (Java) 90
Objective-C
 class and instance objects 80
 classic syntax 84
 dynamic linking 80
 dynamic typing 76
 dynamism 76
 modern syntax 76, 84
 protocols 82
object-oriented programming 58–64
 accessors 66
 benefits 60
 classes 63
 encapsulation 65
 inheritance 64
 instances 63
 learning curve 61
 memory management 71
 messages vs. methods 75
 performance 70
 polymorphism 64
 terminology 62
objects 127
Omni Development 470
OmniWeb 457
opaque background 238
OpenBase 471–481

Data Viewer 481
Database Manager 474
optional arguments 123
outlets 126, 131

P

packages (in Java) 94
panels 134
Pasteboard Owner's Methods 354
pathForResource 431, 437
periodic events 127, 227, 393
polymorphism 64
ports 370
PostScript 256
PostScript Printer Description (PPD) files 420
PowerPlant 128
principalClass 438
process management 45, 47
programming tools 120
 NSArray 172
 NSBundle 382
 NSCopying [P] 356
 NSDictionary 178
 NSEnumerator 177
 NSEvent 392–398
 NSFileWrapper 385
 NSMutableArray 175
 NSMutableCopying [P] 356
 NSMutableDictionary 180
 NSPrinter 420
 NSPrintInfo 413
 NSPrintOperation 416
 NSResponder 398, 398–407
 NSScreen 259
 NSServicesRequest [P] 408
 NSView 237
 NSWindow 259
Project Builder 114
protocols 82, 94, 104, 452
 conformsToProtocol 139
proxy objects 454, 480

Q

QuickDraw 249

R

reference date 160, 161
 release 141, 142
 represented object 217
 responder chain **129**
 vs. target chain 130
 responders **128**, 398–407
 respondsToSelector 83, 129, 138
 retain 141, 142
 Rhapsody
 Blue Box 31
 diagram 32
 evolution of 20–36
 Yellow Box 31
 Rhapsody on Intel 30
 root object 454
 ruler views
 markers in 428–432

S

scripts 405
 security 45, **46**
 semantics tables
 accessors in 122
 class methods in 123
 described 120
 meaning of † 123
 optional arguments 123
 services 126, 280
 Services menu 443
 setting up 444
 setNeedsDisplay 245
 setServiceProvider 447

Smalltalk 74, 75, 171
 SOM 80
 static typing 99
 Stone Design **440**
 strftime 163
 Stroustrup, Bjarne 103
 styleMask **260**
 synchronism **120**, 357–372
 connections 369–372
 kinds of 358
 locking 361, 361–366
 NSConditionLock 363
 NSLock 362
 NSLocking [P] 365
 NSRecursiveLock 364
 NSConnection 369
 NSTask 367
 NSThread 366
 opportunities for 360
 tasks 360, 367–369
 threading 360, 366–367

T

target chain
 vs. responder chain 130
 tasks 360, 361, 367–369
 tasks and threads **53**
 text **120**
 NSCharacterSet 181
 NSMutableCharacterSet 181
 NSMutableString 188
 NSScanner 190
 NSString 183
 NSStringEncoding 184
 threading 366–367
 threads 360
 TIFFany 426–432

U

Unicode 88
UNIX 43
unlockWithCondition 363

V

validateMenuItem 293
vtables 102

W

Window Server 117, 237, 259
windowNum 395

Y

Year 2000 Problem 160
Yellow Box 31
 on Mac OS 35
 on Windows 33

Related Titles from AP Professional

- ALGER, *Secrets of the C++ Masters*
- ANDREWS, *Migrating to Windows 95*
- CRANE, *Mutual Fund Investing on the Internet*
- DE LEEUW, *Digital Cinematography*
- EGE, *Object-Oriented Programming With C++, Second Edition*
- FOLEY, *Windows NT Server Training Guide*
- GRAHAM, *TCP/IP Addressing*
- HELLER, *C++ Training Guide*
- HELLER, *Efficient C/C++ Programming, Second Edition*
- HELLER, *Who's Afraid of C++?*
- KEOGH, *Solving the Year 2000 Problem*
- KEOGH, *Webmaster's Guide to VB Script*
- LEVINE, *Live Java*
- LEVITUS/EVANS, *WebMaster Macintosh (Second Edition)*
- LEVITUS/EVANS, *WebMaster Windows (Second Edition)*
- LOSHIN, *TCP/IP Clearly Explained*
- McKEEHAN, *Safe Surfing: A Family Tour of the Net*
- MURRAY/PAPPAS, *Java with Borland C++*
- MURRAY/PAPPAS, *Visual Basic 5 Training Guide*
- MURRAY/PAPPAS, *Visual J++ Handbook*
- MURRAY/PAPPAS, *Windows 95 and NT Programming with the Microsoft Foundation Class Library*
- OZER, *Publishing Digital Video*
- PFAFFENBERGER, *The Elements of Hypertext Style*
- PFAFFENBERGER, *Netscape Navigator 3.0 (Macintosh Version)*
- PFAFFENBERGER, *Netscape Navigator 3.0 (Windows Version)*
- PFAFFENBERGER, *Netscape Navigator Gold*
- PFAFFENBERGER, *Publish It on the Web! (Macintosh Version)*
- PFAFFENBERGER, *Publish It on the Web! (Windows Version)*
- REISS, *The Eudora User's Guide*
- RIBAR, *The Internet with Windows 95*
- SCHETTINO/O'HARA, *Applescript Applications*
- STEVENS, *C++ Graphics Programming Handbook*
- STEVENS, *Object-Oriented Graphics Programming with C++*
- TITTEL, *ISDN Clearly Explained*
- TITTEL, *PC Networking Handbook*
- TRAISTER, *Going from C to C++*
- TURLEY, *Windows 95 Training Guide*
- TURLEY, *Windows NT Training Guide*
- VACCA, *JavaScript*
- VAUGHAN-NICHOLS, *Intranets*
- WAYNER, *Agents at Large*
- WAYNER, *Digital Cash, Second Edition*
- WAYNER, *Digital Copyright Protection*

WAYNER, *Disappearing Cryptography*

WEISKAMP, *Complete C++ Primer, Second Edition*

WAYNER, *Java and JavaScript Programming*

YOUNG, *Introduction to Graphics Programming for
Windows 95*

Ordering Information



AP Professional

An imprint of ACADEMIC PRESS

A division of HARCOURT BRACE & COMPANY

ORDERS (USA and Canada): 1-800-3131-APP or APP@acad.com

AP Professional Orders: 6277 Sea Harbor Dr., Orlando, FL 32821-9816

Europe/Middle East/Africa: 0-11-44 (0) 181-300-3322

Orders: AP Professional 24-28 Oval Rd., London NW1 7DX

Japan/Korea: 03-3234-3911-5

Orders: Harcourt Brace Japan, Inc., Ichibancho Central Building 22-1, Ichibancho Chiyoda-Ku,
Tokyo 102

Australia: 02-517-8999

Orders: Harcourt Brace & Co., Australia, Locked Bag 16, Marrickville, NSW 2204 Australia

Other International: (407) 345-3800

AP Professional Orders: 6277 Sea Harbor Dr., Orlando, FL 32821-9816

Editorial: 1300 Boylston St., Chestnut Hill, MA 02167 (617) 232-0500

Web: <http://www.apnet.com/approfessional>