DISCOVERING OPENSTEP: A Developer Tutorial

Rhapsody Developer Release

Apple Computer, Inc.

User Interface Tips copyright © 1997 Apple Computer, Inc.
All rights reserved.
[6467.00]

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except to make a backup copy of any documentation provided on CD-ROM. Printed in the United States of America.

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

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

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

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

Apple, and the Apple logo are trademarks of Apple Computer, Inc., registered in the United States and other countries. NeXT, the NeXT logo, NEXTSTEP, the NEXTSTEP logo, Digital Librarian, NeXTmail, and Workspace Manager are trademarks of NeXT Computer, Inc. All other trademarks mentioned belong to their respective owners.

Written by: Terry Donoghue
Tutorial applications by: Terry Donoghue
Art and Production: Terri FitzMaurice
Publications management: Ron Hayden
With help from: Trey Matteson, Ron Hayden, Jean Ostrem, Lynn Cox, Derek Clegg, Mike Ferris, and Frederic Bonnard

Table of Contents

Welcome to Rhapsody 7

A Simple Application 19

Creating the Currency Converter Project 22
Creating the Currency Converter Interface 24
- Customizing the Application's Window 24
- Fields and Buttons 28
- Some Finishing Touches 32

Defining the Classes of Currency Converter 38
- Connecting ConverterController to the Interface 44

Building the Currency Converter Project 54
Run Currency Converter 57

A Forms-Based Application 61

Creating the Travel Advisor Interface 64
- Getting Started 64
- Getting the Table View to Work 97

A Multi-Document Application 115

Setting Up the To Do Project 118
Creating the Model Class (ToDoItem) 119
- Setting Up the Programmatic Interface 119
- Breaktime! 125

Extending an Application Kit Class: An Example 127
The Basics of a Multi-Document Application 138
Managing Documents Through Delegation 149
Managing ToDo's Data and Coordinating its Display 158

Extending the To Do Application 167

Overriding Behavior of an Application Kit Class: An Example 170
Creating and Managing an Inspector (ToDoInspector) 172
Overriding and Adding Behavior to a Class: An Example 190
Setting Up Timers 196
Build, Run, and Extend the Application 199
- Optional Exercises 199

Appendix A - Object-Oriented Programming 203

- Encapsulation 205
- Messages 206
- An Object-Oriented Program 207
- Polymorphism and Dynamic Binding 208

Classes 210
- Object Creation 210
- Inheritance 211
- Defining a Class 213
- The Yellow Box Frameworks 215

Appendix B - Programming Tools and Resources 219

- Other Development Applications 220
- Other Installed Frameworks 220
- Useful Command-Line Tools 221
- Other Programming Resources 222

Introduction Chapter 1

1

What You'll Learn

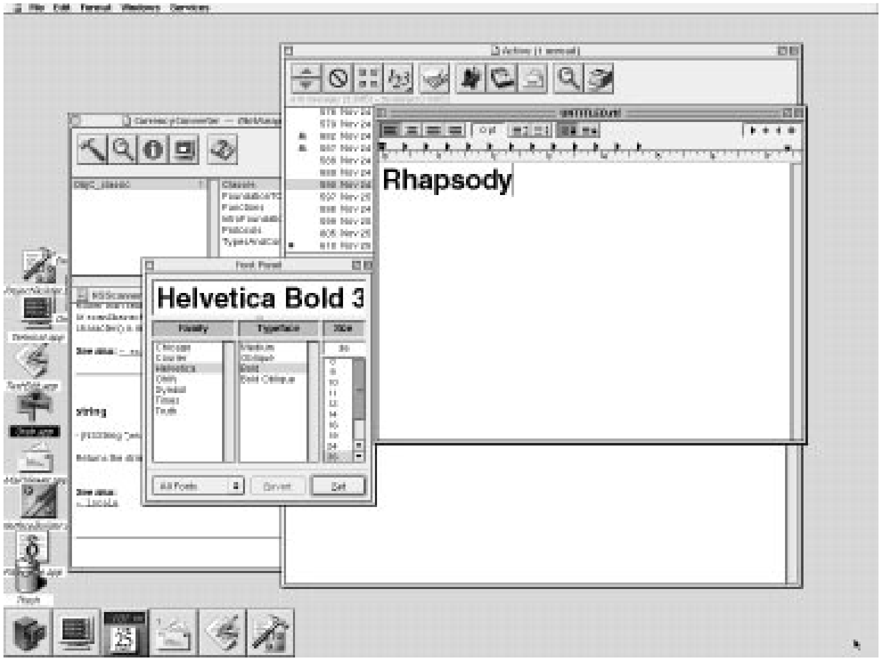
What OpenStep is

How OpenStep fits in with Rhapsody

Rhapsody technologies

Programming in Apple's development environment

Programming with objects



Chapter 1
Welcome to Rhapsody

Welcome to OpenStep.

OpenStep is a new way to make programs that run on Power PC Macintoshes. But OpenStep itself is not new. It is proven technology, and although it poses a learning curve for newcomers, once you learn it, application development will suddenly seem easier and quickened with potential.

This book eases your way into the experience of OpenStep programming. It guides you through the creation of several applications. It encourages you to explore, to "kick the tires." Along the way, it explains important concepts and paradigms, and it uncovers rich lodes in the tools and APIs.

When you've worked through this book, Apple's development environment will not only be less mysterious, but will be an environment that you'll want to program in.

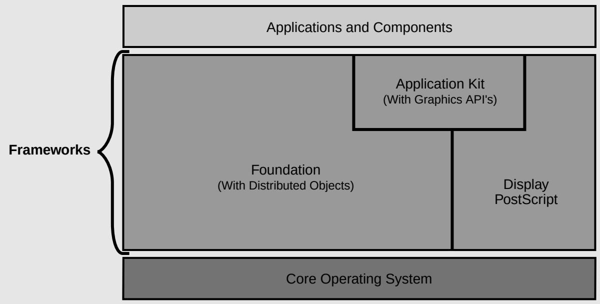
This chapter presents a brief overview of OpenStep—the user experience and the developer experience—and places it in the larger context, which is the next-generation Macintosh operating system.

Imagine a Macintosh…

Imagine a Macintosh:

- That doesn't crash when an application crashes.
- That can draw a graph, send a fax, and play a movie, all at once.
- That seems much faster than any Windows system you've seen.
- That looks a lot like the Mac OS, only better.
- That features some of the most advanced software technology around.
- And that still runs all the Macintosh programs you know and love.

That Macintosh will soon be here. And you're invited to be a part of its genesis.



What is OpenStep?

The new Macintosh operating system, code-named "Rhapsody," has actually been around a long time—almost as long as the Mac itself. That's because it is based on OpenStep, which, as NEXTSTEP, was introduced in 1987. Since then, OpenStep has evolved through many releases, has been adopted by many companies as their development and deployment environment of choice, and has received glowing reviews in the press. It is solid technology, based on a design that was years ahead of anything else and perfected year after year.

So OpenStep is well-regarded and battle-tested. But what is it? OpenStep is an integrated set of shared object libraries, or *frameworks*, plus a run time and a development environment that do three principal things:

- They insulate programs from the internal workings of the core operating system, mediating access to system resources and preventing programs from trashing one another's address space.
- They provide all (or almost all) the functionality that programs typically need.
- They bring the benefits of object-orientation to program development (see page 14).

OpenStep has three core object frameworks: Foundation, Application Kit, and Display PostScript.

Foundation Defines basic object characteristics, and implements mechanisms for object allocation, deallocation, introspection, and distribution. Foundation includes classes for common data types, such as strings, numbers, and collections. And it provides APIs for platform-independent system services, such as dates and times, multithreading support, task and process management, timers, file management, notification, and internationalization (based on the Unicode standard),

Application Kit Consists of classes that generate user-interface objects, that manage and process events, and that offer or assist in application services such as color and font management, printing, text manipulation, and cut-and-paste.

Display PostScript Provides APIs for direct PostScript drawing and image composition, as well as for low-level window, cursor, and event management. The Application Kit uses these APIs to draw the objects of the user interface. (New Graphics APIs will offer a higher level of abstraction for drawing operations.)

Both the Foundation and Display PostScript frameworks interact directly with the core operating system (Display PostScript with the windowing and imaging subsystem); applications can access operating-system services through the APIs of these frameworks.

The part of Rhapsody called the Yellow Box augments and enhances the core OpenStep frameworks with many other frameworks. See page 10 for details.

OpenStep and Mach

You can think of OpenStep as a layer of objects that acts as mediator and facilitator between programs and the core operating system. The stability, performance, and reliability of these programs therefore depend on whether the underlying core operating system has these characteristics.

The core operating system for Rhapsody is an enhanced version of Mach and BSD. Mach—the original foundation of NEXTSTEP—is a mature, robust kernel that provides low-level services such as memory management, tasking, synchronization, timing, and event messaging. These services form the basis of advanced operating-system capabilities: preemptive multitasking, memory protection, full symmetric multiprocessing, and high-performance I/O.

Rhapsody's core operating system will eventually support a variety of file and volume formats, including HFS, UFS, DOS FAT, ISO9660, AFP, and HFS+ (an enhanced version of HFS).

Rhapsody: Where OpenStep Fits In

Rhapsody is a ensemble of technologies in which OpenStep plays a central role. The diagram below depicts Rhapsody on PowerPC-based systems; it shows an enhanced and expanded version of OpenStep as the *Yellow Box*. The Yellow Box has interfaces to the core operating system, to the advanced Macintosh user experience, to a Java virtual machine, and to a Mac OS–compatibility subsystem known as the *Blue Box*.

Blue Box. This is a native Mac OS environment that runs on the Rhapsody kernel as another application (on PowerPC–based systems only). It enables the execution of programs written for current and prior versions of the Mac OS. Because a Blue Box environment is "just" another Rhapsody application, it shares in the benefits of the kernel. For example, one or more Blue Boxes can be running at a time on one machine; if one crashes, it will not affect other Blue Boxes or any other Rhapsody application. A Blue Box can take over the entire screen or it can occupy only a portion of it. In full-screen mode you can use a hot key to switch between the Blue and Yellow Boxes.

Mac OS software running in the Blue Box cannot directly access services provided by the core operating system or the Yellow Box; by the same token, programs written for the Yellow Box cannot directly access services in the Blue Box. Rhapsody supports limited sharing of data between Blue and Yellow Boxes, including copy-and-paste, and will eventually permit communication between them through Apple events.

Some Yellow Box frameworks and development tools are being ported to the Blue Box, allowing Yellow Box development in that environment.

Many current Mac OS technologies, such as QuickDraw 3D, QuickDraw GX, OpenDoc, and QuickTime, are being carried forward into the Blue Box.

Advanced look and feel. The new Rhapsody operating system sports a new exterior that perfects the Mac OS, itself legendary for ease of use. The Rhapsody user experience combines the best visual elements and usage models of the Mac OS and of OPENSTEP for Mach, and it incorporates new paradigms in human-interface design. Yet it is still recognizably a Macintosh operating system.

Programs written for the Yellow Box and Java programs written to the Yellow Box APIs will present the same advanced Macintosh look and feel to users.

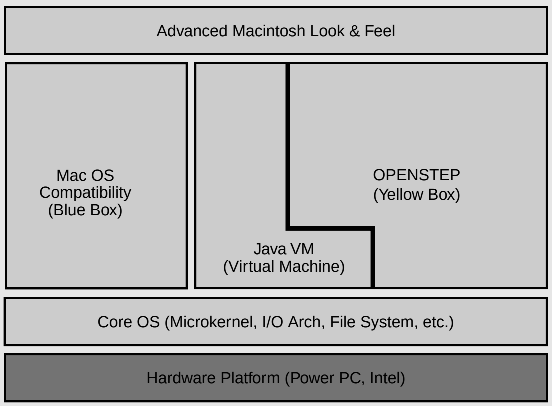
Java virtual machine. Recognizing the increasing importance of Java in software technology, Apple is including a Java virtual machine (VM) as part of the Rhapsody picture. The Yellow Box will offer APIs in Java for accessing the functionality of the core frameworks, and it will include ported versions of the AWT, IFC, AFC, and other Java packages. With these possibilities, you can create Java programs that will run on any platform with a Yellow Box or "100% pure" Java applications that will run on any platform that has a Java virtual machine.

Rhapsodic Variations

With little more than a recompile, you can deploy applications written to the Yellow Box APIs (aka OpenStep) in four different user environments:

- **Rhapsody** (for PowerPC). See diagram below.
- **Rhapsody for Intel Processors**. Same as above, minus the Blue Box.
- **Yellow Box for Windows**. For Windows NT and Windows 95.
- **Yellow Box for Mac OS**. (Forthcoming.)

Applications will exhibit the "look and feel" appropriate to the underlying operating system.



The Yellow Box: A Blend of Technologies

What makes Rhapsody a uniquely rich environment for both users and developers is the assortment of technologies clustered around the core OpenStep frameworks; taken together, they are known as the Yellow Box. Some of these extended frameworks are NeXT products, carried forward; some frameworks are being developed for Rhapsody; and others are being ported from the current Mac OS—QuickTime, QuickDraw 3D, QuickDraw GX, and ColorSync, to name a few.

The sections that follow describe the range of technologies that these extended frameworks will make available. They also summarize some the fundamental technologies incorporated by Rhapsody.

Imaging and Printing Model

The imaging *and* printing model for Rhapsody is Display PostScript. Unlike in the Mac OS, the same mechanism is used to view and print what appears on the screen. You no longer have to duplicate code to send output both to the screen and to PostScript-based devices. The best of Apple's graphic technologies, including ColorSync and QuickDraw GX typography, is being migrated to the Display PostScript model.

Display PostScript has several other advantages over existing Mac OS imaging models. It improves the performance and responsiveness of the graphical user interface. It is a widely accepted standard in the industry. It is also easy to write code for; you use the Graphics APIs or the APIs of the Display PostScript framework. However, applications that must severely limit the overhead for imaging can access the frame buffer directly by using APIs for this purpose, called Interceptor.

Multimedia

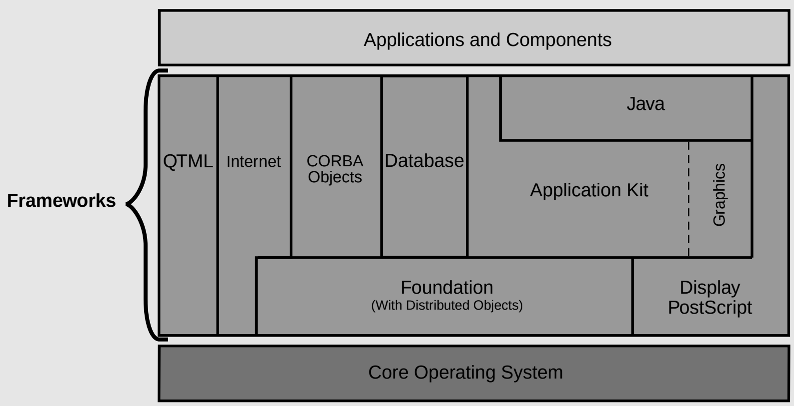
The QuickTime Media Layer (QTML) is a collection of objects and APIs that make it easy for you to give applications rich multimedia content. QTML consists of three separate products:

- QuickTime is the industry-standard multimedia architecture that software vendors and content creators use to store, edit, and play synchronized graphics, sound, video, text, and music.
- QuickTime VR delivers virtual reality in both panoramas and objects.
- QuickDraw 3D enables developers to render real-time three-dimensional graphics.

Distributed Computing

The Yellow Box offers APIs for creating distributed applications, thereby eliminating the need to write code for many low-level network operations. The Foundation framework includes Distributed Objects, a technology that permits objects in different task or threads—on a single host or across a network—to communicate with each other.

The Yellow Box will also support CORBA/IIOP (including an object request broker), making it easy to create industrial-strength applications that work across different types of networks.



Internet

Apple intends Rhapsody to be a major technological force in the world of the Internet. The development platform features APIs for Internet-based mail, messaging, directories, and security services. Moreover, the Yellow Box includes the WebObjects framework which, along with WebObjects Builder (an interactive application for the creation of dynamic Web pages), enables the speedy prototyping and development of dynamic Web-server applications. These applications can access data in standard relational databases and can communicate with applet components on the client browser. The Yellow Box also offers built-in HTML rendering capabilities.

Database Integration

The Enterprise Objects Framework provides applications with access to data on local relational database systems. Adaptors for Oracle, Sybase, and Informix, and ODBC-compliant databases are available separately. Enterprise Objects achieves persistent storage of data through a process of mapping objects to tables.

The Yellow Box will also make a local database engine available, allowing you to build and test database applications locally. When these applications are deployed in a client/server environment, the customer then needs to acquire a Web-application server (WebObjects) or to get an adaptor for the supported databases.

Localization and Internationalization

You can easily localize Yellow Box applications largely because of a well-designed localization architecture and Unicode support, both built into the Application Kit. In this architecture, user-interface elements—as archived objects and as resources—are kept separate from the executable. It's therefore possible to have a single code base that is qualified for various locales. You can even have multiple localizations bundled with one application, greatly reducing the overall footprint of an application in its various localizations. Since localization bundles can be easily added to or removed from an existing application, new localizations can be distributed through updaters.

Because the Yellow Box uses Unicode 2.0 as its native character set, applications can easily handle all of the world's languages. The prevalence of Unicode also eliminates many character-encoding hassles. Still there are Yellow Box APIs to help translate between Unicode and other major character sets in use today.

Apple's development environment supports localization in several important ways. It gives you an easy way to identify which files are to be localized (and for which language). Andit enables you to create a series of archivable user interfaces, each designed for a particular locale.

Text and Fonts

The Yellow Box offers a powerful set of text services that can be readily adapted by text-intensive applications requiring high performance. These services, which can support text buffers larger than 32K, include kerning, ligatures, tab formatting, and rulers.

By the Unified Release, the Yellow Box will support a variety of font formats, including Type 1 PostScript, Type 3 PostScript, Type 42 PostScript, and TrueType (including the typographic capabilities of TrueType GX). The goal for Rhapsody is for an open font architecture that makes it easy for users to work with any font format they want.

Microsoft Windows

You can develop a Yellow Box application on one platform, say Rhapsody for Power PC, and deploy it on another supported platform, including Windows, with little more than a recompilation. Yellow Box applications that run on Microsoft Windows have a range of capabilities at their disposal. With OLE/COM, Yellow Box applications can transparently communicate with OLE-enabled applications such as Microsoft Word. They can also use ActiveX controls within the Windows environment. OLE/COM and ActiveX support—as well as the ability to make Win32 calls from your code—permits the development of industrial-strength Windows applications that are seamlessly integrated with other Windows applications.

Component Technologies

One of the key advantages of the Yellow Box as a development environment is the capability for developing programs quickly by assembling reusable components. With the proper programming tools and a little work, you can build Yellow Box components that can be packaged and distributed for others to use. Applications are an obvious example of this component technology, but there are others. With the Yellow Box and Apple's development tools, you can create:

- Frameworks, which other developers can use to create programs by writing code based on the framework APIs
- Bundles containing executable code and associated resources, which programs can dynamically load
- Palettes containing custom user-interface objects that other developers can drag and drop into their own user interfaces using the Rhapsody development tools

With the Yellow Box's component architecture, you can easily create and distribute extensions and plug-ins for applications. You can also develop components based on JavaBeans technology in both the Yellow Box and the Blue Box. JavaBeans components will be integrated into the Rhapsody development tools. On Windows you can use ActiveX components.

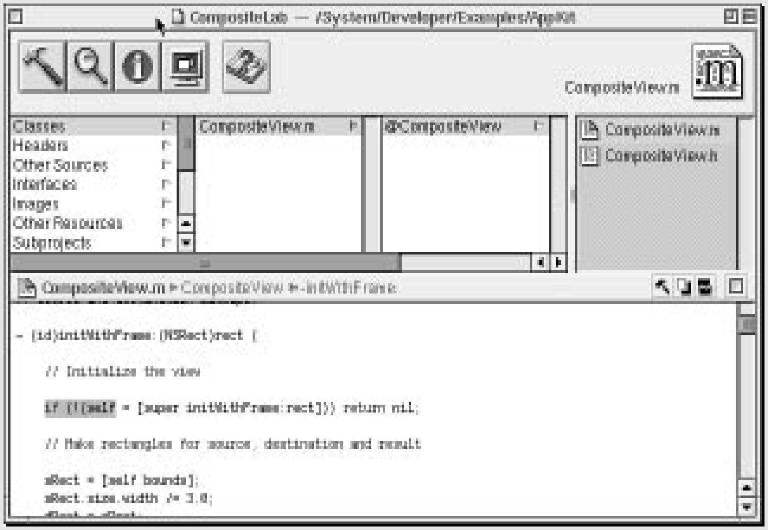
Programming in the Apple Development Environment

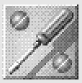
Apple has a powerful, integrated, cross-platform development environment for the Yellow Box on Rhapsody. With it you can easily build applications, and easily deploy them on multiple platforms: Windows NT, Windows 95, and (of course) Rhapsody on both PowerPC–based and Pentium-based machines. You develop the application for one platform and, with little more than a simple recompile, the same application is ready to run on another platform. You can also develop Rhapsody applications written in Java which will run on any platform supporting Apple's Java virtual machine (see "Java Looming" on the following page).

The Apple development environment consists of a suite of applications and tools that deliver maximum productivity from the frameworks, subsystems, libraries, components, and other resources of Rhapsody. The principal applications are Interface Builder and Project Builder.

Project Builder is an application that manages software-development projects and that orchestrates and streamlines the development process. It integrates a project browser, a full-featured code editor, language-savvy symbol recognition, a class browser, sophisticated project search capabilities, header file and documentation access, build customization, a graphical debugger, Java support, and a host of other features.

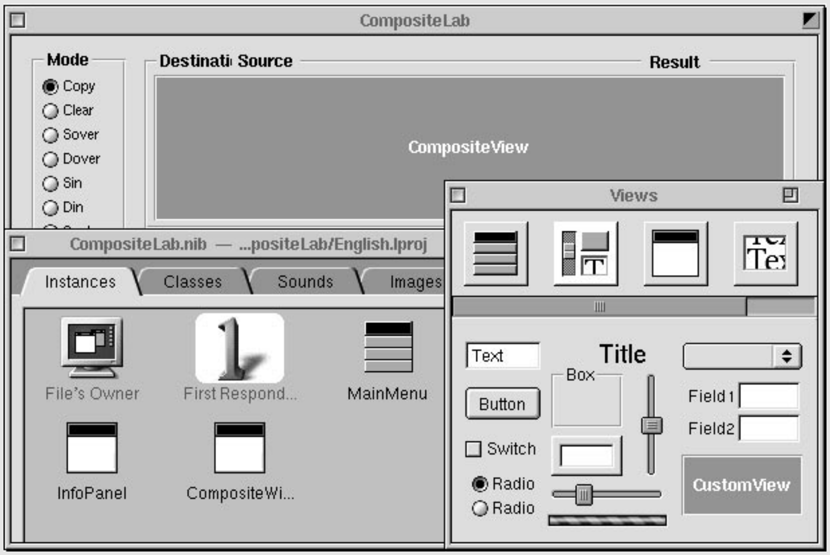
Project Builder allows you to plug in compilers from other tool vendors (such as Metrowerks). It facilitates the creation of different types of projects (such as palettes, frameworks, and bundles, in addition to applications). It also provides programmatic hooks for integrating source-control management systems.





Interface Builder makes it easy to create application interfaces. You just drag an object from a palette and drop it on the graphical user interface you're creating. You can then set attributes of these objects through an inspector panel and you can connect them to other objects in your application so they can send messages to each other. Interface Builder also assists in the definition of custom classes and allows you to test an interface without having to compile a line of code.

Standard palettes hold an assortment of Application Kit objects. Other palettes can include Yellow Box objects from other frameworks, third-party objects, and custom compiled objects. You can also store non-compiled configurations of objects on *dynamic palettes*. Interface Builder archives and restores elements of a user interface (including connections) as objects—it doesn't "hardwire" them into the interface. Interface Builder also enables youto connect your application to JavaBeans and ActiveX components.



Other Development Tools

Apple's development environment for the Yellow Box has much more than Project Builder and Interface Builder to offer. There are other applications: FileMerge, which allows you to compare and selectively merge files and directories; Yap, which allows you to preview and test PostScript code; and MallocDebug and other applications that analyze and optimize code that you've written.

In addition, the development environment includes a shell application (**Terminal.app**) with which you can run many development utilities. However, use of command-line utilities is optional; they are not required for Yellow Box development. (Eventually, many of these utilities will be incorporated into new development applications.)

Pick Your Language

In developing applications for Rhapsody's Yellow Box, you have a choice of programming language. You can write programs, in whole or in part, in C, C++, Java, and Objective-C. Soon scripting languages will also be supported.

Some developers have the notion that Objective-C is difficult. They are mistaken. You shouldn't dread the thought of learning Objective-C. It is a simple and elegant language. A typical developer, especially one experienced in C++, should need no more than a day or two to learn Objective-C.

Note: Yellow Box programs cannot be completely written in C++ because you cannot subclass Yellow Box classes in C++. For your program to take advantage of the Yellow Box frameworks, C++ objects must be integrated with Objective-C or Java objects.

Java Looming

Apple is aware of the growing importance of Java and expects that Java will become the language of choice for many developers. Since Java is central to Apple's system and development strategies, both the Blue and Yellow Boxes will feature high-performance Java virtual machines and will include the latest versions of the Java Development Kit (JDK).

Besides hosting native Java packages—including AWT, JFC, AFC, and IFC—the Yellow Box will provide access to its own APIs in Java. You will be able to subclass Yellow Box classes in Java and mix "pure" Java and Yellow Box objects in your code. You will be able to write "100% pure" Java applications that can run on any platform that has the appropriate virtual machine. Or you can write Java applications that use Yellow Box APIs; these applications can run—without recompiling—anywhere the Yellow Box is available. Rhapsody will also integrate JavaBeans into the Yellow Box run time and into Interface Builder palettes.

Project Builder will include several features that specifically support the development of Java applications.

How Apple's Development Environment Compares

The Yellow Box development environment is, in several ways, like traditional Macintosh development environments such as MPW and CodeWarrior. For example, like these products, it permits a great deal of customization, and it possesses sophisticated searching capabilities. But the Yellow Box development environment is different in many ways, both large and small. Some of the differences take some getting used to, such as the bindings of shortcut keys and the way you browse the project.

Apple is listening to its developers and is rapidly evolving its development tools to meet the needs and expectations of this community. In addition, traditional vendors of Macintosh development tools, such as MetroWerks, are busy porting existing tools to Rhapsody or are creating new tools. For example, MetroWerks is developing a tool (Latitude) that will assist Macintosh developers in porting their PowerPlant and other projects to the Yellow Box. Eventually, Rhapsody developers will be able to "mix and match" the exact tool set that suits them best.

Programming With Objects

For some Mac OS developers, the most striking disparity they'll experience when they start developing Yellow Box programs is not the tool set. It is the shift in mind set that is required for object-oriented programming (OOP).

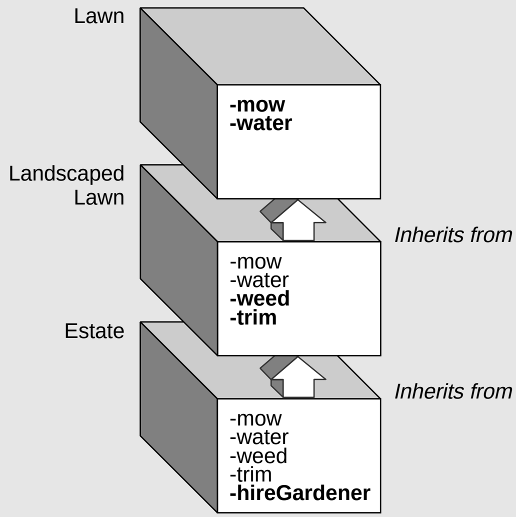
Instead of thinking in terms of procedures and data, you have to think in terms of objects—discrete programmatic units containing their own data as well as procedures that act on that data. An object-oriented program is composed of objects of different types, each type fulfilling a specialized role within the program. In such a program, objects are constantly sending messages to each other—that is, they are requesting other objects to execute a procedure. The object receiving the message performs what is requested and, in some cases, returns an object or another value. (For more on how object-oriented programs works, see the appendix "Object-Oriented Programming.")

Learning how to program with objects takes some initial effort, but with some familiarity, object-oriented programming begins to seem natural, elegant, and powerful. And, with the rich functionality of the Yellow Box frameworks to tap, application development becomes easier—you get many application features "for free." Programming with objects, especially Yellow Box objects, increases your productivity by freeing you from many repetitive coding tasks. You have more time to accomplish what is truly creative.

To mesh your custom code with framework objects, you must create a subclass of at least one of the framework classes. The subclass implements behavior or logic specific to your application and obtains the services it needs from framework objects. Moreover, a custom subclass inherits attributes and behavior from its superclass, again without you having to write a single line of code (see illustration). Often one or two subclasses is all that is required to achieve quite substantial results.

Of course, to program effectively with the Yellow Box you must learn what services you can obtain from framework classes and what attributes and behavior you inherit from them. Even for developers experienced in object-oriented programming, the Yellow Box frameworks pose their own learning hurdle. You need to become familiar with the class hierarchy, to discover what classes can do, and learn how they interact with one another.

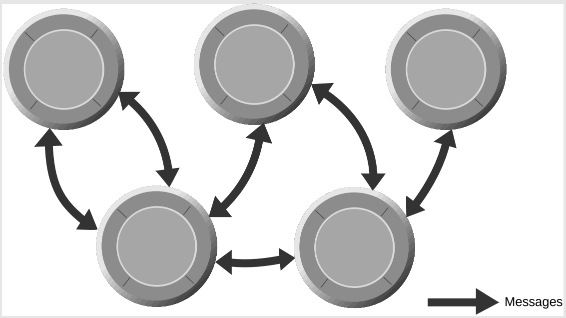
This learning requirement is unavoidable, regardless of the development environment. But Apple tries to ease the effort required with tool features such as a class browser and documentation such as the framework references and this book, which introduces some of the more fundamental classes.



The Advantage of Objects

Object-orientation is the software equivalent of the Industrial Revolution. In the same way that modern factories assemble products out of prefabricated components rather than manufacture every product from scratch, object-orientation allows programmers to build complex software by reusing software components called objects. Specifically, objects lead to several measurable advantages:

Greater reliability. By breaking complex software projects into small, self-contained, and modular objects, object-orientation ensures that changes to one part of a software project will not adversely affect other portions of the software. Being small, each of these objects is a well-tested module of code, and so the overall reliability of the software increases.

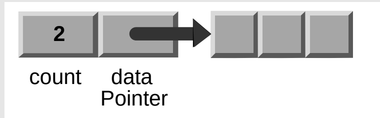


Easier maintainability. Since objects are modular and usually small (in terms of the overall code size of a project), bugs in code are easier to locate. Developers can also change the implementation of an object without causing havoc in other parts of an application.

Greater productivity through reuse. One of the principal benefits of object-orientation is reuse. One object can be integrated into many applications. And through subclassing, you can create specialized objects merely by adding the code unique to the new object. Objects of the new subclass inherit functionality from the superclass, reducing coding and promoting greater reliability.

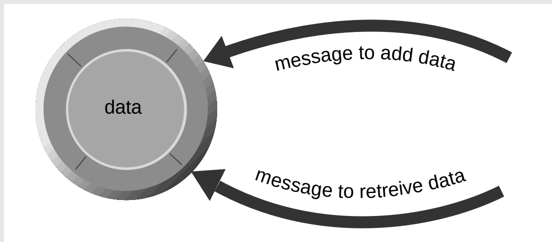
An Example

Object-oriented programming delivers its greatest benefits to large and complex programs. But its advantages can also be demonstrated with a simple data structure such as might be used in any application.



With procedural programming techniques, the application is directly responsible for data manipulation. One problem with this is illustrated in the picture above. It shows a data structure consisting of a **count** variable and a data pointer. Since the application directly manipulates the data, it has the opportunity to introduce inconsistencies. Here, it has added an item to the data, but has forgotten to increment the count; the **coun**t variable says there are still only two data elements when in fact there are three. The structure has become inconsistent and unreliable.

Another problem is that all parts of the application must have intimate knowledge about the structure of the data. If the allocation of data elements is changed from a statically allocated array to a dynamically allocated linked list, it would affect every part of the application that accesses, adds, or deletes elements from the list.



With an object-oriented programming paradigm, the application as a whole doesn't directly manipulate the data structure; rather, that task is entrusted to a particular object. Since the application doesn't directly access the data, it can't introduce inconsistencies. Note also that it's possible to change the implementation of the object without breaking other parts of the application. For example, the data storage method could be changed to optimize performance. So long as the object responds to the same messages, other parts of the application are unaffected by internal implementation details.

Currency Converter Tutorial Chapter 2

2

What You'll Learn

Creating a simple graphical user interface

Creating a custom subclass

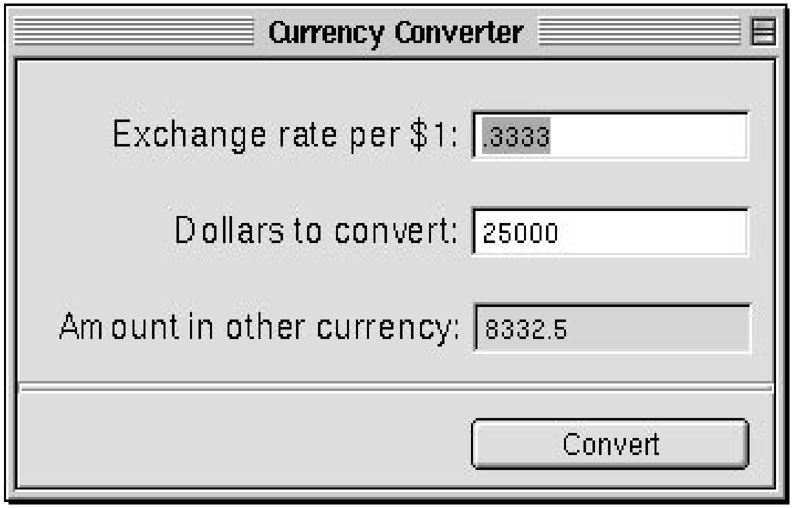
Connecting objects in the application

Sending a message to an object

Responding to a message

Building a project

Getting help



You can find the Currency Converter project in the **AppKit** subdirectory of **/System/Developer/Examples**.

Chapter 2
A Simple Application

The application that you are going to create in this tutorial is called Currency Converter. It is a simple application, yet it exemplifies much of what software development with OpenStep is about. As you'll discover, Currency Converter is amazingly easy to create, but it's equally amazing how many features you get "for free"— as with all OpenStep applications.

Currency Converter converts a dollar amount to an amount in another currency, given the rate of that currency relative to the dollar. You type a rate and an amount into text fields and then click a button to see the result. Instead of clicking the button, you can also press the Return key. You can double-click the converted amount, copy it (with the Edit menu's Copy command) and paste it in another application that takes text. You can tab between the first two fields. You can do many other things common to OpenStep applications.

By following the steps of this chapter, you will become familiar with the two most important OpenStep applications for program development: Interface Builder and Project Builder.

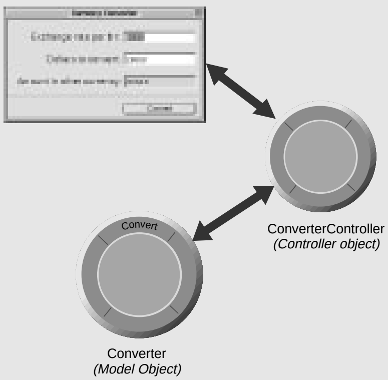
Currency Converter's Design, the Development Process, and a Design Paradigm

An object-oriented application should be based on a design that identifies the objects of the application and clearly defines their roles and responsibilities. You normally work on a design before you write a line of code. You don't need any fancy tools for designing many applications; a pencil and a pad of paper will do.

Currency Converter is an extremely simple application, but there's still a design behind it. This design is based upon the Model-View-Controller paradigm, a model behind many designs for object-oriented programs (see next page). This design paradigm aids in the development of maintainable, extensible, and understandable systems. But first, you might want to read "Why an Object Looks Like a Jelly Donut" on page 29 to understand the symbol used in the design diagram.

This design for Currency Converter is intended to illustrate a few points, and so may be overly designed for something so simple. It is quite possible to have the application's controller class, ConverterController, do the computation and do without the Converter class.

You can divide responsibility within Currency Converter among two custom objects and the user interface, taken as a collection of ready-made Application Kit objects. The Converter object is responsible for computing a currency amount and returning that value. Between the user interface and the Converter object is a *controller object*, ConverterController. ConverterController coordinates the activity between the Converter object and the UI objects.



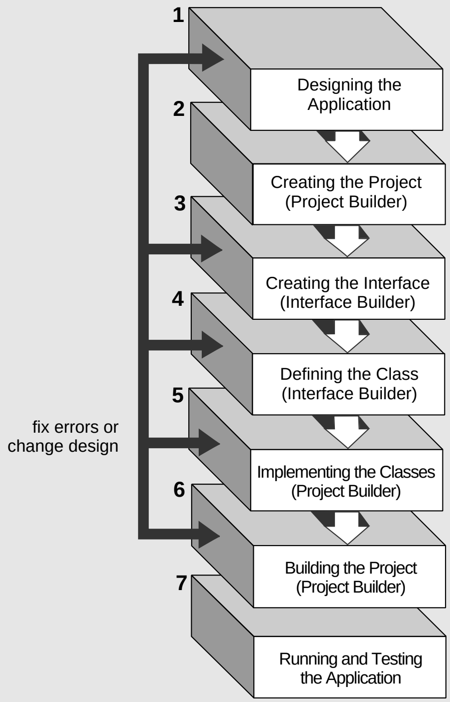
The ConverterController class assumes a central role. Like all controller objects, it communicates with the interface and with model objects, and it handles tasks specific to the application. ConverterController gets the values that users enter into fields, passes these values to the Converter object, gets the result back from Converter, and puts this result in a field in the interface.

The Converter class merely computes a value from two arguments passed into it and returns the result. As with any model object, it could also hold data as well as provide computational services. Thus, objects that represent customer records (for example) are akin to Converter. By insulating the Converter class from application-specific details, the design for Currency Converter makes it more reusable, as you'll see in the Travel Advisor tutorial.

Typical Development Workflow

This chapter introduces the typical flow of work involved in developing an OpenStep application

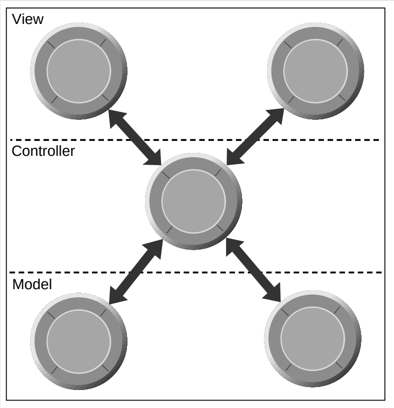
.



Note: Although this diagram shows the design phase at the beginning of the workflow process, application design can take place any time in the early stages of the project. It is often recommended as the first stage, however, and it is a good idea to review the design occasionally and modify it if necessary.

The Model-View-Controller Paradigm

A common and useful paradigm for object-oriented applications, particularly business applications, is Model-View-Controller (MVC). Derived from Smalltalk-80, MVC proposes three types of objects in an application, separated by abstract boundaries and communicating with each other across those boundaries.



Model Objects

This type of object represents special knowledge and expertise. Model objects hold a company's data and define the logic that manipulates that data. For example, a Customer object, common in business applications, is a Model object. It holds data describing the salient facts of a customer and has access to algorithms that access and calculate new data from those facts. A more specialized Model class might be one in a meteorological system called Front; objects of this class would contain the data and intelligence to represent weather fronts. Model objects are not directly displayed. They often are reusable, distributed, persistent, and portable to a variety of platforms.

View Objects

A View object in the paradigm represents something visible on the user interface (a window, for example, or a button). A View object is "ignorant" of the data it displays. The Application Kit usually provides all the View objects you need: windows, text fields, scroll views, buttons, browsers, and so on. But you might want to create your own View objects to show or represent your data in a novel way (for example, a graph view). You can also group View objects within a window in novel ways specific to an application. View objects, especially those in kits, tend to be very reusable and so provide consistency between applications.

Controller Object

Acting as a mediator between Model objects and View objects in an application is a Controller object. There is usually one per application or window. A Controller object communicates data back and forth between the Model objects and the View objects. It also performs all the application-specific chores, such as loading nib files and acting as window and application delegate. Since what a Controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application's code. (This last statement does not mean, however, that Controller objects *cannot* be reused; with a good design, they can.)

Because of the Controller's central, mediating role, Model objects need not know about the state and events of the user interface, and View objects need not know about the programmatic interfaces of the Model objects. You can make your View and Model objects available to others from a palette in Interface Builder.

Hybrid Models

MVC, strictly observed, is not advisable in all circumstances. Sometimes it's best to combine roles. For instance, in a graphics-intensive application, such as an arcade game, you might have several View objects that merge the roles of View and Model. In some applications, especially simple ones, you can combine the roles of Controller and Model; these objects join the special data structures and logic of Model objects with the Controller's hooks to the interface.

A Note on Terminology

The Application Kit and Enterprise Objects Framework reserve special meanings for "view object" and "model." A view object in the Application Kit denotes a user-interface object that inherits from NSView. In the Enterprise Objects Framework, a model establishes and maintains a correspondence between an enterprise object class and data stored in a relational database. This book uses "model object" only within the context of the Model-View-Controller paradigm.

Creating the Currency Converter Project

Every Rhapsody application starts out as a *project*. A project is a repository for all the elements that go into the application, such as source code files, makefiles, frameworks, libraries, the application's user interface, sounds, and images. You use the Project Builder application to create and manage projects.

1 **Launch Project Builder.**

Locate the Project Builder application (icon at right).

Double-click the icon to start the application.

Project Builder is located at **/System/Developer/Apps/ProjectBuilder.app**.

When Project Builder starts up, it displays the New Project panel. The New Project panel lets you specify a new project's name, location, and type.

1 **Make a new project.**

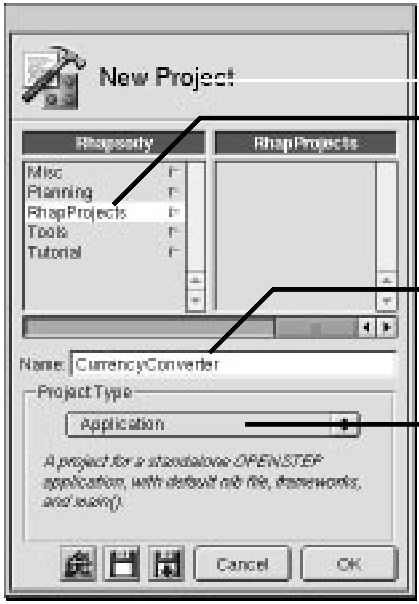
Choose New from the Project menu (Project ►New).

In the New Project panel, choose the Application project type from the pop-up list.

Using the file browser, go to the directory you want the project to be in.

Type "CurrencyConverter" in the Name field.

Click OK to create the project.

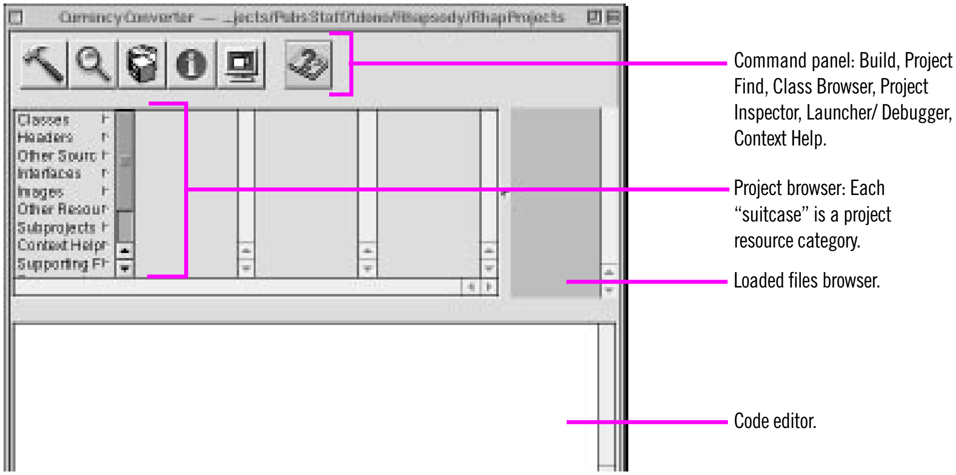


Often projects are kept in a common directory.

The name specified here becomes the name of the project directory and the default name of the application itself.

Make sure Application is the project type.

Project Builder creates a project directory named after the project—in this case CurrencyConverter—and populates this directory with an assortment of ready-made files and directories. It then displays its main window.



Go ahead and click an item in the left column of the project browser (a grouping of project resources sometimes called a "suitcase"); see what some of these suitcases contain already:

- **Other Sources:** This suitcase contains **CurrencyConverter_main.m**, the **main()** routine that loads the initial set of resources and runs the application. (You shouldn't have to modify this file.)

- **Interfaces:** This suitcase contains the nib files (extension".nib") which specify the application's user interface. More on nib files in the next step.

- **Supporting Files:** This suitcase contains the project's default makefiles and template source-code files. You can modify the preamble and postamble makefiles, but you must leave **Makefile** unchanged.

- **Frameworks**: This suitcase contains the frameworks (which are similar to libraries) which the application imports.

Project Indexing

When you create or open a project, after some seconds you may notice triangular "branch" buttons appearing after source code files in the browser. Project Builder has indexed these files.

During indexing Project Builder stores all symbols of the project (classes, methods, globals, etc.) in virtual memory. This allows Project Builder to access project-wide information quickly. Indexing is indispensable to such features as name completion and Project Find. (More on these features later.)

Usually indexing happens automatically when you create or open a project. You can turn off this option if you wish. Choose Preferences from the Tools menu and then choose the Indexing display. Turn off the "Index when project is opened" switch.

You can also index a project at any time by choosing Tools ▶ Indexer ▶ Index Subproject. If you want to do without indexing (maybe you have memory constraints), choose Tools ▶ Indexer ▶ Purge Indices.

Creating the Currency Converter Interface

When you create an application project, Project Builder puts the *main nib file* in the Interfaces suitcase. A nib file is primarily a description of a user interface (or part of a user interface). The main nib file contains the main menu and any windows and panels you want to appear when your application starts up; at start-up time, each application loads the main nib file.

Customizing the Application's Window

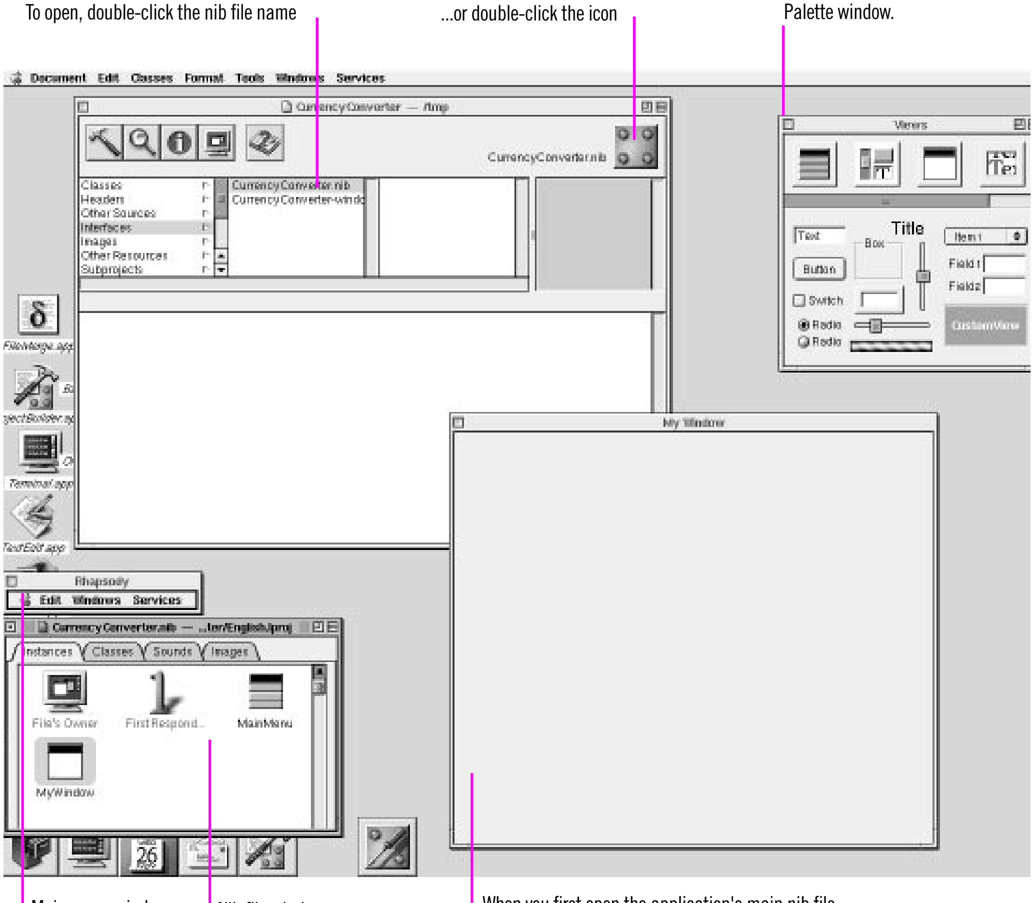
At the beginning of a project, the main nib file is like a blank canvas, ready for you to craft the interface. Look in the Interfaces suitcase for nib files.

1 **Open the main nib file.**

Locate **CurrencyConverter.nib** in the project browser.

Double-click to open.

To open, double-click the nib file name ...or double-click the icon Palette window.



Main menu window. Nib file window When you first open the application's main nib file, Interface Builder displays a blank window.

By default, a blank window entitled "My Window" will appear when the application is launched.

What's in a Nib File

Every application has at least one nib file. The main nib file contains the application menu and often a window and other objects. An application can have other nib files as well. Each nib file contains:

Archived Objects Encoded information on OPENSTEP objects, including their size, location, and position in the object hierarchy (for view objects, determined by superview/subview relationship). At the top of the hierarchy of archived objects is the File's Owner object, a proxy object that points to the actual object that owns the nib file.

Images Image files that you drag and drop over the nib file window or over an object that can accept them (such as a button or image view).

Class References Interface Builder can store the details of OPENSTEP objects and objects that you palettize (static palettes), but it does not know how to archive instances of your custom classes since it doesn't have access to the code. For these classes, Interface Builder stores a proxy object to which it attaches class information.

Connection Information Information about how objects within the object hierarchy are interconnected. Connector objects special to Interface Builder store this information. When you save the document, connector objects are archived in the nib file along with the objects they connect.

When You Load a Nib File

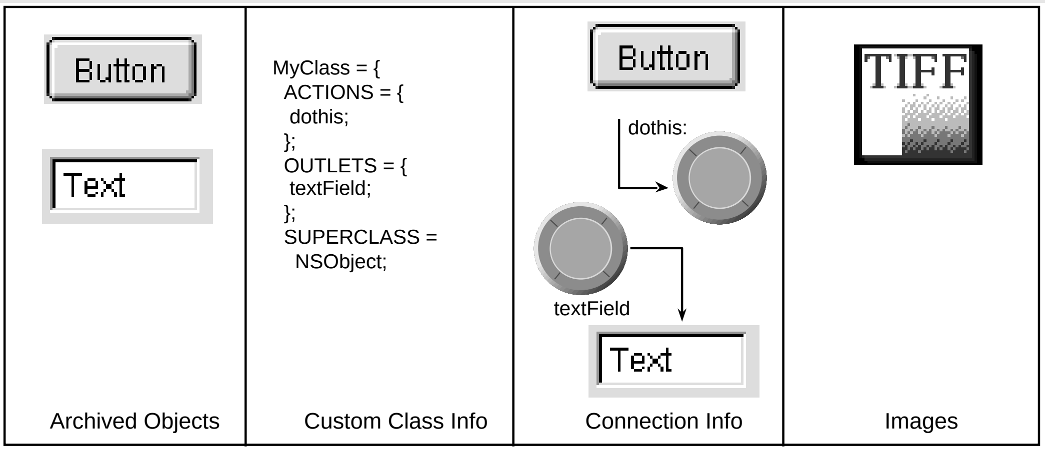
In your code, you can load a nib file by sending the NSBundle class **loadNibNamed:owner:** or **loadNibFile:externalNameTable:withZone:**. messages. When you do this, the run-time system does the following:

- It unarchives the objects from the object hierarchy, sending each object an **initWithCoder:** message after allocating memory for it.
- It unarchives each proxy object and queries it to determine the identity of the class that the proxy represents. Then it creates an instance of this custom class (**alloc** and **init**) and frees the proxy.
- It unarchives the connector objects and allows them to establish connections, including connections to File's Owner.
- It sends **awakeFromNib** to all objects that were derived from information in the nib file, signalling that the loading process is complete.

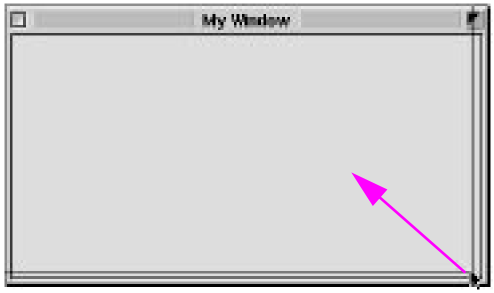
Connections and Accessor Methods

When OpenStep establishes connections during the course of loading a nib file, it sets the values of the source object's outlets. It first tries to set an outlet through the "set" accessor method if the source object implements it. For example, if the source object has an outlet named "contraption," the system first sees if that object responds to "setContraption:" and, if it does, it invokes the accessor method. If the source object doesn't implement the accessor method, the system sets the outlet directly.

Problems naturally ensue if a "set" accessor method does something other than directly set the outlet. One common example is an accessor method that sets the *string value* of an outlet referring to a text field (**setStringValue:**). After loading, the value of the outlet is **nil** because the "set" accessor method did not directly assign the value.



1 **Resize the window.**



Make the window smaller by dragging a corner of the window inward.

Most objects on an interface have attributes that you can set in the Inspector panel's Attributes display.

1 **Set the window's title and attributes.**

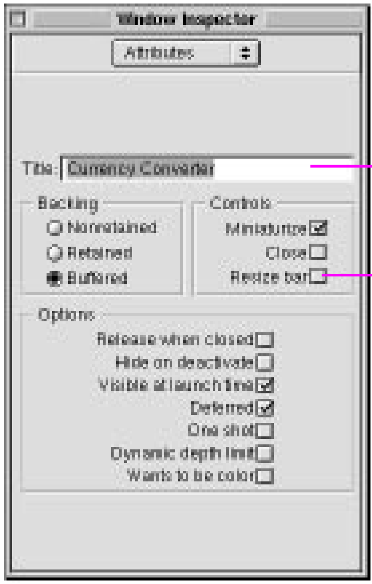
Click the window to select it.

Choose Tools ► Inspector.

Select the Attributes display from the pop-up list.

Enter the window title.

Turn off the resize bar.



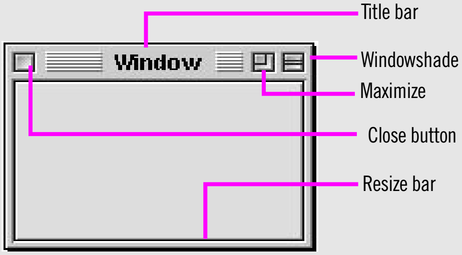
The title of the major window in an application is often the application name.

When this option is turned off, the windows's resize bar disappears.

Note: You can also bring up the Attributes display of the inspector by typing Control-1.

A Window in OpenStep

A window in OpenStep looks very similar to windows in other user environments such as Windows or Mac OS. It is a rectangular area on the screen in which an application displays controls, fields, text, and graphics. Windows can be moved around the screen and stacked on top of each other like pieces of paper. A typical OpenStep window has a title bar, a content area, and several control objects.



Many user-interface objects other than the *standard window* depicted above are windows. Menus, pop-up lists, and pull-down lists are primarily windows, as are all varieties of panels: attention panels, inspectors, and tool palettes, to name a few. In fact, *anything* drawn on the screen must appear in a window.

NSWindow and the Window Server

Two interacting systems create and manage OpenStep windows. On the one hand, a window is created by the Window Server. The Window Server is a process integrating the Window System and Display Postscript. The Window Server draws, resizes, hides, and moves windows using Postscript primitives. The Window Server also detects user events (such as mouse clicks) and forwards them to applications.

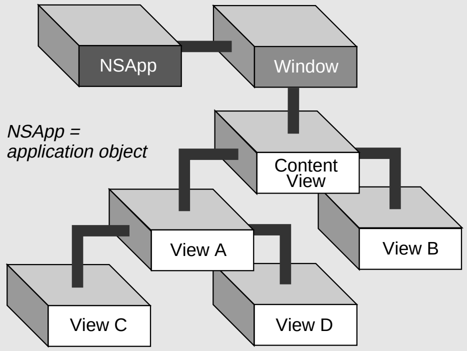
The window that the Window Server creates is paired with an object supplied by the Application Kit: an instance of the NSWindow class. Each physical window in an object-oriented program is managed by an instance of NSWindow (or subclass).

When you create an NSWindow object, the Window Server creates the physical window that the NSWindow object will manage. The Window Server references the window by its window number, the NSWindow by its own identifier.

Application, Window, View

In a running OpenStep application, NSWindow objects occupy a middle position between an instance of NSApplication and the views of the application. (A view is an object that can draw itself and detect user events.) The NSApplication object keeps a list of its windows and tracks the current status of each. Each window, on the other hand, manages a hierarchy of views in addition to its PostScript window.

At the "top" of this hierarchy is the *content view*, which fits just within the window's content rectangle. The content view encloses all other view (its *subviews*), which come below it in the hierarchy. The NSWindow distributes events to views in the hierarchy and regulates coordinate transformations among them.



Another rectangle, the *frame rectangle*, defines the outer boundary of the window and includes the title bar and the window's controls. The lower-left corner of the frame rectangle defines the window's location relative to the screen's coordinate system and establishes the base coordinate system for the views of the window. Views draw themselves in coordinate systems transformed from (and relative to) this base coordinate system.

See page 153 for more on the view hierarchy.

Key and Main Windows

Windows have numerous characteristics. They can be on-screen or off-screen. On-screen windows are "layered" on the screen in tiers managed by the Window Server. On-screen windows also can carry a status: *key* or *main*.

Key windows respond to key presses for an application and are the primary recipient of action messages from menus and panels. Usually a window is made key when the user clicks it. Key windows have black title bars. Each application can have only one key window.

An application has one main window, which can often have key status as well. The main window is the principal focus of user actions for an application. Often user actions in a modal key window (typically a panel such as the Font panel or an inspector) have a direct effect on the main window. In this case, the title bar of the main window (when it is not key) is a dark gray.

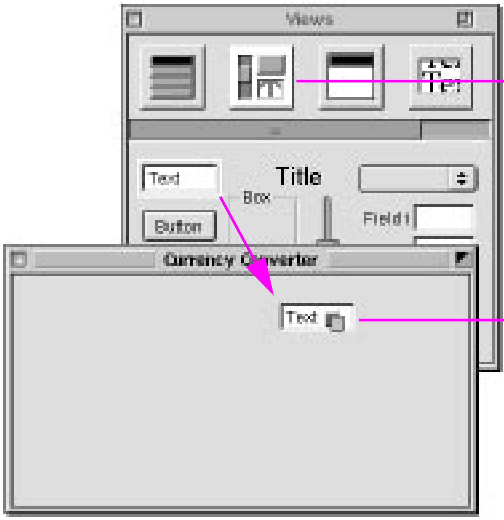
Fields and Buttons

Fields and buttons are the most common types of objects found on interfaces. Put these and other palette objects on the window using the “drag and drop” technique.

1 **Put a text field on the interface and resize and initialize it.**

Select the Views palette.

Drag a text field from the palette onto the window.



Click this icon to select the Views palette. This palette contains an assortment of commonly used Application Kit objects.

Drag a text field and drop it (that is, release the mouse button) over the “surface” of the window.

Move an object by dragging it around the surface of the window.

To initialize the text field, double-click “Text” and press Delete.

You must get rid of the word “Text” in this field; otherwise, that’s what the field will show when the nib file is loaded.

The text field should be longer so it can hold more digits (you’re dealing with millions here):

Lengthen the text field.

Drag a resize handle in the direction you want the object to grow.

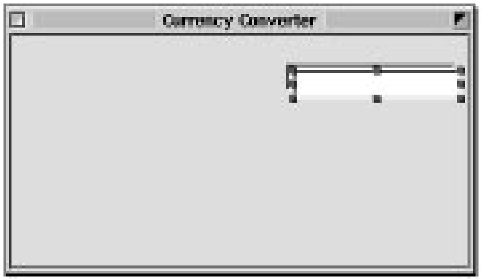
Currency Converter needs two more text fields, both the same size as the first. You have two options: you can drag another object from the palette and make it the same size, or you can duplicate the first object.

1 **Duplicate an object.**

Select the text field.

Choose Edit ► Copy.

Choose Edit ► Paste.



The new text field appears slightly offset from the original field. Reposition it below the first text field.

Get the third field from the palette and make it the same size as the first field.

1 **Make objects the same size.**

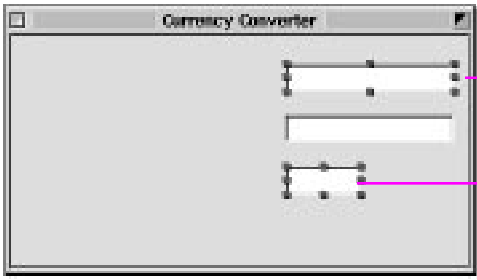
Drag a text field onto the window.

Delete “Text” from the text field.

Select the first text field.

Shift-click to select the new text field.

Choose Format ► Size ► Same Size



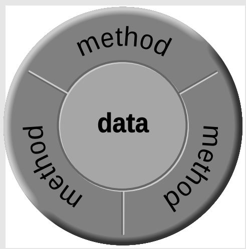
The first object you select should have the dimensions you want the other objects in the selection to take.

Shift-click multiple objects to include them in the selection.

You’re not done yet with these text fields. The bottom text field displays the result of the computation. It should not be editable and therefore should, by convention, have a non-white background.

Why an Object Looks Like a Jelly Donut

Or a lifesaver. Or a slashed tire. Or segmented unity. This book depicts objects as this symbol:



Why this unlikely shape?

This symbol illustrates *data encapsulation*, the essential characteristic of objects. An object consists of both data and procedures for manipulating that data. Other objects or external code cannot access that data directly, but must send *messages* to the object requesting its data.

An object’s procedures (called *methods*) respond to the message and may return data to the requesting object. As the symbol suggests, an object’s methods do the encapsulating, in effect mediating access to the object’s data. An object’s methods are also its interface, articulating the ways in which the object communicates with the world outside it.

The donut symbol also helps to convey the *modularity* of objects. Because an object encapsulates a defined set of data and logic, you can easily assign it to particular duties within a program. Conceptually, it is like a functional unit—for instance, “Customer Record”—that you can move around on a design board; you can then plot communication paths to and from other objects based on their interfaces.

See the appendix “Object Oriented Programming,” for a fuller description of data encapsulation, messages, methods, and other things pertaining to objects.

1 **Change the attributes of a text field.**

Select the third text field.

Choose Tools ► Colors.

Select the grayscale palette of the Color panel.

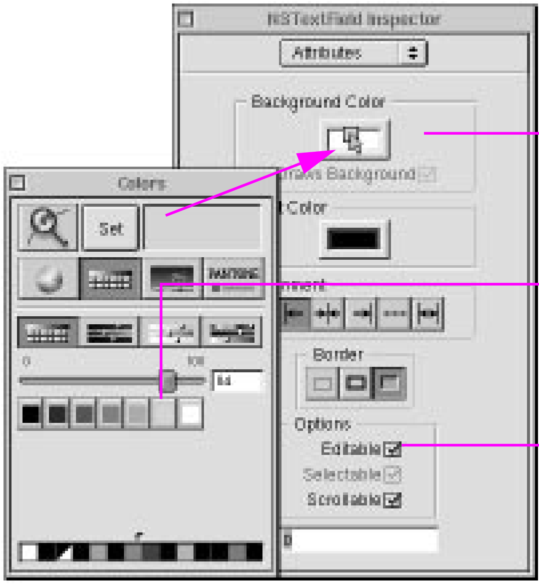
Select the color that is the same as the window background.

Choose Tools ► Inspector.

Select the Inspector panel's Attributes display.

Drag the gray color from the Color panel into the Background Color well.

Turn off the Editable and Scrollable options.



Drag the gray color into the well to apply the color to the selected object.

This color blends the text field into the window background.

With the Editable attribute turned off, users cannot alter the contents of the text field.

Keep Selectable as an option so users can copy and paste the contents to other applications.Keep Selectable as an option so users can copy and paste the contents to other applications.

The Views palette provides a "Title" object that you can easily adapt to be a text-field label. (The title object is actually a text field, set to have a gray background and no border, and to be non-editable and non-selectable.) Text in the title object is centered by default, but labels are often aligned from the right.

1 **Assign labels to the fields.**

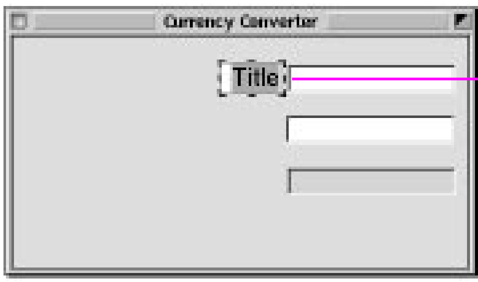
Drag a "Title" object onto the window.

Double-click to select the text.

Choose Inspector from the Tools menu.

Select the Attributes display.

Click the middle button under Alignment to align the text with the right edge of the text field.



The text is highlighted when it is selected.

The size of the text is rather large for a label, so change it. You set font family, typeface, and size with the standard OpenStep Font panel.

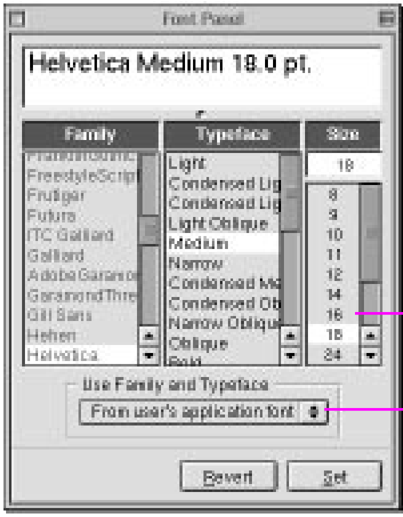
Make sure the object's text is selected.

Choose Format ▶ Font ▶ Font Panel.

Set the label text to 16 points.

Make two copies of the label.

Position all labels to the left of their text fields.



The font of the "Title" object is 18 points Helvetica. Click here and then click the Set button to set the font size to 16 points.

You should select the font that users request for applications in case the font you select is not available on the user's system.

When you cut and paste objects that contain text, like these labels, the object should be selected and not the text the object contains; if the text is selected, de-select it by clicking outside the text, then click the object again to select it.

Type the text of each label.

Double-click to select "Title," then type the text of the label in place of the selection.

1 **Add a button to the interface and initialize it.**

Drag the button object from the Views palette and put it on the lower-right corner of the window.

Make the button the same size as a text field.

Change the title of the button to "Convert".

You can resize buttons the same way you resize text fields or any other object on a window.

Double-click the title of the button to select the text.

Some Finishing Touches

Currency Converter's interface is almost complete. You've probably noticed that the final interface for Currency Converter (shown on the first page of this chapter) has a decorative line between the text fields and the button. This line is easy to make.

1 **Create a horizontal decorative line.**

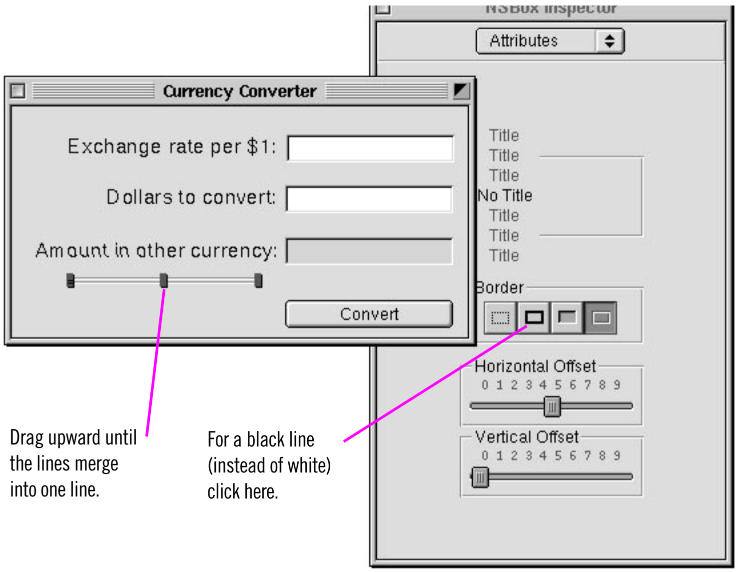
Drag a box object from the Views palette onto the interface.

Bring up the Attributes display for the box (Control-1), select No Title, and set the Vertical Offset to zero.

Drag the bottom-middle resize handle of the box upward until the horizontal lines meet.

Position the line above the button.

Drag the end points of the line until the line extends across the window.



Another finishing touch you might make is to align the text fields and labels in neat rows and columns. Interface Builder gives you several ways to align selected objects precisely on a window:

- Pressing arrow keys (with the grid off, the selected objects move one pixel)
- Using a reference object to put selected objects in rows and columns
- Specifying origin points in the Size display of the Inspector panel
- Using a grid (see side bar below)

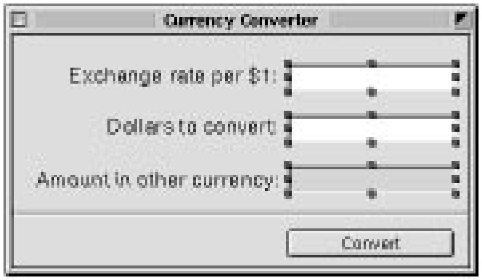
For Currency Converter, use the columns-and-rows technique.

1 **Align the text fields and labels in rows and columns.**

Select the three text fields and choose Format ▶ Align ▶ Make Column.

Select the first text field and its label and choose Format ▶ Align ▶ Make Row.

Repeat the last step for the second and third text fields and their labels.



COLUMNS

First select the object whose vertical position the other objects should adopt (the reference object).

Shift-click the other objects to include them in the selection. Making a column evens the spacing between objects in the selection.

ROWS

When you make a row, the selected objects rest on a common horizontal baseline.

The final step in composing the Currency Converter interface has more to do with behavior than appearance. You want the user to be able to tab from the first editable field to the second, and back again to the first.

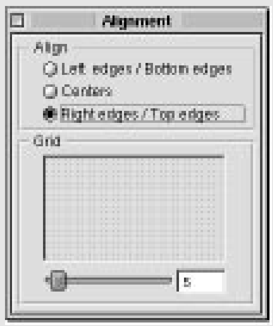
How does this happen? Objects such as windows and views can acquire a temporary status called *first responder*. The first responder is the object on the window that is the current focus of keyboard events. All objects inheriting

Aligning on a Grid

You can align objects on a window by imposing a grid on the window. When you move objects in this grid, they "snap" to the nearest grid intersection like nails to a magnet. You set the edges of alignment and the spacing of the grid (in pixels) in the Alignment panel. Choose Format ▶Align ▶ Alignment to display this panel.

Be sure the grid is turned on before you move objects (Format ▶Align ▶ Turn Grid On).

You can move selected user-interface objects in Interface Builder by pressing an arrow key. When the grid is turned on, the unit of movement is whatever the grid is set to (in pixels). When the grid is turned off, the unit of movement is one pixel.

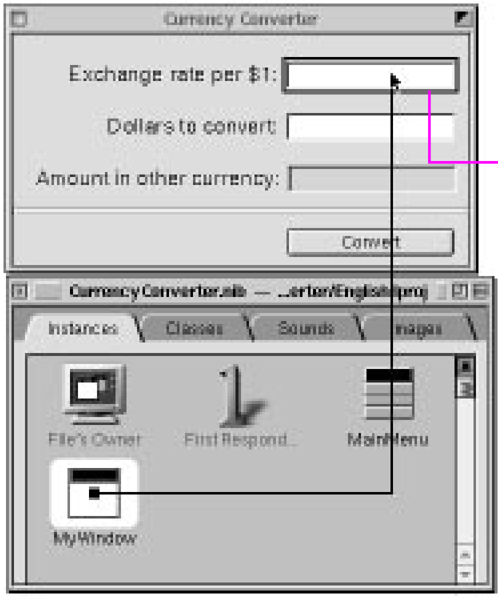


from NSWindow have an outlet named **initialFirstResponder** for designating the first responder when the window is first opened.

1 **Enable tabbing between text fields.**

Select the window icon in the nib file window.

Control-drag a connection line from the icon to the first text field ("Exchange Rate per $1").



When you press Control and drag the mouse from an object, a connection line is drawn.

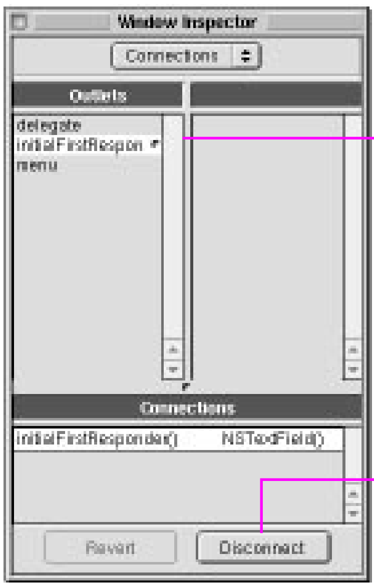
When a line encloses the destination object, release the mouse button.

Modifier keys (such as Control) may vary by platform. You can customize some key bindings to suit your habits. See the on-line *Programming Topics* for more on custom key bindings.

When you make a visual connection such as this, Interface Builder brings up the Connections display of the Inspector panel:

In the Connections display of the Window inspector (which appears automatically), select **initialFirstResponder**.

Click Connect.



Select this outlet (the dimple indicates an outlet that has been connected).

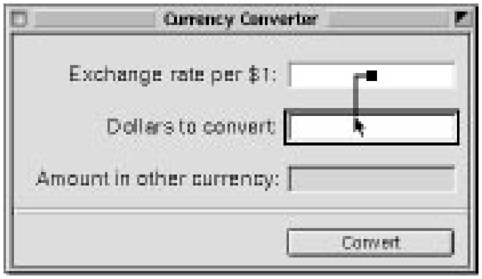
When you make a connection the title of this button toggles to "Disconnect."

View objects on Interface Builder's palettes have an outlet named **nextKeyView** for designating the next object to become first responder. If the object is a text field, and the Send Action on Enter attribute is checked in Interface Builder, the field receives keyboard events when users press the Tab key. Since the default tabbing order follows the position in the view hierarchy, if you want a different order you must connect fields through the **nextKeyView** variable.

Select the first text field.

In the Attributes display of the inspector, check the Send Action on Enter switch.

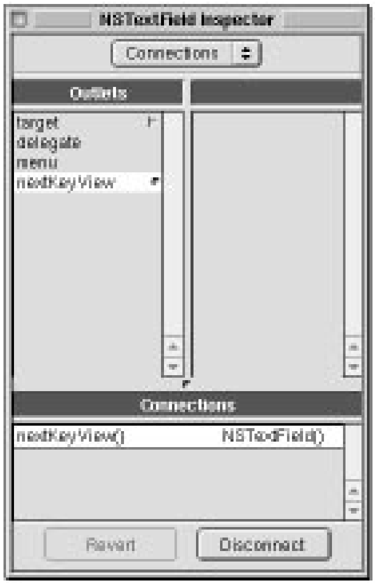
Control-drag a connection line from it to the second text field.



In the Inspector panel (Connections display) select **nextKeyView** and click Connect.

Repeat the same procedure, going from the second text field to button.

Repeat again, this time going from the button to the first text field.



Don't connect the **nextKeyView** outlet of the "Amount in Other Currency" field; this field is not supposed to be editable.

1 **Test the interface.**

Choose Document ► Save to save the interface to the nib file.

Choose Document ► Test Interface.

Try various operations in the interface (see suggestions on the following page).

When finished, choose Exit from the File menu.

The **initialFirstResponder** and **nextKeyView** variables are *outlets*. An outlet is the identifier of an object that another object stores as an instance variable. Outlets enable communication between objects. See page 40 for more information on outlets.

The CurrencyConverter interface is now complete. Interface Builder lets you test an interface without having to write one line of code.

An OpenStep Application — What You Get "For Free"

The simplest OpenStep application, even one without a line of code added to it, includes a wealth of features that you get "for free." You do not have to program these features yourself. You can see this when you test an interface in Interface Builder.

To enter test mode, choose Test Interface from the Document menu. Interface Builder simulates how your application (in this case, Currency Converter) would run, minus the behavior added by custom classes. Go ahead and try things out: move your windows, type in fields, click buttons.

Application and Window Behavior

In test mode Currency Converter behaves almost like any other application on the screen. Click elsewhere on the screen, and Currency Converter is deactivated, becoming totally or partially obscured by the windows of other applications.

.

Reactivate Currency Converter by clicking on its window or by double-clicking its icon (the default terminal icon) in the workspace. Then move the window around by its title bar. Here are some other tests you can make:

- Click the Edit menu. Its items appear and disappear when you release the mouse button, as with any application menu.
- Click the miniaturize button or choose the Hide command. Double-click the icon to get the application back.
- Click the close button, and the Currency Converter window disappears. (Choose Quit from the main menu and re-enter test mode to get the window back.)

If we had configured Currency Converter's window in Interface Builder to retain the resize bar, we could also resize it now. We could also have set the auto-resizing attributes of the window and its views so that the window's objects would resize proportionally to the resized window or would retain their initial size (see Interface Builder Help for details on auto-resizing).

Controls and Text

The buttons and text fields of Currency Converter come with many built-in behaviors. Click the Convert button. Notice how the button is highlighted momentarily.

.

If you had buttons of a different style, such as option buttons, they would also respond in characteristic ways to mouse clicks.

Now click in one of the text fields. See how the cursor blinks in place. Type some text and select it. Use the commands in the Edit menu to copy it and paste it in the other text field.

Do you recall the **nextKeyView** connections you made between Currency Converter's text fields? Insert the cursor in a text field, press the Tab key and watch the cursor jump from field to field.

When You Add Menu Commands

Interface Builder gives every new application a default main menu that includes the Info, Edit, Window, and Services menus. Some of these menus, such as Info, contain ready-made sets of commands. For example, with the Services menu (whose items are added by other applications at run time) you can communicate with other OpenStep applications, and with the Windows menu you can manage your application's windows.

Currency Converter needs only a few commands: the Quit and Hide commands and the Edit menu's Copy, Cut, and Paste commands. You can delete the unwanted commands if you wish. However, you could also add new ones and get "free" behavior. An application designed in Interface Builder can acquire extra functionality with the simple addition of a menu or menu command, *without* the need for compilation. For example:

- The Font submenu adds behavior for applying fonts to text in NSText objects, like the one in the scroll view object in the DataViews palette. Your application gets the Font panel and a font manager "for free."
- The Text submenu allows you to align text anywhere there is editable text, and to display a ruler in the NSText object for tabbing, indentation, and alignment.

Many objects that display text or images can print their contents as PostScript data. Later you'll learn how to add the Print menu command and have it invoke this capability.

See page 72 for an example of customizing OpenStep menus.

An OpenStep Application — The Possibilities

An OpenStep application can do an impressive range of things without a formidable programming effort on your part.

Document Management

Many applications create and manage repeatable, semi-autonomous objects called *documents*. Documents contain discrete sets of information and support the entry and maintenance of that information. A word-processing document is a typical example. The application coordinates with the user and communicates with its documents to create, open, save, close, and otherwise manage them.

The final tutorial in this book describes how to create an application based on a multi-document architecture.

File Management

An application can use the Open panel, which is created and managed by the Application Kit, to help the user locate files in the file system and open them. It can also use the Save panel to save information in files. OpenStep also provides classes for managing files in the file system (creating, comparing, copying, moving, and so forth) and for managing user defaults.

Communicating With Other Applications

OpenStep gives an application several ways to exchange information with other applications:

- **Pasteboard**: The pasteboard is a global facility for sharing information among applications. Applications can use the pasteboard to hold data that the user has cut or copied and may paste into another application.
- **Services**: Any application can access the services provided by another application, based on the type of selected data (such as text). An application can also provide services to other applications such as encryption, language translation, or record-fetching.
- **Drag-and-drop**: If your application implements the proper protocol, users can drag objects to and from the interfaces of other applications.

Custom Drawing and Animation

OpenStep lets you create your own custom views that draw their own content and respond to user actions. To assist you in this, OpenStep provides image-compositing and event-handling API as well as PostScript operators, operator functions, and client library functions.

Localization

OpenStep provides API and tool support for localizing the strings, images, sounds, and nib files that are part of an application

Editing Support

You can get several panels (and associated functionality) when you add certain menus to your application's menu bar in Interface Builder. These "add-ons" include the Font panel (and font management), the Color panel (and color management), and, although it's not a panel, the text ruler and the tabbing and indentation capabilities the Text menu brings with it.

Formatter classes enable your application to format numbers, dates, and other types of field values. Support for validating the contents of fields is also available.

Printing

With just a simple Interface Builder procedure, OpenStep automates simple printing of views that contain text or graphics. When a user clicks the control, an appropriate panel helps to configure the print process. The output is WYSIWYG.

Several Application Kit classes give you greater control over the printing of documents and forms, including features such as pagination and page orientation.

Help

You can create context-sensitive help for your application using Interface Builder, Project Builder, and an RTF text editor (such as TextEdit). If the user clicks an object on the application's interface while pressing a Help key, a small window containing concise information on the object is displayed. Your application can also incorporate Tool Tips—short descriptions that appear when the mouse pointer hovers over an object on the interface—and comprehensive Help in any format (for example, HTML).

Plug and Play

You can design some applications so that users can incorporate new modules later on. For example, a drawing program could have a tools palette: pencil, brush, eraser, and so on. You could create a new tool and have users install it. When the application is next started, this tool appears in the palette.

Defining the Classes of Currency Converter

Interface Builder is a versatile tool for application developers. It enables you not only to compose the application's graphical user interface, but it gives you a way to define much of the *programmatic* interface of the application's classes and to connect the objects eventually created from those classes.

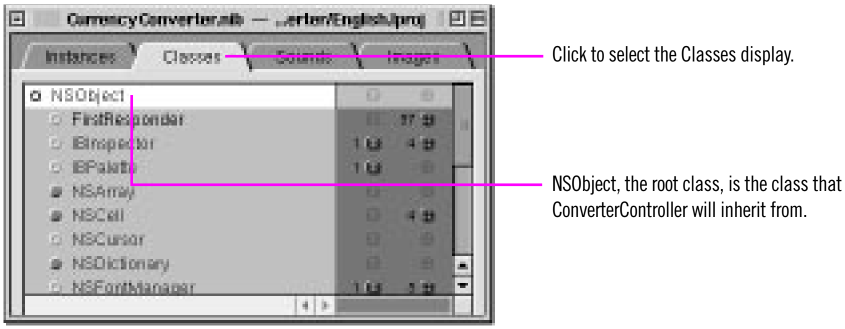
You must go to the Classes display of the nib file window to define a class. Once there, the first thing you must do is select the *superclass*, the class your new *subclass* will inherit from. Let's start with the ConverterController class.

1 **Specify a subclass.**

Go to the Classes display of the nib file window.

Select NSObject, the superclass of your custom classes.

Choose Classes ► Subclass.



After you choose the Subclass command, "MyNSObject" appears under "NSObject" highlighted.

Enter the name of the subclass: "ConverterController."

Press Return.

Now your class is established in the hierarchy of classes within the nib file. Next, specify the paths for messages travelling between the ConverterController object and other objects. In Interface Builder you specify these paths as *outlets* and *actions*.

Before You Go On

Here's some basic terminology:

Outlet An object held as an instance variable and typed as **id**. Objects in applications often hold outlets as part of their data so they can send messages to the objects referenced by the outlets. An outlet helps your program to track or manipulate something in the interface.

id The generic (or dynamic) type of objects (technically the address of an object).

Action Refers both to a message sent to an object when the user clicks a button or manipulates some other control object and to the method that is invoked.

Control object A user-interface object (a device) with which users can interact to affect events in the application. Control objects include buttons, text fields, forms, sliders, and browsers. All control objects inherit from NSControl.

See *Paths for Object Communication: Outlets, Targets, and Actions* on page 40. for a more detailed description of outlets and actions. See page 107 for more on control objects and their relation to cells and formatters.

Class Versus Object

To newcomers to the subject, explanations of object-oriented programming might seem to use the terms "object" and "class" interchangeably. Are an object and a class the same thing? And if not, how are they different? How are they related?

An object and a class are both programmatic units. They are closely related, but serve quite different purposes in a program.

First, classes provide a taxonomy of objects, a useful way of categorizing them. Just as you can say a particular tree is a pine tree, you can identify a particular object by its class. You can thereby know its purpose and what messages you can send it. In other words, a class describes the type of an object.

Second, you use classes to generate *instances*—or objects. Classes define the data structures and behavior of their instances, and at run time create and initialize these instances. In a sense, a class is like a factory, stamping out instances of itself when requested.

What especially differentiates a class from its instance is data. An instance has its own unique set of data but its class, strictly speaking, does not. The class defines the structure of the data its instances will have, but only instances can hold data.

A class, on the other hand, implements the behavior of **all** of its instances in a running program. The donut symbol used to represent objects is a bit misleading here, because it suggests that each object contains its own copy of code. This is fortunately not the case; instead of being duplicated, this code is shared among all current instances in the program.

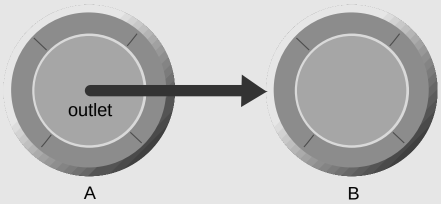
Implicit in the notion of a taxonomy is *inheritance*, a key property of classes. Classes exist in a hierarchical relationship to one another, with a *subclass* inheriting behavior and data structures from its *superclass*, which in turn inherits from its superclass.

See the appendix, "Object-Oriented Programming," for more on these and other aspects of classes.

Paths for Object Communication: Outlets, Targets, and Actions

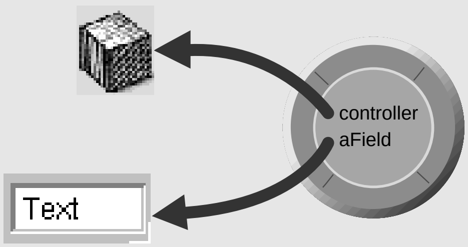
Outlets

An outlet is an instance variable that identifies an object.



You can communicate with other objects in an application by sending messages to outlets.

An outlet can reference any object in an application: user-interface objects such as text fields and buttons, windows and panels, instances of custom classes, and even the application object itself.



Outlets are declared as:

```
id  variableName;
```

You can use **id** as the type for any object; objects with **id** as their type are *dynamically typed*, meaning that the class of the object is determined at run time. You can statically type an object as a pointer to a class name, and you can declare these objects as instance variables. But statically typed objects are typically not outlets. What distinguishes outlets is their relationship to Interface Builder.

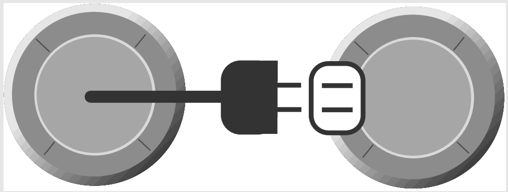
Interface Builder can "recognize" outlets in code by their declarations, and it can initialize outlets. You usually set an outlet's value in Interface Builder by drawing connection lines between objects. There are ways other than outlets to reference objects in an application, but outlets and Interface Builder's facility for initializing them are a great convenience.

When You Make a Connection in Interface Builder

As with any instance variable, outlets must be initialized at run time to some reasonable value—in this case, an object's identifier (**id** value). Because of Interface Builder, an application can initialize outlets when it loads a nib file.

When you make a connection in Interface Builder, a special connector object holds information on the source and destination objects of the connection. (The source object is the object with the outlet.) This connector object is then stored in the nib file. When a nib file is loaded, the application uses the connector object to set the source object's outlet to the **id** value of the destination object.

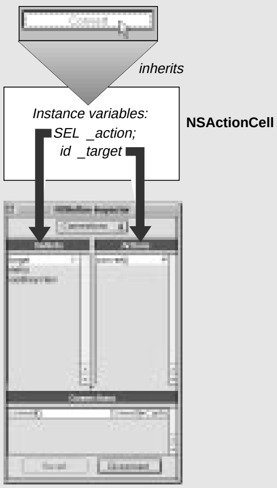
It might help to understand connections by imagining an electrical outlet (as used in the Classes display of the nib file window) embedded in the destination object. Also picture an electrical cord extending from the outlet in the source object. Before the connection is made the cord is unplugged and the value of the outlet is undefined; after the connection is made (the cord is plugged in), the **id** value of the destination object is assigned to the source object's outlet



Target/Action in Interface Builder—What's Going On

As you'll soon find out, you can view (and complete) target/action connections in Interface Builder's Connections inspector. This inspector is easy to use, but the relation of target and action in it might not be apparent. First, **target** *is an outlet of a cell object* that identifies the recipient of an action message. Well (you say) what's a cell object and what does it have to do with a button?

One or more cell objects are always associated with a control object (that is, an object inheriting from NSControl, such as a button). Control objects "drive" the invocation of action methods, but they get the target and action from a cell. NSActionCell defines the target and action outlets, and most kinds of cells in the Application Kit inherit these outlets



For example, when a user clicks the Convert button of Currency Converter, the button gets the required information from its cell and sends the message **convert:** to the target outlet, which is an instance of your custom class ConverterController.

In the Actions column of the Connections inspector are all action methods defined by the class of the target object and known by Interface Builder. Interface Builder identifies action methods because their declarations follow the syntax:

```
- (void)doThis:(id)sender;
```

It looks in particular for the argument **sender**.

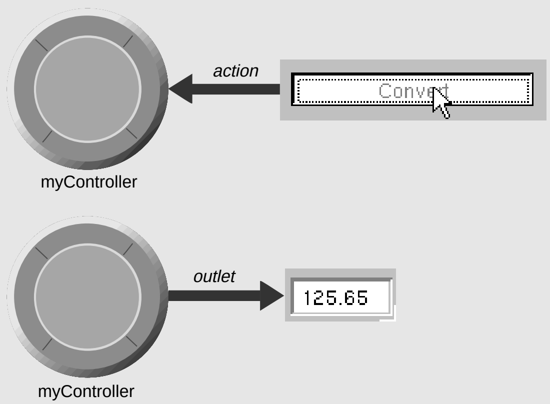
Which Direction to Connect?

Usually the outlets and actions that you connect belong to a custom subclass of NSObject. For these occasions, you need only follow a couple simple rules to know which way to draw a connection line in Interface Builder:

- To make an action connection, draw a line *to* the custom instance *from* a control object in the user interface, such as a button or a text field.
- To make an outlet connection, draw a line *from* the custom instance *to* another object in the application.

Another way to clarify connections is to consider who needs to find whom. With outlets, the custom object needs to find some other object, so the connection is from the custom object to the other object. With actions, the control object needs to find the custom object, so the connection is from the control object.

These are only rules of thumb for the common case, and do not apply in all circumstances. For instance, many OpenStep objects have a **delegate** outlet; to connect these, you draw a connection line from the OpenStep object to your custom object.



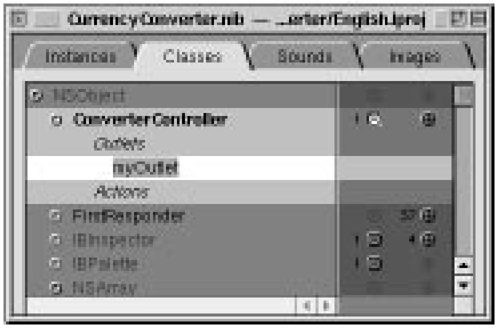
1 **Define your class's outlets.**

In the nib file window, click the electrical-outlet icon to the right of the class.

Choose Classes ► Add Outlet.

Type the name of the outlet in place of the selected "myOutlet." Name the first outlet **rateField**.

Press Return.



Repeat the last three steps to define two other outlets:

dollarField
totalField

ConverterController has one action method, **convert:**. When the user clicks the Convert button, a **convert:** message is sent to the target, ConverterController.

1 **Define your class's actions.**

In the Classes display of the nib file window, click the crosshairs icon.

Choose Classes ► Add Action.

Type the name of the action method, **convert:**.

Press Return.

Before You Go On

Exercise: ConverterController needs to access the text fields of the interface, so you've just provided outlets for that purpose. But ConverterController must also communicate with the Converter class (yet to be defined). To enable this communication, add an outlet named **converter** to ConverterController.

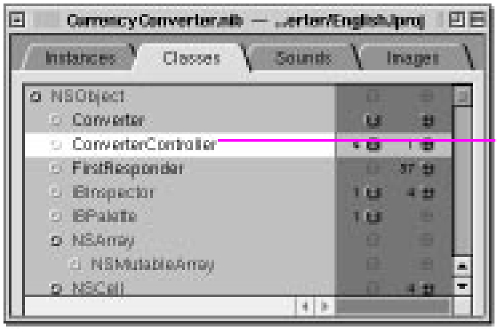
Connecting ConverterController to the Interface

As the final step of defining a class in Interface Builder, you create an instance of your class and connect its outlets and actions.

1 **Generate an instance of the class.**

In the Classes display, select the ConverterController class.

Choose the Classes ► Instantiate.



Click any other class name to collapse the outlets and actions of the subclass you're working on. If they are already collapsed, make sure your subclass is selected.

Note: The Instantiate command does not generate a true instance of ConverterController but creates a stand-in object used for establishing connections. When the nib file's contents are unarchived, Interface Builder will create true instances of these classes and use the proxy objects to establish the outlet and action connections.

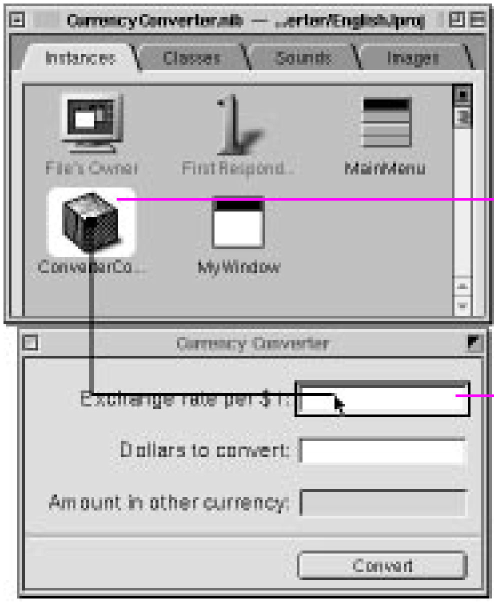
When you instantiate a class (that is, create an instance of it), Interface Builder switches to the Instances display and highlights the new instance, which is named after the class.

Now you can connect this ConverterController object to the user interface. By connecting it to specific objects in the interface, you initialize your outlets. ConverterController will use these outlets to get and set values in the interface.

1 Connect the custom class to the interface via its outlets.

In the Instances display of the nib file window, Control-drag a connection line from the ConverterController instance to the first text field.

When the field is outlined in black, release the mouse button.

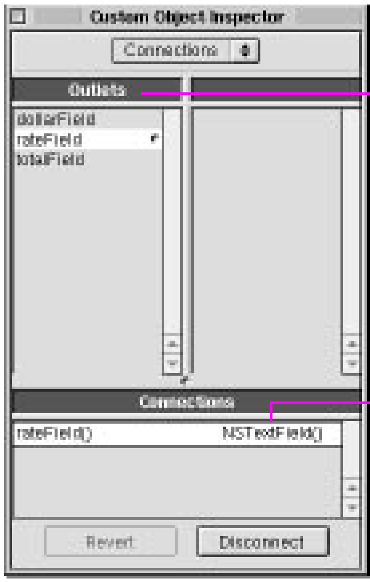


Interface Builder brings up the Connections display of the Inspector panel. This display shows the outlets you have defined for ConverterController.

In the Connections display, select the outlet that corresponds to the first field (**rateField**).

Click the Connect button.

Following the same steps, connect ConverterController's **dollarField** and **totalField** outlets to the appropriate text fields.

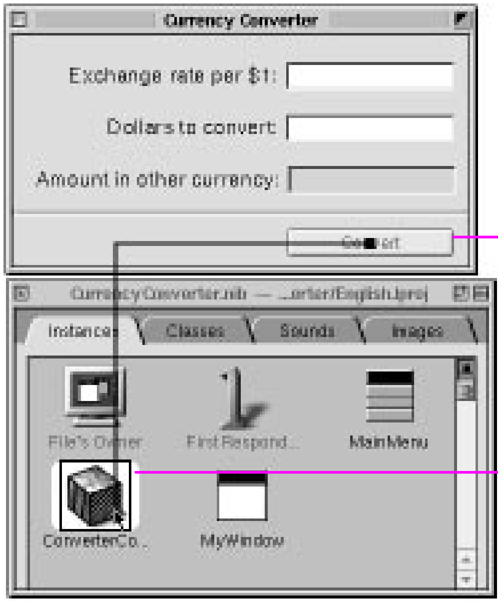


To receive action messages from the user interface—to be notified, for example, when users click a button—you must connect the control objects that emit those messages to CurrencyConverter. The procedure for connecting actions is similar to that for outlets, but with one major difference. When you connect an action, always start the connection line from a *control object* (such as a button, text field, or form) that sends an action message; you usually end the connection at an instance of your custom class. That instance is the *target* outlet of the control object.

1 **Connect the interface's controls to the custom object through the defined actions.**

Control-drag a connection line from the Convert button to the ConverterController instance in the nib file window.

When the instance is outlined in black, release the mouse button.



The source object of an action connection must be a control object.

When a black line encloses an object, it will be selected as the destination object of the connection if you release the mouse button.

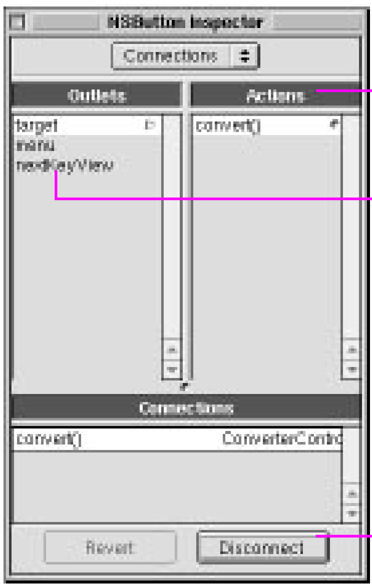
The Connections display of the Inspector panel shows the action methods you have specified for ConverterController.

In the Connections display, make sure **target** in the Outlets column is selected.

Select **convert:** in the Actions column.

Click the Connect button.

Save the CurrencyConverter nib file (Document ▶ Save).



If you had defined other actions for Converter Controller, they would have appeared in this column.

Interface Builder allows you to set these outlets directly for buttons.

Make sure that you click here to establish the connection.

You've finished defining the classes of Currency Converter—almost.

Before You Go On

Exercise: While connecting ConverterController's outlets, you probably noticed that one outlet remains unconnected: **converter**. This outlet identifies an instance of the Converter class in the Currency Converter application, but this instance doesn't exist yet.

Define the Converter class. This should be pretty easy because Converter, as you might recall, is a model class within the Model-View-Controller paradigm. Since instances of this type of class don't communicate directly with the interface, there is no need for outlets or actions. Here are the steps to be completed:

1. In the Classes display, make Converter a subclass of NSObject.
2. Instantiate the Converter class.
3. Make an outlet connection between ConverterController and Converter.

When you are finished, save **CurrencyConverter.nib.**

Optional Exercise

Text fields and action messages: Users can also activate the Convert button by pressing the Return key. In Currency Converter this key event occurs when the cursor is in a text field. Text fields are control objects just as buttons are; when the user presses the Return key and the cursor is in a text field, an action message is sent to a target object if the action is defined and the proper connection is made.

Connect the second text field (that is, the one with the "Dollars to Convert" label) to the **convert:** action method of ConverterController. You won't be disconnecting the prior action connection because multiple control objects in an interface can invoke the same action method.

Optionally, you can connect the second text field to the Convert button via the latter's **performClick:** action method. This method simulates a mouse click on the button and consequently invokes the action method of the button's target.

Implementing the Classes of Currency Converter

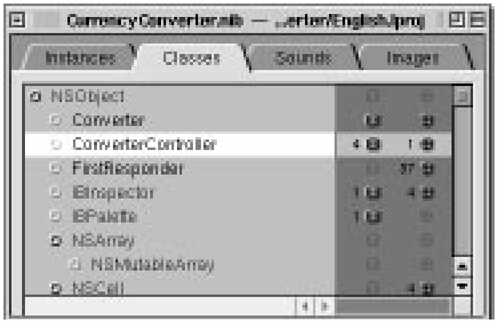
Interface Builder generates source code files from the (partial) class definitions you've made. These files are "skeletal," in the sense that they contain little more than essential Objective-C directives and the class-definition information. You'll usually need to supplement these files with your own code.

1 **In Interface Builder, generate header and implementation files.**

Go to the Classes display of the nib file window.

Select the ConverterController class.

Choose Classes ► Create Files.



Make sure your class is selected before you choose Create Files.

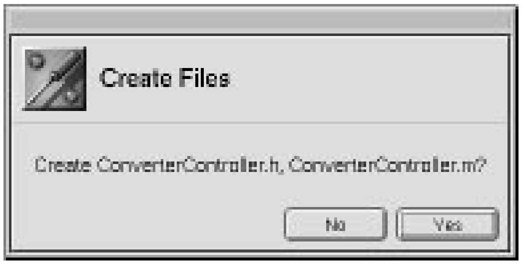
Interface Builder then displays two attention panels, one after the other:

Click Yes in response to a "create files" attention panel.

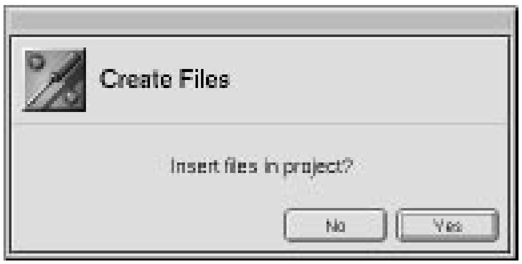
Click Yes in response to an "insert files in project" attention panel.

Repeat for the Converter class.

Save the nib file.



Click Yes to confirm that you want Interface Builder to generate the header and implementation files for your class.



Click Yes to confirm that you want the source code files added to the project. If, for example, you wanted to add the files to another project, you would click No.

Now we leave Interface Builder for this application. You'll complete the application using Project Builder.

Objective-C Quick Reference

The Objective-C language is a superset of ANSI C with special syntax and run-time extensions that make object-oriented programming possible. Objective-C syntax is uncomplicated, but powerful in its simplicity. You can mix standard C and even C++ code with Objective-C code.

The following summarizes some of the more basic aspects of the language. See *Object-Oriented Programming and the Objective-C Language* for complete details. Also, see "Object-Oriented Programming" in the appendix for explanations of terms that are italicized.

Declarations

- Dynamically type objects by declaring them as **id**:

  ```
  id myObject;
  ```

  Since the class of dynamically typed objects is resolved at run time, you can refer to them in your code without knowing beforehand what class they belong to. Type outlets in this way as well as objects that are likely to be involved in *polymorphism* and *dynamic binding*.

- Statically type objects as a pointer to a class:

  ```
  NSString *mystring;
  ```

  You statically type objects to obtain better compile-time type checking and to make code easier to understand.

- Declarations of *instance methods* begin with a minus sign (-); *class methods* begin with a plus sign (+):

  ```
  - (NSString *)countryName;
  + (NSDate *)calendarDate;
  ```

- Put the type of value returned by a method in parentheses between the minus sign (or plus sign) and the beginning of the method name. (See above example.) Methods returning no explicit type are assumed to return **id**. Methods that return nothing should have a return type of **void**.

- Method argument types are in parentheses and go between the argument's *keyword* and the argument itself:

  ```
  - (id)initWithName:(NSString *)name
      andType:(int)type;
  ```

  Be sure to terminate all declarations with a semicolon.

- By default, the scope of an instance variable is protected, making that variable directly accessible only to objects of the class that declares it or of a subclass of that class. To make an instance variable private (accessible only within the declaring class), insert the **@private** directive before the declaration.

Messages and Method Implementations

- Methods are procedures implemented by a class for its objects (or, in the case of class methods, to provide functionality not tied to a particular instance). Methods can be public or private; public methods are declared in the class's header file (see above). Messages are invocations of an object's method that identify the method by name.

- Message expressions consist of a variable identifying the receiving object followed by the name of the method you want to invoke; enclose the expression in brackets.

  ```
  [anObject doSomethingWithArg:this];
  ```

  receiver *method to invoke (with possible arguments)*

  As in standard C, terminate statements with a semicolon.

- Messages often get values returned from the invoked method; you must have a variable of the proper type to receive this value on the left side of an assignment.

  ```
  int result = [anObj calcTotal];
  ```

- You can nest message expressions inside other message expressions. This example gets the window of a form object and makes the returned NSWindow object the receiver of another message.

  ```
  [[form window]
      makeKeyAndOrderFront:self];
  ```

- A method is structured like a function. After the full declaration of the method comes the body of the implementing code enclosed by braces.

- Use **nil** to specify a null object; this is analogous to a null pointer. Note that some OpenStep methods do not accept **nil** as an argument.

- A method can usefully refer to two implicit identifiers: **self** and **super**. Both identify the object receiving a message, but they affect differently how the method implementation is located: **self** starts the search in the receiver's class whereas **super** starts the search in the receiver's superclass. Thus,

  ```
  [super init];
  ```

  causes the **init** method of the superclass to be invoked.

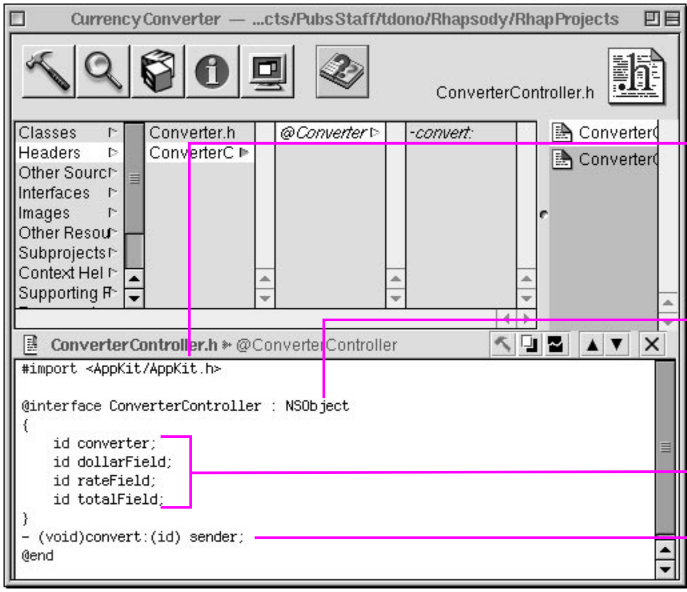
- In methods you can directly access the instance variables of your class's instances. However, *accessor methods* are recommended instead of direct access, except in cases where performance is of paramount importance. Chapter 4, "Travel Advisor Tutorial," describes accessor methods in greater detail.

1 **Examine an interface (header) file in Project Builder.**

Click Project Builder's main window to activate it.

Select Headers in the project browser.

Select **ConverterController.h**.



For the classes of an application, Project Builder imports the Application Kit header files, which import the Foundation header files.

Interface definitions begin with **@interface** and the class name. The superclass appears after the colon.

Instance variables go between the braces.

Method declarations follow the second brace and the definition ends with **@end**.

You can add instance variables or method declarations to a header file generated by Interface Builder. This is commonly done, but it isn't necessary in ConverterController's case. But we do need to add a method to the Converter class that the ConverterController object can invoke to get the result of the computation. Let's start by declaring the method in **Converter.h**.

1 **Add a method declaration.**

Select **Converter.h** in the project browser.

Insert a declaration for **convertAmount:byRate:**.

```
#import <AppKit/AppKit.h>

@interface Converter:NSObject
{
}
- (float)convertAmount:(float)amt byRate:(float)rate;

@end
```

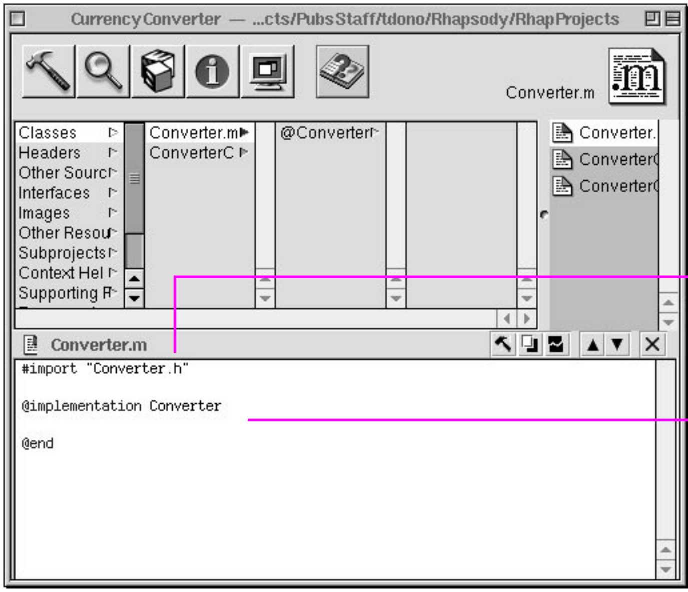
This declaration states that **convertAmount:byRate:** takes two arguments of type **float**, and returns a **float** value. When parts of a method name have colons, such as **convertAmount:** and **byRate:**, they are *keywords* which introduce arguments. (These are keywords in a sense different from keywords in the C language.) All declarations of instance methods begin with a dash (-), followed by a space.

Now you need to update both implementation files. First examine **Converter.m**.

1 Examine an implementation file.

Click Classes in the project browser.

Select **Converter.m**.



The associated header file is imported automatically.

Begin the implementation section with **@implementation** and the class name. Method implementations go between here and **@end**.

For this class, implement the method declared in **Converter.h**. Between **@implementation Converter** and **@end** add the following code:

1 Implement the classes.

Type the code at right between **@implementation** and **@end** in **Converter.m**.

```
- (float)convertAmount:(float)amt byRate:(float)rate
{
    return (amt * rate);
}
```

The method simply multiplies the two arguments and returns the result. Simple enough. Next update the "empty" implementation of the **convert:** method that Interface Builder generated.

Select **ConverterController.m** in the project browser.

Update the **convert**: method as shown by the example.

Import **Converter.h**.

```
- (void)convert:(id)sender
{
    float rate, amt, total=0.0;

    amt = [dollarField floatValue];                        A
    rate = [rateField floatValue];
    total = [converter convertAmount:amt byRate:rate];     B
    [totalField setFloatValue:total];                      C
    [rateField selectText:self];                           D
}
```

The **convert:** method does the following:

A Gets the floating-point values typed into the rate and dollar-amount fields.

B Invokes the **convertAmount:byRate:** method and gets the returned value.

C Uses **setFloatValue:** to write the returned value in the Amount in Other Currency text field (**totalField**).

D Sends **selectText:** to the rate field; this selects any text in the field or, if there is no text, inserts the cursor so the user can begin another calculation.

Be sure to import **Converter.h** (that is, include the directive **#import "Converter.h"**). ConverterController invokes a method defined in the Converter class, so it needs to be aware of the method's declaration.

Before You Go On

Each line of the **convert:** method shown above, excluding the declaration of **float**s, is a message. The "word" on the left side of a message expression identifies the object receiving the message (called the "receiver"). These objects are identified by the outlets you defined and connected. After the receiver comes the name of the method that the sending object (called the "sender") wants to invoke. Messages often result in values being returned; in the above example, the local variables **rate**, **amt**, and **total** hold these values.

Before you build the project, add a small bit of code to **ConverterController.m** that will make life a little easier for your users. When the application starts up, you want Currency Converter's window to be selected and the cursor to be in the Exchange Rate per $1 field. We can do this only after the nib file is unarchived, which establishes the connection to the text field **rateField**. To enable set-up operations like this, **awakeFromNib** is sent to all objects when unarchiving concludes. Implement this method to take appropriate action.

1 **Implement the awakeFromNib method.**

Type the code shown at right.

Save all code files.

```
- (void)awakeFromNib
{
    [rateField selectText:self];                        A
    [[rateField window] makeKeyAndOrderFront:self];     B
}
```

A You've seen the **selectText:** message before, in the **convert:** implementation; it selects the text in the text field that receives the message, inserting the cursor if there is no text.

B The **makeKeyAndOrderFront:** message does as it says: It makes the receiving window the key window and puts it before all other windows on the screen. This message also *nests* another message, **[rateField window]**. This message returns the window to which the text field belongs, and the **makeKeyAndOrderFront:** method is then sent to this returned object.

What Happens When You Build an Application

By clicking the Build button in Project Builder, you run the build tool. By default, the build tool is **make**, but it can be any build utility that you specify as a project default in Project Builder. The build tool coordinates the compilation and linking process that results in an executable file. It also performs other tasks needed to build an application.

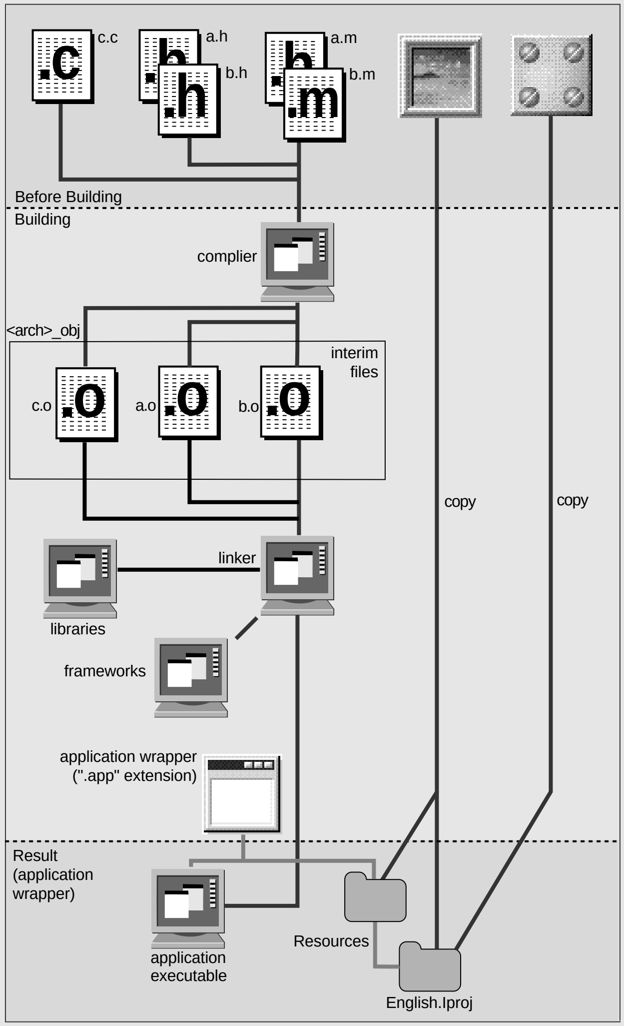
The build tool manages and updates files based on the dependencies and other information specified in the project's makefiles. Every application project has three makefiles: **Makefile**, **Makefile.preamble**, and **Makefile.postamble**. **Makefile** is maintained by Project Builder—don't edit it directly—but you can modify the other two to customize your build.

The build tool invokes the compiler, passing it the source code files of the project. Compilation of these files (Objective-C, C++, and standard C) produces machine-readable object files for the architecture or architectures specified for the build. The build utility puts these files in an architecture-specific subdirectory of **dynamic_obj**.

In the linking phase of the build, the build tool executes the linker, passing it the libraries and frameworks to link against the object files. Frameworks and libraries contain precompiled code that can be used by any application. Linking integrates the code in libraries, frameworks, and object files to produce the application executable file.

The build tool also copies nib files, sound, images, and other resources from the project to the appropriate localized or non-localized locations in the application wrapper.

An application wrapper on Windows is a directory with an extension of ".app". It contains the application executable and the resources needed by that executable.



Building the Currency Converter Project

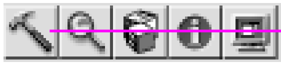
The Build process in Project Builder compiles and links the application guided by the information stored in the project's makefiles. You must begin builds from the Project Build panel.

1 **Build the project.**

Save source code files and any changes to the project.

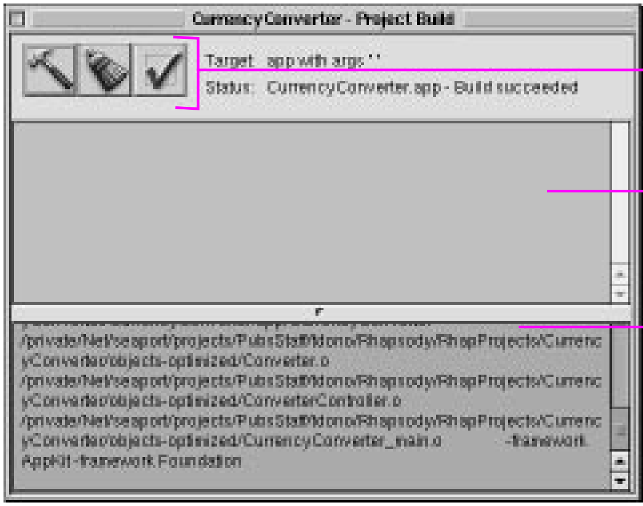
Click the Build button on the main window (icon shown at right).

Click the Build button on the Project Build panel (same icon).



Displays Project Build panel.

When you click the Build button on the main window, the Project Build panel is displayed.



Build, Clean, and Build Options buttons.

Build error browser.

Detailed build results.

You can begin building at any time by pressing Command-B.

When you click the Build button on the Project Build panel, the build process begins; Project Builder logs the build's progress in the lower split view. When Project Builder finishes—and encounters no errors along the way—it displays "Build succeeded."

You don't have to maintain makefiles in Project Builder. It updates **Makefile** according to the variables specified through its user interface. You can customize the build process by modifying the **Makefile.preamble** and **Makefile.postamble** files. For more information on customizing these files, see the on-line Help for Project Builder and Interface Builder.

Of course, rare is the project that is flawless from the start. Project Builder is likely to catch some errors when you first build your project. To see the error-checking features of Project Builder, introduce a mistake into the code.

1 **Build the project after correcting errors.**

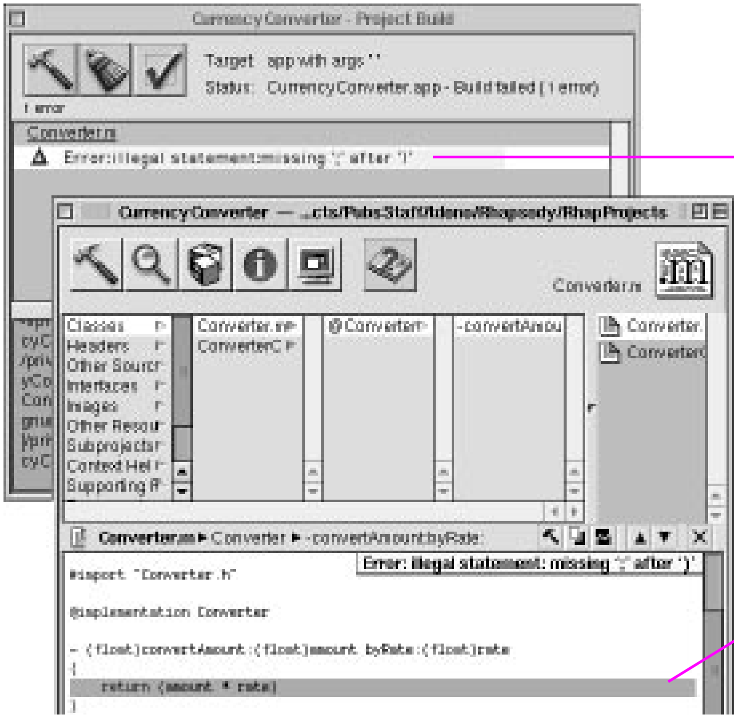
Delete a semicolon in the code, creating an error.

Click the Build button on the Project Build panel.

Click the error-notification line that appears in the build error browser (upper split view).

Fix the error in the code.

Re-build.



To navigate to an error in a code file, click the line describing the error.

Project Builder then highlights the line that contains the error.

You can use Project Builder's graphical debugger or **gdb** to track bugs down. See "Using the Graphical Debugger" on page 110 for an overview of the graphical debugger.

Where To Go For Help

Help on Development Tools

Project Builder and Interface Builder provide context-sensitive help on the details of their use. To activate context-sensitive help, Help-click a control, field, menu command, or other areas of the application. A small window appears that briefly describes the selected object. (The next click dismisses the window.)

These applications also provide Tool Tips, short descriptions of parts of the interface that briefly appear when the mouse pointer hovers over those areas. You can turn Tool Tips off.

Project Builder and Interface Builder also provide comprehensive task-based Help, accessible from the Apple menu.

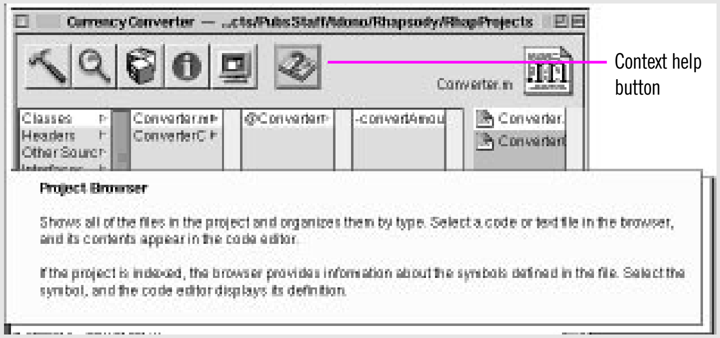
Help on APIs

Project Builder gives you several ways to get information on OpenStep APIs when you're developing an application.

Project Find. The Project Find panel allows you to search for definitions of, and references to, classes, methods, functions, constants, and other symbols in your project. Since it is based on project indexing, searching is quick and thorough and leads directly to the relevant code. Help for Interface Builder or Project Builder contains full task-based instructions for using Project Find.

Reference Documentation Lookup. If the results of a search using Project Find include OpenStep symbols, you can easily get related reference documentation that describes that symbol. See "Finding Information Within Your Project" on page 96 for instructions on the use of this feature.

Frameworks. Under Frameworks in the project browser, you can browse the header files and documentation related to OpenStep frameworks within Project Builder. The Application Kit and Foundation frameworks always are included by default for application projects.



Rhapsody Technical Documentation

Most Rhapsody programming documentation is located in **/System/Documentation/Developer**.

Reference

- API reference documentation. Includes specifications of classes, protocols, functions, types, and constants. This documentation is located in the appropriate Rhapsody frameworks, except for information that is common to all frameworks (**Reference**).
- Development tools reference. Covers the compiler, the debugger, and other tools (**Reference/DevToolsRef**).

Tasks and Concepts

- *Discovering OpenStep: A Developer Tutorial* (this manual).
- "Object-Oriented Programming and the Objective-C Language": an on-line reference and users guide for Objective-C.
- "Topics in OpenStep Programming" contains concepts and programming procedures.

Run Currency Converter

Congratulations. You've just created your first OpenStep application. Find **CurrencyConverter.app** in the Workspace, launch it, and try it out. Enter some rates and dollar amounts and click Convert. Also, select the text in a field and choose the Services menu; this menu now lists the other applications that can do something with the selected text.

Of course, the more complex an application is, the more thoroughly you will need to test it. You might discover errors or shortcomings that necessitate a change in overall design, in the interface, in a custom class definition, or in the implementation of methods and functions.

Although it's a simple application, Currency Converter still introduced you to many of the concepts, tools, and skills you'll need to develop OpenStep applications. Let's review what you've learned:

- Composing a graphical user interface (GUI) with Interface Builder
- Testing the interface
- Designing an application using the Model-View-Controller paradigm
- Specifying a class's outlets and actions
- Connecting the class instance to the interface via its outlets and actions
- Class implementation basics
- Building an application and error resolution

Optional Exercise

Nesting Messages: You can nest message expressions; in other words, you can use the value returned by a message as the receiver of another message or as a message argument. It is thus possible to rewrite the first three messages of the ConverterController's **convert:** method as one statement:

```
total = [converter convertAmount:[dollarsField floatValue]
        byRate:[rateField floatValue]];
```

It is possible to go even further. Try to incorporate the fourth message (**[totalField setFloatValue:total]**) of the **convert:** method into the above statement.

Travel Advisor Tutorial Chapter 3

3

What You'll Learn

Using forms and table views

Grouping objects

Adding images to applications

Formatting and validating fields

Simple printing

Object allocation and initialization

Using collection objects and string objects

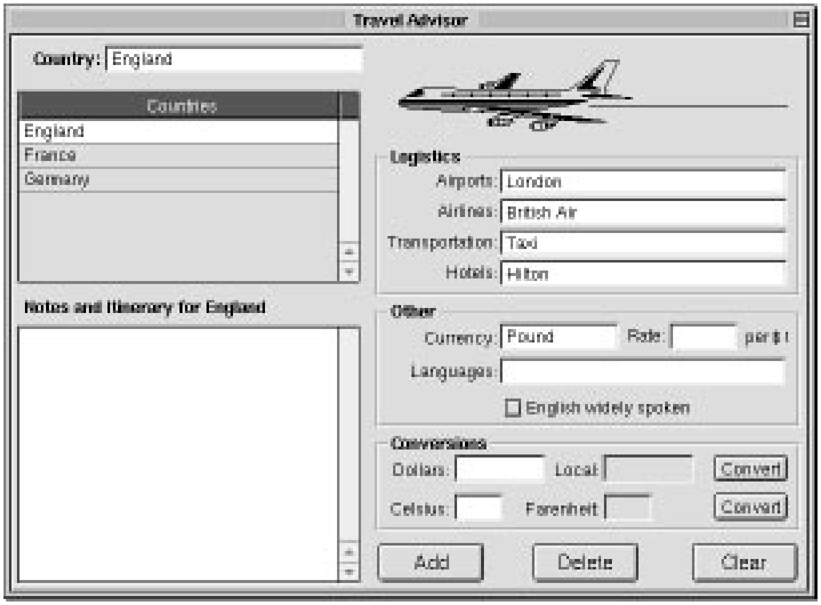
Delegation and notification

Archiving and unarchiving objects

Object ownership, retention, and disposal

Using the graphical debugger

Finding project information



You can find the Travel Advisor project in the **AppKit** *subdirectory of* **/System/Developer/Examples.**

Chapter 3
A Forms-Based Application

In this chapter you create Travel Advisor, a considerably more complex application than Currency Converter. Travel Advisor is a forms-based application used for entering, viewing, and deleting records on countries that the user travels to. Users enter a country name and information associated with that country. When they click Add, the country appears in the table below the country name. They can select countries in the table, and the information on that country appears in the forms. The application also performs temperature and currency conversions.

Travel Advisor — An Overview

This chapter presents a lot of information on OpenStep programming. Among other things, you'll learn how to:

- Use several new objects on Interface Builder's palettes.
- Assign an icon to an application.
- Print the contents of a view.
- Use collection objects (NSArray and NSDictionary) and NSString objects.
- Archive and unarchive object data.
- Format and validate field contents.
- Manage events through delegation.
- Quickly find information related to your project.
- Use Project Builder's graphical debugger.

Perhaps most interestingly, you will *reuse* the Converter class you implemented in the previous tutorial.

Note: You can find the TravelAdvisor project in the **AppKit** subdirectory of **/System/Developer/Examples**.

The Design of Travel Advisor

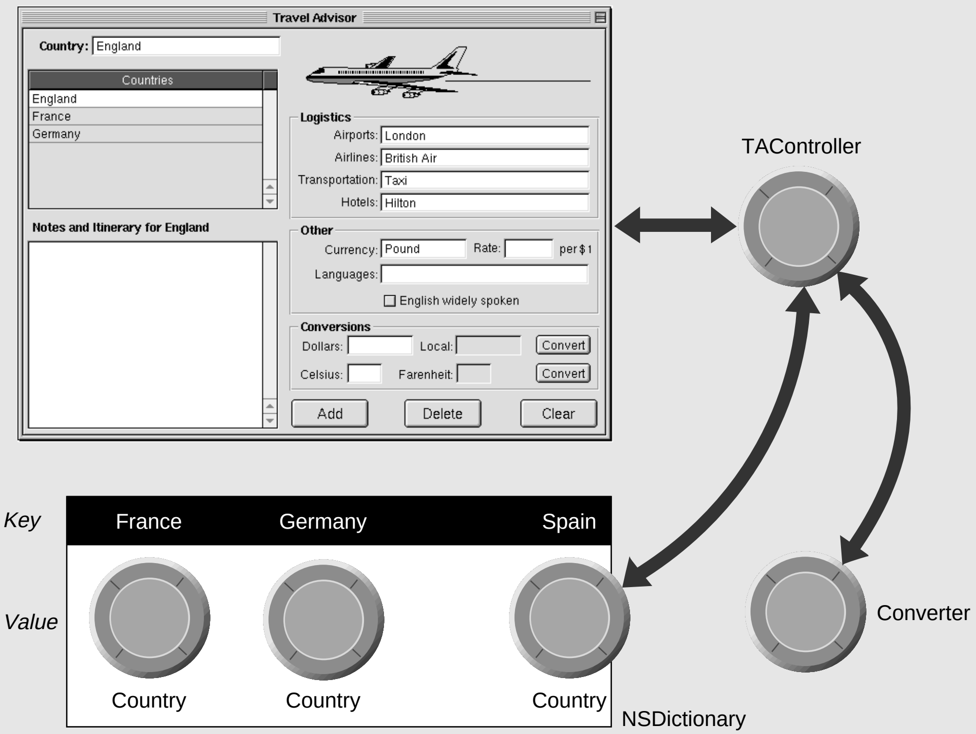
Travel Advisor is much like Currency Converter in its basic design. Like Currency Converter, it's based on the Model-View-Controller paradigm. A controller object (TAController) manages a user interface comprised of Application Kit objects. Also as before, the controller sends a message to the Converter object to get the result of a computation. In other words, the Converter object is reused.

Travel Advisor's view objects, in terms of Model-View-Controller, are all off-the-palette Application Kit objects, so the following discussion concentrates on those parts of the design distinctive to Travel Advisor.

Model Objects

Travel Advisor's design is more interesting and dynamic than Currency Converter's because it must display a unique set of data depending on the country the user selects. To make this possible, the data for each country is stored in a Country object. These objects encapsulate data on a country (in a sense, they're like records in a relational database). The application can manage potentially hundreds of these objects, tracking each without recourse to a "hardwired" connection.

Another model object in the application is the instance of the Converter class. This instance does not hold any data, but does provide some specialized behavior.



Controller

The controller object for the application is TAController. Like all controller objects, TAController is responsible for mediating the flow of data between the user interface (the View part of the paradigm) and the model objects that encapsulate that data: the Country objects. Based on user choices in the interface, TAController can find and display the requested Country object; it can also save changes made by users to the appropriate Country object.

What makes this possible is an NSDictionary object (called a *dictionary* from here on). A dictionary is a container that stores objects and permits their retrieval through key-value associations. The key is some identifier paired with an object in the dictionary (the object often holds the identifier as one of its instance variables). To get the object, you send a message to the dictionary using the key as an argument (**objectForKey:**). For example:

```
NSColor *aColor = [aDictionary objectForKey:
@"BackgroundColor"];
```

A Country object holds the name of a country as an instance variable; this country name also functions as the dictionary key. When you store a Country object in the dictionary, you also store the country name (in the form of an NSString) as the object's key. Later you retrieve the object by sending the dictionary the message **objectForKey:** with the country name as argument.

Storing Data Source Information. TAController also manages the data source for the table view on the interface. It stores the keys of the dictionary in an array object (NSArray), sorted alphabetically. When the table view requests data, the TAController "feeds" it the objects in the array.

Creation of Country Objects. Another important point of design is the manner in which the Country objects are created. Instead of Interface Builder creating them, the TAController object creates Country objects in response to users clicking the Add button.

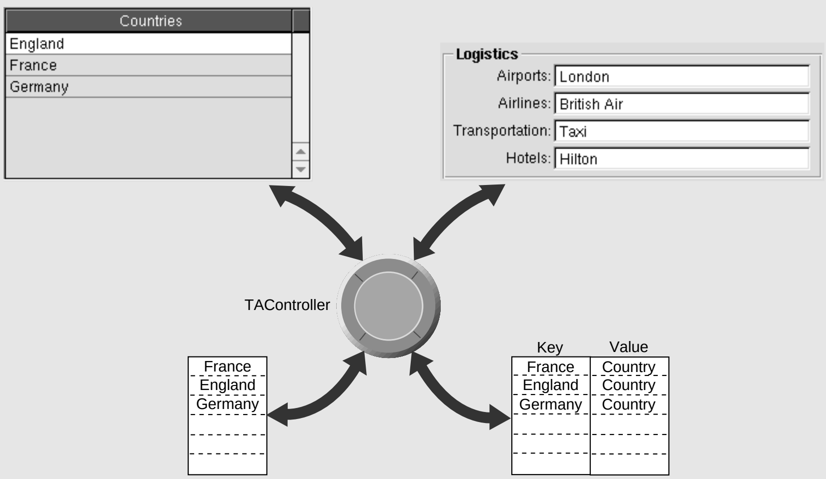
Delegation and Notification. An essential aspect of design not evident from the diagram are the roles *delegation* and *notification* play. The TAController object is the delegate of the application object and thereby receives messages that enable it to manage the application, which includes tracking the edited status of Country objects, initiating object archival upon application termination, and setting up the application at launch time.

How TAController Manages Data

The TAController class plays a central role in the Travel Advisor application. As the application's controller object, it transfers data from the model objects (Country instances) to the fields of the interface and, when users enter or modify data, back to the correct Country object. The TAController must also coordinate the data displayed in the table view with the current object, and it must do the right thing when users select an item in the table view or click the Add or Delete button. All custom code specific to the user interface resides in TAController.

The mechanics of this activity require an array (NSMutableArray) and a dictionary (NSMutableDictionary) for storing and accessing Country data. The diagram below illustrates the relationship among interface components, TAController, and the sources of data.

The dictionary contains Country objects (values) that are identified by the names of countries (keys). The dictionary is the source of data for the fields of Travel Advisor. The array derives from the dictionary and is sorted. It is the source of data for the table view.



Creating the Travel Advisor Interface

In creating the interface of Travel Advisor, you'll be exercising the capabilities of Interface Builder much more than you did with Currency Converter.

Getting Started

You should be familiar with many of the objects on the Travel Advisor interface because you've encountered them in the Currency Converter tutorial. The following illustration points out the objects that are new to you in this tutorial.

1 **Create the application project.**

Start Project Builder.

Choose New from the Project menu.

In the New Project panel, select the Application project type.

Name the application "TravelAdvisor" and click OK.

1 **Open the application's nib file.**

Click Interfaces in the project browser.

Select **TravelAdvisor.nib**, and double-click its icon.

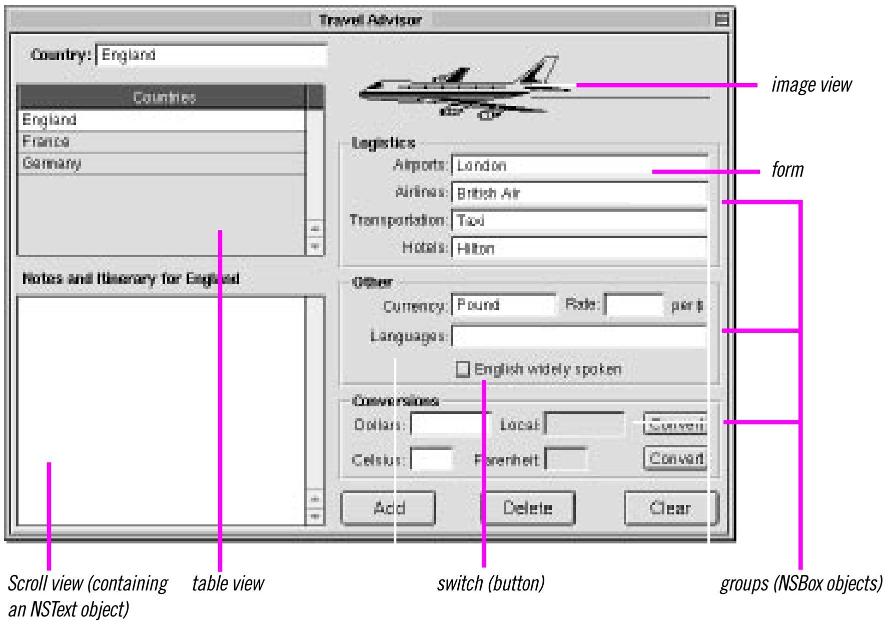
1 **Customize the application's window.**

In Interface Builder:

Resize the window, using the example at right as a guide.

In the Attributes display of the Inspector panel, entitle the window "Travel Advisor."

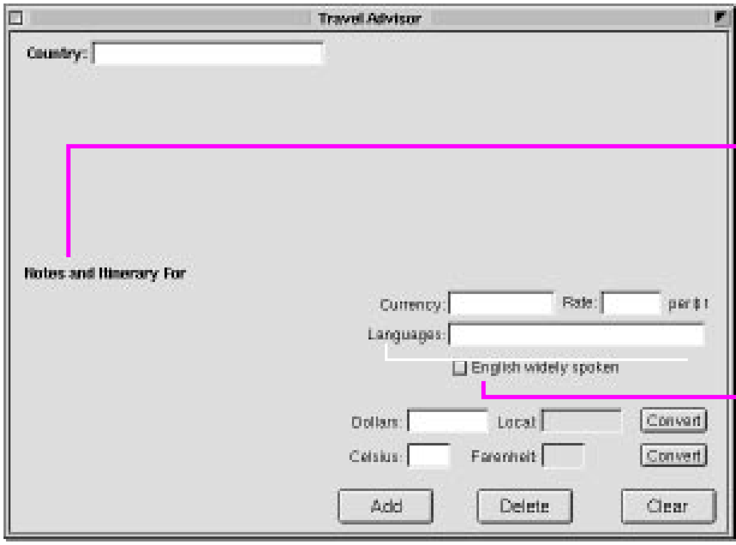
Turn off the resize bar.



The following pages describe the purpose of each new object found on Interface Builder's palettes and explain how to set these objects up for Travel Advisor. Before getting to these new objects, start with the familiar ones: buttons and text fields.

1 Put the text fields, labels, and buttons on the window.

Position, resize, and initialize the objects as shown.



Be sure this label contains enough "padding" for the longest country name.

Drag the switch object from the views palette and drop it here.

You might think the "English widely spoken" object is a new kind of object. It's actually a button, a special style of button called a switch.

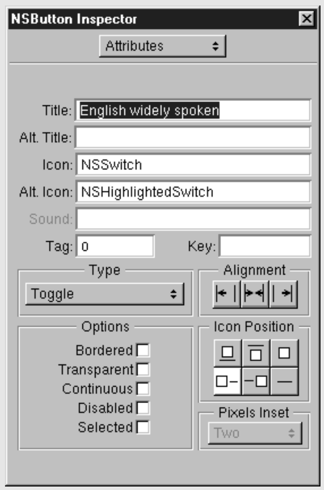
Set up the switch.

Double-click to select text, then type new label.

Varieties of Buttons

If in Interface Builder you select the "English widely spoken" switch and bring up the Attributes inspector, you can see that the switch is a button set up in a special way.

Buttons are two-state control objects. They are either off or on, and this state can be set by the user or programmatically (**setState:**). For certain types of buttons (especially standard buttons like Currency Converter's Convert button), when the state is switched, the button sends an action message to a target object. Toggle-type buttons—such as switches and radio buttons— visually reflect their state. Applications can learn of this state with the **state** message. You can make your own buttons, associating icons and titles with a button's off and on states, and positioning title and icon relative to each other.

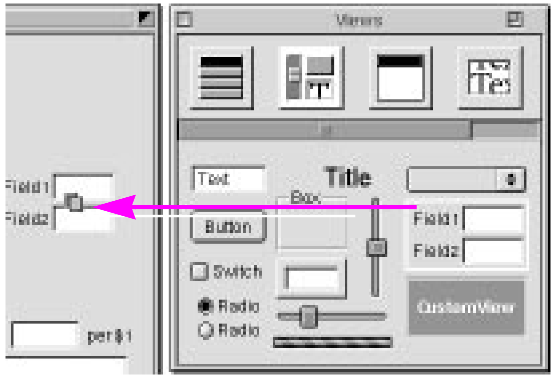


New Objects: Forms, Groups, and Scroll Views

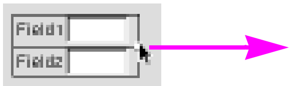
Construct the “Logistics” section of the interface using a form object.

1 **Place a form on the interface and prepare it.**

Drag the form object from the Views palette.

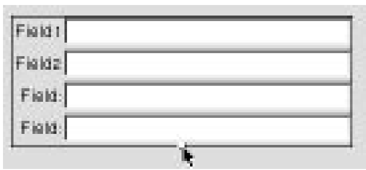


Increase the size of the form’s fields by dragging the middle resize handle sideways.



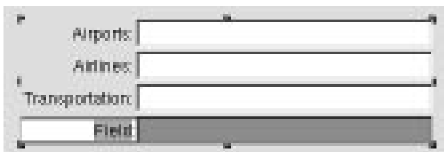
Drag to lengthen the fields.

Create two more form fields by Alternate-dragging the bottom-middle resize handle downward.



As you alternate-drag, new form fields appear underneath the cursor.

Rename the field labels.



Double-click to select label text.

Type the new label text and click outside the form to set the text.

To make titled sections of the fields, forms, and buttons on the Travel Advisor interface, group selected objects. By grouping them, you put them in a box.

1 Group the objects on the interface.

Select the two Convert buttons and the Dollars, Local, Celsius, Fahrenheit labels and text fields.

Choose Format ► Group ► Group in Box.

Double-click “Title” to select it.

Choose Format ► Font ► Bold to make the title bold face.

Rename “Title” to “Conversions.”

Repeat for the next two groups: “Logistics” and “Other.”

To select the objects as a group, drag a selection rectangle around them or Shift-click each object. (To make a selection rectangle, start dragging from an empty spot on the window.)

After you choose the Group in Box command, the objects are enclosed by a titled box.

Boxes are a useful way to organize and name sections of an interface. In Interface Builder you can move, copy, paste, and do other operations with the box as a unit. For Travel Advisor, you don’t need to change the default box attributes.

Before You Go On

The box, an instance of NSBox, is the *superview* of all of its grouped objects. (A *view*, simply put, is any object visible on a window.) A superview encloses its *subviews* and is the next in line to respond to user actions if its subviews cannot handle them.

The scroll view on the DataViews palette encloses a text object (an instance of NSText). This object allows users to enter, edit, and format text with minimal programmatic involvement on your part.

More About Forms

Forms are labelled fields bound vertically in a matrix. The fields are the same size and each label is to the left of its field. Forms are ideal objects for applications that display and capture multiple rows of data, as do many corporate client-server applications.

The editable fields in a form are actually cells that you programmatically identify through zero-based indexing; the first cell is at index 0 of the matrix, the second cell at index 1, and so on. NSForm defines the behavior of forms; individual cells are instances of NSFormCell. Access these cells with NSForm’s **cellAtIndex:** method.

Form Attributes

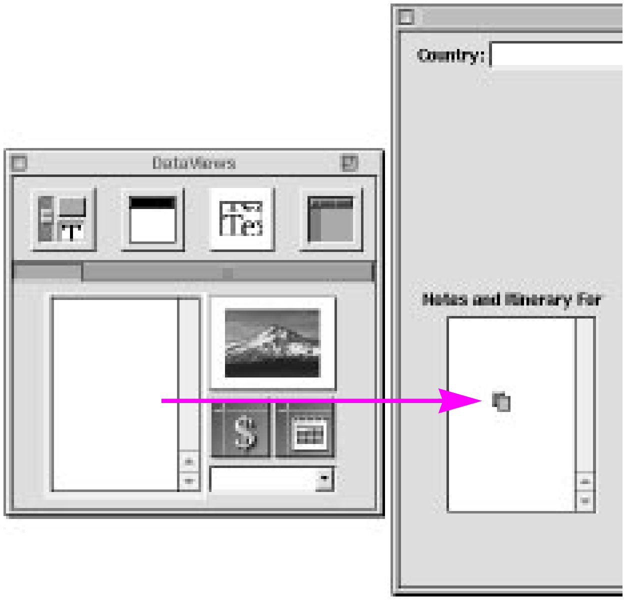
In addition to the obvious controls in the Forms inspector, there’s the “Cell tags = positions” attribute. Switching this on assigns tags to each NSFormCell that correspond to the cells’ indices. (A tag is a number assigned to an object that is used to identify and access that object. You’ll use tags extensively in the next tutorial.)

The Scrollable option, turned on by default, enables the user to type long entries in fields, scrolling contents to the left as characters are entered.

1 **Put the scroll view on the window and resize it.**

Drag the scroll view from the DataViews palette and drop it on the lower-left corner of the window.

Resize the scroll view.

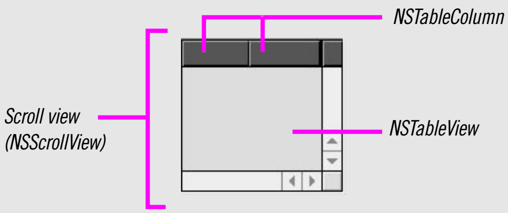


You don't need to change any of the default attributes of the scroll view (but you might want to look at the attributes that you can set, if you're curious).

More About Table Views

A table view is an object for displaying and editing tabular data. Often that data consists of a set of related records, with rows for individual records and columns for the common fields (attributes) of those records, Table views are ideal for applications that have a database component, such as Enterprise Objects Framework applications.

The table view on Interface Builder's TabulationViews palette is actually several objects, bound together in a scroll view. Inside the scroll view is an instance of NSTableView in which data is displayed and edited. At the top of the table view is an NSTableHeaderView object, which contains one or more column headers (instance of NSTableColumn).



Later in this tutorial you will learn some basic techniques for accessing and managing the data in a table view. Here's a quick preview of the essential pieces:

- **Data source**. The data source is any object in your application that supplies the NSTableView with data. The elements of data (usually records) must be identifiable through zero-based indexing. The data source must implement some or all of the methods of the NSTableDataSource informal protocol.
- **Column identifier**. Each column (NSTableColumn) of a table view has an identifier associated with it, which can be either an NSString or a number. You use the identifier as a key to obtain the value of a record field.
- **Delegate methods**. NSTableView sends several messages to its delegate, giving it the opportunity to control the appearance and accessibility of individual cells, and to validate or deny editing in fields.

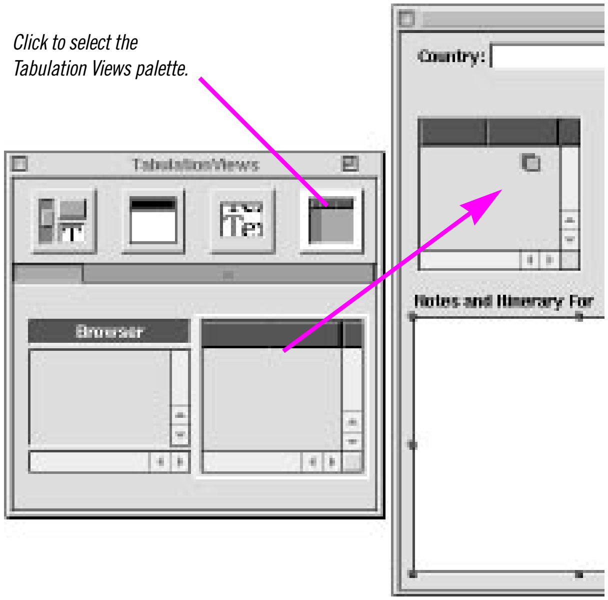
More New Objects: Table Views, Image Views, and Menus

Next, add a table view for displaying the list of countries.

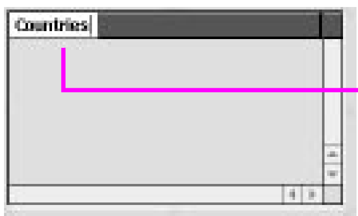
1 **Place and configure the table view.**

Drag the table view object from the TabulationViews palette.

Resize the table view.



Set the title of the first column to "Countries."



Double-click column twice (first to select the column, second to insert the cursor). Type "Countries", then click anywhere outside the column.

Make the table header only one column.

When this cursor appears over the line separating columns, drag the line so that it's flush with the right edge.

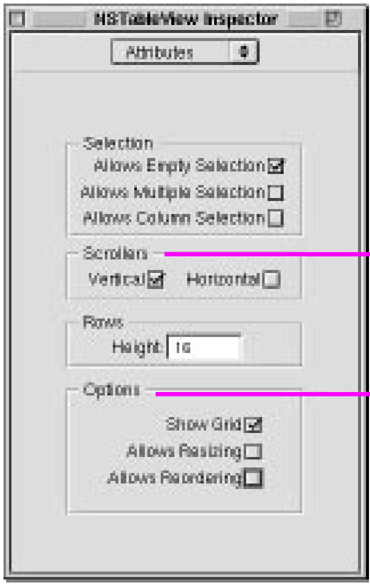
You can also delete the unneeded column by selecting it and pressing the Delete key.

The other object on the TabulationViews palette is a *browser*. It is just as suitable for the Travel Advisor application as a table view. Browsers are ideal for displaying hierarchically structured information (such as is found in typical file systems) as well as single-level views of data such as the list of countries in Travel Advisor. A table view can also handle single-column rows of data easily.

To configure the table view, you must set attributes of two component objects: the NSTableView object and the NSTableColumn object.

Select the NSTableView by double-clicking the interior of the table view.

Set the attributes as shown at right.



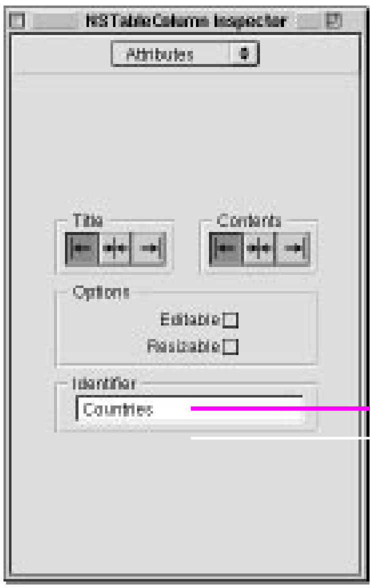
Since this is a single-column view and country names are of limited length, you need only the vertical scroller in case there's more countries than can be shown at once.

Whether to show the grid is a matter of personal preference, but turn off resizing and reordering. The user shouldn't be able to affect the contents of the column directly.

The Attributes display for NSTableView is the same as that for NSScrollView.

Click the left column to select it.

Set the NSTableColumn attributes as shown at right.



Type the name with which you want to identify the column programmatically. For Travel Advisor, make this the same as the column title.

The Travel Advisor window is nearly complete. For a decorative touch, you're next going to add an image to the interface.

1 **Add an image to the interface.**

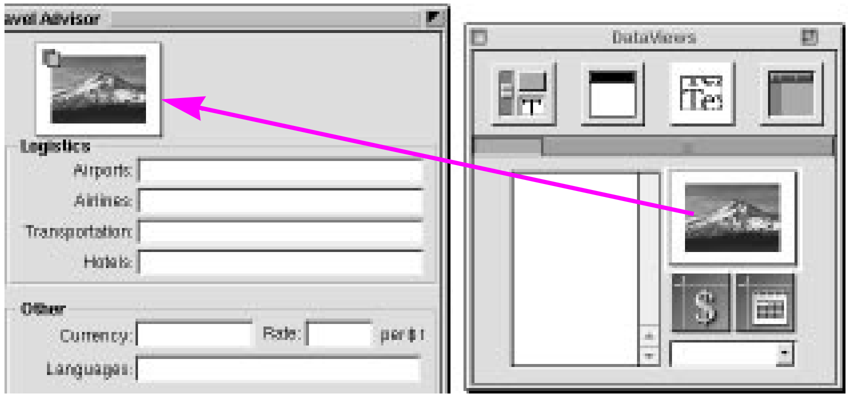
Select the DataViews palette (see example).

Drag the image view onto the window.

In Project Builder:

Double-click Images in the project browser.

In the Open panel, select the file **Airline.eps** in **/SystemDeveloper/Examples/AppKit/TravelAdvisor**.



Before You Go On

Sometimes buttons are the preferred objects for holding images—for instance when you want a different image for either state of a button. But when buttons are disabled, any image they display is dimmed. So for decorative images, use image views (NSImageView) instead of buttons.

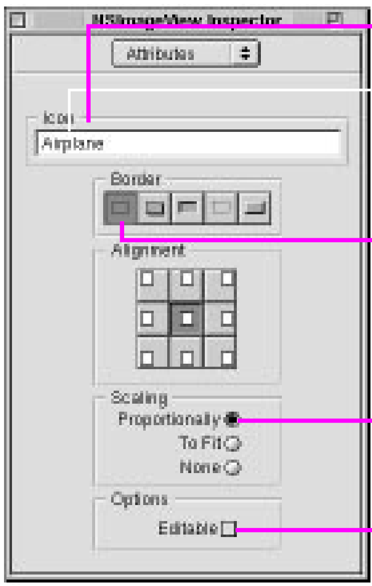
When you drop an image over a button or image view, Interface Builder adds it to the both the nib file and the project (upon your approval). You can add the image only to the nib file by dropping the image over the nib file window. Resources in a nib file are accessible only when the nib file has been loaded; an application's project-wide resources are always accessible.

In Interface Builder:

In the Attributes inspector for the image view, type the name of the image and set the NSImageView attributes.

Make the image view (and the enclosed image) small enough to fit between the menu bar and the Logistics group.

Add a "velocity" line behind the airplane. (Tip: Make an untitled black box with a vertical offset of zero and run the top and bottom lines together.)



Enter the name of the image file, minus the extension. The image can be in any acceptable format, and must be a part of the project.

You can also insert an image in an image view and add it to the project by dragging it from the File Viewer and dropping it over the image view.

The border of the image should not be visible.

Since the image is larger than the image view, have it scale proportionally.

Uncheck if you don't want users to affect the image in any way.

Travel Advisor's menu contains default submenus and commands. You need a submenu and menu commands that are not included in the default set and that are not found on the Menus palette. Use the Submenu and the Item cells to create customized menus and menu items, respectively.

1 Add a menu and menu items to the menu bar.

Select the Menus palette.

Drag the generic Submenu item and drop it between the Edit and Window submenus.

Double-click Submenu to select the menu title; change the name to "Records".

Click the new Records menu to expose the Item command.

Add three Items to the Records submenu (making four altogether) by dragging them from the Menus palette.

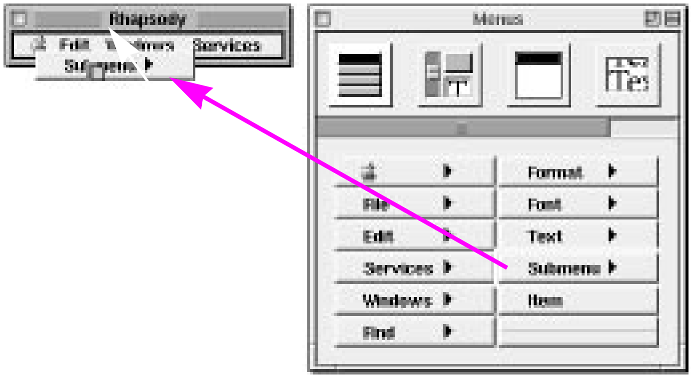
Change the command names to those shown at right.

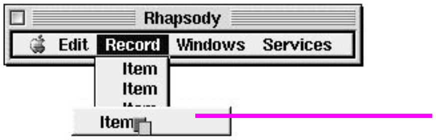
Add Command-key equivalents to the right of the Next Record and Prior Record commands.

Drag an Item cell and drop it between the Windows and Services submenus.

Change "Item" to "Print Notes...".

Remove unnecessary menu items from the File menu.





To insert a menu item, drag it from the Menus palette and drop it between or after the menu items currently on the menu.

To add a Command-key equivalent, double-click the area on the right side of a menu item and then press the key you want assigned.

Put the print command here for now.

To delete a menu item, select it and choose Delete from the Edit menu or press Command-x.

You don't need to add any menu items to the Services submenu. Applications can offer their services to other applications, based on the operations they can perform on types of selected data. As part of advertising their services, these applications specify the menu items to be used to access those services. At run time, these submenus and commands appear in the Services menu. For more on services, see "Services" in the on-line Programming Topics.

Finishing Touches: Formatters, Printing, and the Application Icon

One way to make your application's user interface more attractive is to format the contents of fields that display currencies and other numeric data. Fields can have fixed decimal digits, can limit numbers to specific ranges, can have currency symbols, and can show negative values in a special color. Interface Builder provides two formatter objects on its standard palettes, one for formatting dates and the other for formatting numbers. You'll use the second of these.

1 **Apply formatters to the rate and currency fields.**

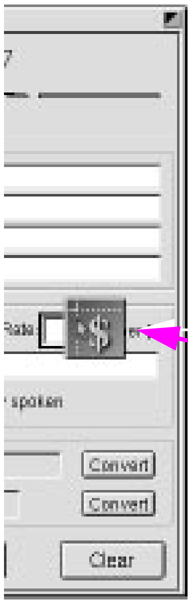
Select the DataViews palette in the Palette window.

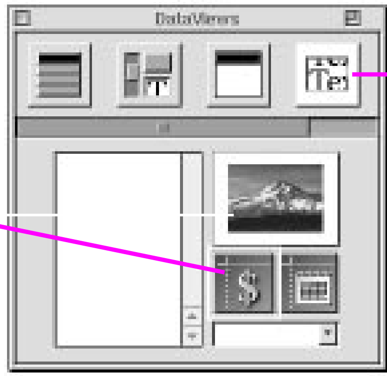
Drag a number-formatter object and drop it over the Rate field.

In the Formatter display of the inspector, specify a rate format by selecting the table-view row with the "99.99" format.

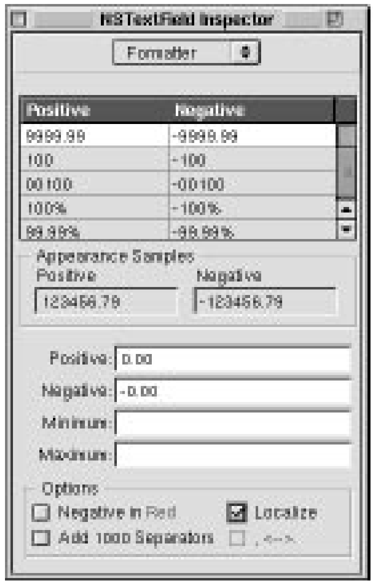
Type a zero in the field to initialize it.

Repeat for the Dollar and Local fields, but apply a suitable format.





Click to select DataViews palette.



When a text field (or other control) has a formatter applied to it, Interface Builder's inspector includes a Formatter display when that field is selected.

Click a predefined format in the table view to apply it to the field, or specify a custom format in the Positive and Negative fields.

For the Dollar and Local fields, specify the first predefined format ($9,999.99).

Formatters are objects that "translate" the values of certain objects to specific on-screen representations; formatters also convert a formatted string on a user interface into the represented object.

You can create, set, and modify formatter objects programmatically as well as by using Interface Builder. And you can create your own special formatter objects (such as ones, for example, that format phone numbers) and "palettize" them. For more on formatters, see "Behind 'Click Here': Controls, Cells, and Formatters" on page 107.

You can now connect many of the objects on the Travel Advisor interface through outlets and actions defined by the Application Kit. As you might recall, windows have an **initialFirstResponder** outlet for the object in the window that should be the initial focus of events. Text fields have a **nextKeyView** outlet that you connect so that users can tab from field to field. Forms also have a **nextKeyView** outlet for tabbing. (The fields within a form are already interconnected, so you don't need to connect them.)

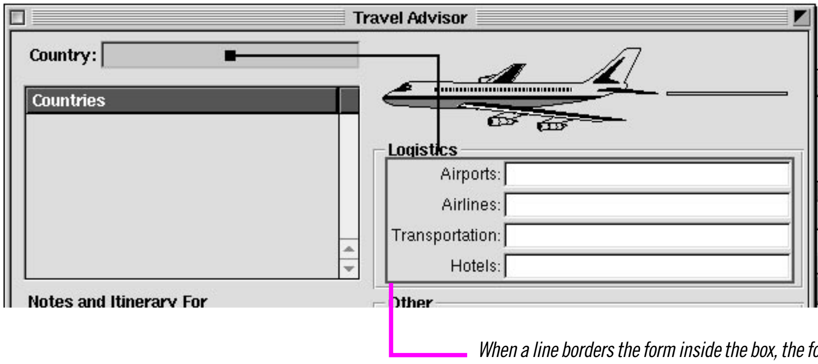
1 **Connect Application Kit outlets for inter-field tabbing and printing.**

Make a connection from the window icon in the nib file window to the Country field.

Select **initialFirstResponder** in the Connections display of the inspector and click Connect.

In top-to-bottom sequence, connect the fields and the form through their **nextKeyView** outlets.

When you reach the Languages field, connect it with the Country field, making a loop.



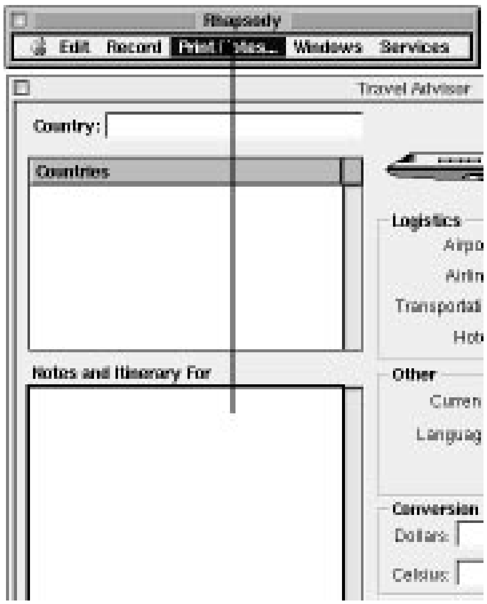
When a line borders the form inside the box, the form is selected. Release the mouse button and set the **nextKeyView** *outlet connection in the Connections inspector.*

The Application Kit also has "preset" actions that you can connect your application to. The NSText object in the scroll view can print its contents as can all objects that inherit from NSView. To take advantage of this capability, "hook up" the menu command with the NSText action method for printing.

Connect the Print Notes menu command to the text object in the scroll view.

Select the **print:** action method in the Connections display of the Inspector panel.

Click the Connect button in the Inspector's Connection display.



Make sure the text object (the white rectangle) is selected and not the scroll view that encloses it.

The final step in crafting the Travel Advisor interface has nothing to do with the main window, but with what users see of your application when they encounter it in the File Viewer: the application's icon.

1 Add the application icon.

In Project Builder:

Open the Project Inspector.

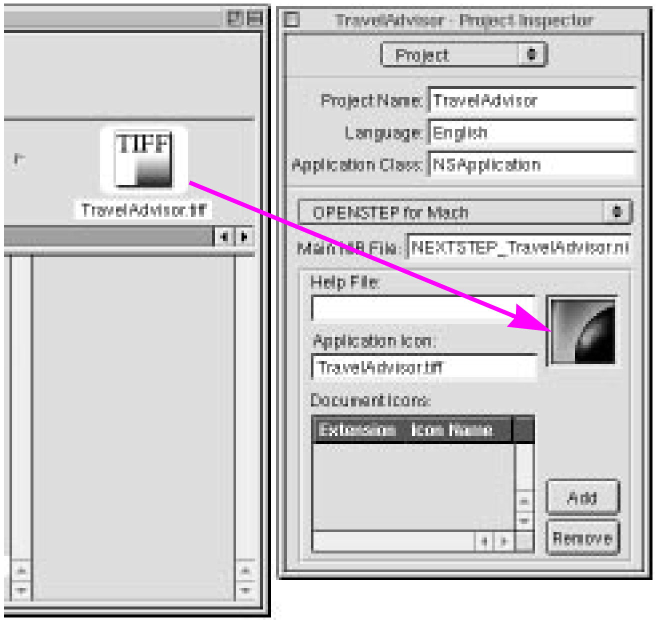
Go to the Project Attributes display of the inspector.

Click in the Application Icon field.

In File Viewer:

Locate **TravelAdvisor.eps** in **/System/Developer/Examples/AppKit/TravelAdvisor**.

Drag the image file into the icon well in the Project Attributes display.



1 Test the interface.

You're finished with the Travel Advisor interface. Test it by choosing Test Interface from Interface Builder's File menu.Try the following:

- Press the Tab key repeatedly. Notice how the cursor jumps between the fields of the form, and how it loops from the Languages field to the Country field. Press Shift-Tab to make the cursor go in the reverse direction.
- Enter some text in the scroll view, then click the Print Notes menu item. The Print dialog box is displayed. Print the text object's contents.
- Also in the scroll view, press the Return key repeatedly until a scroll box appears in the scroll bar.

Defining the Classes of Travel Advisor

Travel Advisor has three classes: Country, Converter, and TAController. Only TAController has outlets and actions. And, rather than defining the Converter class, you are simply going to add it to the project from the CurrencyConverter project and reuse it.

1 **Specify the Country and TAController classes.**

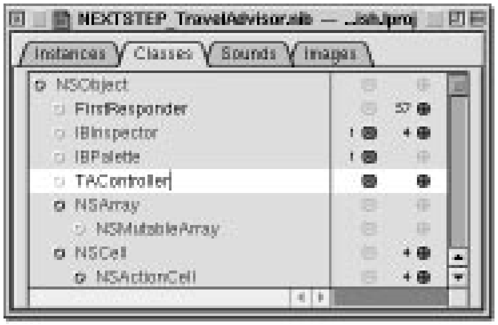
In Interface Builder, bring up the Classes display of the nib file window.

Select NSObject as the superclass.

Choose Subclass from the Classes menu.

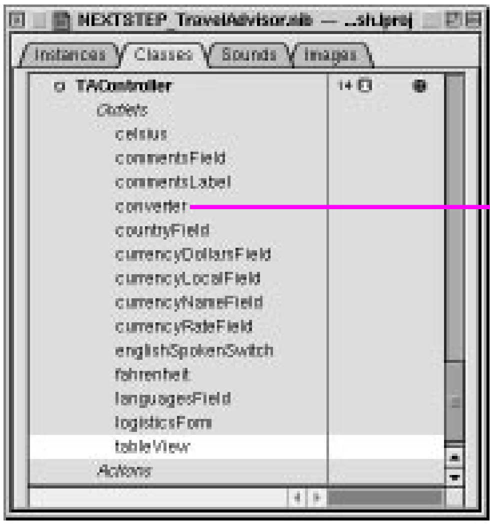
Type "Country" in place of "MyNSObject."

Repeat for class TAController.



1 **Specify TAController's outlets and actions.**

Add the outlets shown in the nib file window at right.



Through this outlet the TAController object establishes a connection with the instance of the Converter class. You will reuse this class later in this section.

Define the action methods shown in the nib file window at right.

In OpenStep there are many ways to reuse objects. For example, subclassing an existing class to obtain slightly different behavior is one way to reuse the functionality of the superclass. Another way is to integrate an existing class—like the Converter class—into your project.

1 **Reuse the Converter class.**

In Project Builder:

Double-click Classes in the project browser.

In the Add Classes panel, navigate to the CurrencyConverter project directory in **/System/Developer/Examples/AppKit.**

Select **Converter.m** and click OK.

When asked if you want to include the header file, click OK.

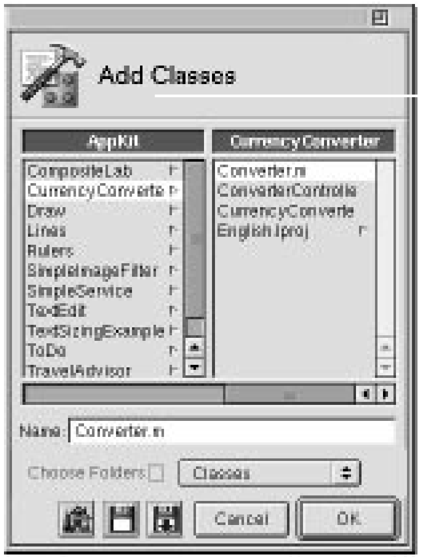
In Interface Builder:

Select the superclass of Converter (NSObject) in the Classes display of the TravelAdvisor nib file window.

Choose Classes ► Read File.

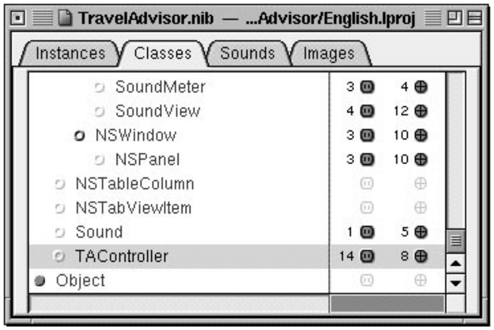
In the Open panel, select **Converter.h** in the TravelAdvisor project directory.

Click OK.



When you're finished with this procedure, the Converter class is copied both to the TravelAdvisor project and to the TravelAdvisor main nib file.

1 **Generate instances of the TAController and Converter classes.**

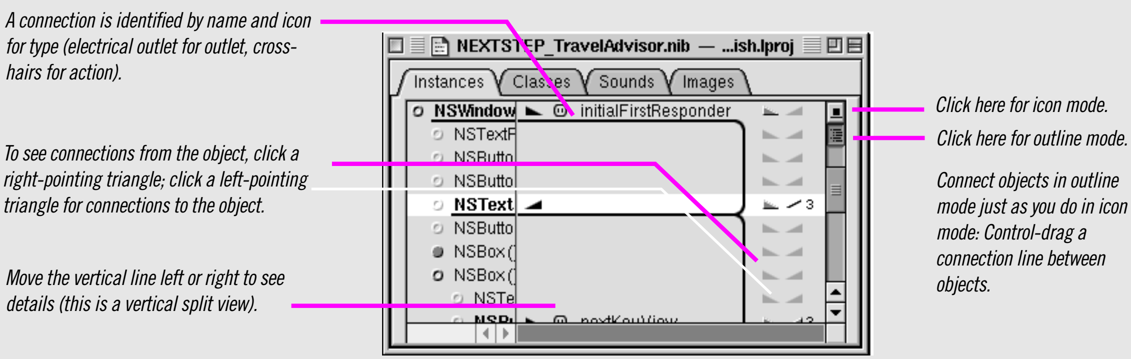


You don't need to instantiate the Country class in the nib file because it is not involved in any outlet or action connections. However, you must create an instance of TAController for making connections. TAController interacts behind the scenes with users as they manipulate the application's interface and mediates the data coming from and going to Country objects. It therefore needs access to interface objects and should be made the target of action messages.

Checking and Making Connections in Outline Mode

The nib file window of Interface Builder gives you two modes in which to view the objects in a nib file and to make connections between those objects. So far you've been working in the *icon mode* of the Instances display, which pictorially represents objects such as windows and custom objects.

Outline mode, as the phrase suggests, represents objects in a hierarchical list: an outline. The advantages of outline mode are that it represents all objects and graphically indicates the connections between them. You can connect objects through their outlets and actions in outline mode, as well as disconnect them by Control-clicking a connection line.



1 **Connect the TAController instance to its outlets and actions.**

Connect TAController to the outlets listed in this table.

Outlet	Make Connection To
celsius	Text field labeled “Celsius”
commentsLabel	Label that reads “Notes and Itinerary for”
commentsField	Text object within scroll view
converter	Instance of Converter class (cube in Instances display)
countryField	Text field labeled “Country”
currencyDollarsField	Text field labeled “Dollars”
currencyLocalField	Text field labeled “Local”
currencyNameField	Text field labeled “Currency”
currencyRateField	Text field labeled “Rate”
englishSpokenSwitch	Switch (button) labeled “English widely spoken”
fahrenheit	Text field labeled “Fahrenheit”
languagesField	Text field labeled “Languages”
logisticsForm	Form in group (box) labeled “Logistics”; the form is selected when a gray line borders it.
tableView	The area underneath the “Countries” column

File’s Owner

Every nib file has one owner, represented by the File’s Owner icon in a nib file window. The owner is an object, external to the nib file, that relays messages between the objects unarchived from the nib file and the other objects in your application.

You specify a file’s owner programmatically, in the second argument of NSBundle’s **loadNibNamed:owner:.** The File’s Owner icon in Interface Builder is a “proxy” object for that owner. Although you can assign owners to this object in Interface Builder, this doesn’t necessarily guarantee anything about the file’s real owner.

In the main nib file File’s Owner always represents NSApp, the global NSApplication constant. The main nib file is automatically created when you create an application project; it is loaded in **main()** when an application is launched.

Nib files other than the main nib file— *auxiliary nib files*—contain objects and resources that an application may load only when it needs them (for example, an Info panel). You must specify the owner of auxiliary nib files.

You can determine or set the class of the current nib file’s owner in Interface Builder by selecting the File’s Owner icon in the nib file window and then displaying the Custom Class inspector view. You’ll get to practice this technique when you learn how to create multi-document applications in the next tutorial.

Connect the TAController instance to control objects in the interface via its actions.

Action	Make Connection From
addRecord:	"Add" button
blankFields:	"Clear" button
convertCelsius:	"Convert" button to the right of the "Fahrenheit" field
convertCurrency:	"Convert" button to the right of the "Local" field
deleteRecord:	"Delete" button
handleTVClick:	The table view (the area beneath the "Countries" column header)
nextRecord:	The "Next Record" menu command on the Records submenu
prevRecord:	The "Prior Record" menu command on the Records submenu
switchChecked:	The "English widely spoken" switch

Before You Go On

You're next going to connect objects through an outlet defined by several OpenStep classes. This outlet, named **delegate**, is assigned the **id** value of a custom object. As the delegate of NSApp (the NSApplication object), TAController will receive messages from it as certain events happen.

Every application has a global NSApplication object (called NSApp) that coordinates events specific to the application. Among many other messages, NSApp sends a message to its delegate notifying it that the application is about to terminate. Later, you will implement TAController so that, when it receives this message, it archives (saves) the dictionary containing the Country objects.

Compiled and Dynamic Palettes

A palette is an area on Interface Builder's Palettes window that holds one or more reusable objects. You can add these objects to your application's interface using the drag-and-drop technique. There are two types of palettes: dynamic and compiled (also called "static palettes"). To the user they seem identical, but the differences are many.

Static palettes are built as a project and have code defining their objects; dynamic palettes include no special code—they're unique configurations of objects found on static palettes. Consequently, static palettes must be compiled, but you can create dynamic palettes on the fly, without writing and compiling code. Objects on static palettes can have their own inspectors and editors, which dynamic-palette objects cannot have.

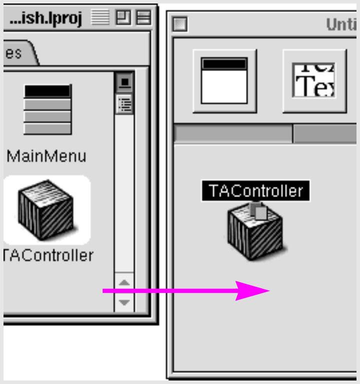
You usually create a static palette as a way to distribute your objects—and the logic informing these objects' behavior—to potential users. Many developers of commercial OpenStep objects make use of static palettes as a distribution medium. Creating static palettes (and their inspectors and editors) is a more complex process than creating dynamic palettes, but the resulting product has more value added to it.

Using Dynamic Palettes

Dynamic palettes are a great convenience. You can save groups of objects, with or without their interconnections, to a dynamic palette at any time. You can save dynamic palettes and store them in the file system, just as you do with the traditional compiled palette. You can remove the palette from the Palette viewer and, when you need it again, load it back into Interface Builder.

To store objects on a dynamic palette:

- Choose Tools ► Palettes ► New to create a blank palette.
- Select objects singly or in groups on the interface or in the nib file window (either icon or outline mode).
- Alternate-drag these objects and drop them on the blank palette.



Alternate-drag objects to move them onto palettes, to move them around palettes, and to take them off of palettes.

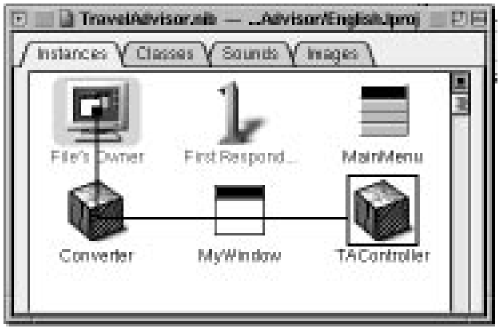
You can use dynamic palettes to:

- Store collections of often-used View objects configured with specific sizes and other attributes. For instance, you could have a "standard" text field of a certain length, font, and background color stored on a dynamic palette.
- Hold windows and panels that are replicated in your projects (such as Info panels).
- Store versions of interfaces.
- Keep interconnected objects as a template that you can later use as-is or modify for particular circumstances. For instance, you could store a group of text fields and their delegate, or a set of controls and their connections to a controller object.
- Assist in prototyping and group work. For example, you could mail a palette file containing an interface to interested parties.

1 Connect the delegate outlet.

Drag a connection line from File's Owner to the TAController object.

In the Connections display of the Inspector panel select delegate and click OK.



Notice that the direction of the connection is from the File's Owner (which is the application object) to the TAController object.

1 Generate source code files for the TAController and Country classes.

Save **TravelAdvisor.nib**.

Select the class in the Classes display of the nib file window.

Choose Classes ► Create Files.

Respond Yes to the confirmation messages.

When you generate the header and implementation files for all classes of Currency Converter, you are finished with the Interface Builder portion of development. Be sure you save the nib file before you switch over to Project Builder.

You can assign delegates programmatically or by using Interface Builder. For more information, see "Getting in on the Action: Delegation and Notification" on page 100.

Implementing the Country Class

Although it has no outlets, the Country class defines a number of instance variables that correspond to the fields of Travel Advisor.

1 **Declare instance variables.**

In Project Builder, click Headers in the project browser, then select **Country.h**.

Add the declarations shown between the braces at right.

```
@interface Country : NSObject <NSCoding>   A
{
    NSString *name;                        B
    NSString *airports;
    NSString *airlines;
    NSString *transportation;
    NSString *hotels;
    NSString *languages;
    BOOL     englishSpoken;
    NSString *currencyName;
    float    currencyRate;                 C
    NSString *comments;
}
```

A Declares that the Country class adopts the NSCoding protocol.

B Explicitly types the instance variable as "a pointer to class NSString"—or an NSString object. See below for more about the NSString class.

C Declare non-object instance variables the same way you declare them in C programs. In this case, **currencyRate** is of type **float**.

NSString: A String for All Countries

NSString objects represent character strings. They're behind almost all text in an application, from labels to spreadsheet entries to word-processing documents. NSStrings (or *string objects*) supplant that familiar C programming data type, **char** *.

"But why?" you might be saying. "Why not stick with the tried and true?" By representing strings as objects, you confer on them all the advantages that belong to objects, such as persistency and the capability for distribution. Moreover, thanks to data encapsulation, string objects can use whatever encoding is needed and can choose the most efficient storage for themselves.

The most important rationale for string objects is the role they play in *internationalization*. String objects contain Unicode characters rather than the narrow range of characters afforded by the ASCII character set. Hence they can represent words in Chinese, Arabic, and many other languages.

The NSString and NSMutableString classes provide API to create static and dynamic strings, respectively, and to perform string operations such as substring searching, string comparison, and concatenation.

None of this prevents you from using **char** * strings, and there are occasions where for performance or other reasons you should. However, the public interfaces of OpenStep classes now use string objects almost exclusively. A number of NSString methods enable you to convert string objects to **char** * strings and back again.

The Foundation Framework: Capabilities, Concepts, and Paradigms

The Foundation framework consists of a base layer of classes that specify fundamental object behavior plus a number of utility classes. It also introduces several paradigms that define functionality not covered by the Objective-C language. Notably, the Foundation framework:

- Makes software development easier by introducing consistent conventions for things such as object deallocation
- Supports Unicode strings, object persistence, and object distribution
- Provides a level of operating-system independence, enhancing application portability

Root Class

NSObject, the principal root class, provides the fundamental behavior and interface for objects. It includes methods for creating, initializing, deallocating, copying, comparing, and querying objects (introspection). Almost all OpenStep objects inherit ultimately from NSObject.

Deallocation of Objects

The Foundation framework introduces a mechanism for ensuring that objects are properly deallocated when they're no longer needed. This mechanism, which depends on general conformance to a policy of object ownership, automatically tracks objects that are marked for release within a loop and deallocates them at the close of the loop. See "Object Ownership, Retention, and Disposal" on page 88 for more information.

Data Storage and Access

The Foundation framework provides object-oriented storage for

- Arrays of raw bytes (NSData) and characters (NSString)
- Simple C data values (NSValue and NSNumber)
- Objective-C objects of any class (NSArray, NSDictionary, NSSet, and NSPPL)

NSArray, NSDictionary, and NSSet (and related mutable classes) are *collection classes* that also allow you to organize and access objects in certain ways (see "The Collection Classes" on page 86).

Text and Internationalization

NSString internally represents text in various encodings, most importantly Unicode, making applications inherently capable of expressing a variety of written languages. NSString also provides methods for searching, combining, and comparing strings. NSCharacterSet represents various groupings of characters which are used by NSString. An NSScanner object scans numbers and words from an NSString object. For more information, see "NSString: A String for All Countries" on page 83.

You use NSBundle objects to load code and localized resources dynamically (see "Only When Needed: Dynamically Loading Resources and Code" on page 126). The NSUserDefaults class enables you to store and access default values based on locale as well as user preferences.

Object Persistence and Distribution

NSSerializer makes it possible to represent the data that an object contains in an architecture-dependent way. NSCoder and its subclasses take this process a step further by storing class information along with the data, thereby enabling archiving and distribution. Archiving (NSArchiver) stores encoded objects and other data in files. Distribution denotes the transmission of encoded object data between different processes and threads (NSPortCoder, NSConnection, NSDistantObject, and others).

Other Functionality

Date and time. The NSDate, NSCalendarDate, and NSTimeZone classes generate objects that represent dates and times. They offer methods for calculating temporal differences, for displaying dates and times in any desired format, and for adjusting times and dates based on location in the world.

Application coordination. NSNotification, NSNotificationCenter, and NSNotificationQueue implement a system for broadcasting notifications of changes within an application. Any object can specify and post a notification, and any other object can register itself as an observer of that notification. You can use an NSTimer object to send a message to another object at specific intervals.

Operating system services. Many Foundation classes help to insulate your code from the peculiarities of disparate operating systems.

- NSFileManager provides a consistent interface for file-system operations such as creating files and directories, enumerating directory contents, and moving, copying, and deleting files.
- NSThread lets you create multi-threaded applications.
- NSProcessInfo enables you to learn about the environment in which an application runs.
- NSUserDefaults allows applications to query, update, and manipulate a user's default settings across several domains: globally, per application, and per language.

Country.h also declares a dozen or more methods. Most of these are *accessor methods*. Accessor methods fetch and set the values of instance variables. They are a critical part of an object's interface.

1 **Declare methods.**

After the instance variables, add the declarations listed here.

```
/* initializtion and de-allocation */
- (id)init;                                    A
- (void)dealloc;
/* archiving and unarchiving */
- (void)encodeWithCoder:(NSCoder *)coder;      B
- (id)initWithCoder:(NSCoder *)coder;
/* accessor methods */
- (NSString *)name;
- (void)setName:(NSString *)str;
- (NSString *)airports;
- (void)setAirports:(NSString *)str;
- (NSString *)airlines;
- (void)setAirlines:(NSString *)str;
- (NSString *)transportation;
- (void)setTransportation:(NSString *)str;
- (NSString *)hotels;
- (void)setHotels:(NSString *)str;
- (NSString *)languages;
- (void)setLanguages:(NSString *)str;
- (BOOL)englishSpoken;
- (void)setEnglishSpoken:(BOOL)flag;
- (NSString *)currencyName;
- (void)setCurrencyName:(NSString *)str;
- (float)currencyRate;
- (void)setCurrencyRate:(float)val;
- (NSString *)comments;
- (void)setComments:(NSString *)str;
```

A **Object initialization and deallocation.** In OpenStep you usually create an object by allocating it (**alloc**) and then initializing it (**init** or **init**... variant):

```
Country *aCountry = [[Country alloc] init];
```

When Country's **init** method is invoked, it initializes its instance variables to known values and completes other start-up tasks. Similarly, when an object is deallocated, its **dealloc** method is invoked, giving it the opportunity to release objects it's created, free **malloc**'d memory, and so on.

B **Object archiving and unarchiving.** The **encodeWithCoder:** declaration indicates that objects of this class are to be archived. Archiving encodes an object's class and state (typically instance variables) and stores it in a file. Unarchiving, through **initWithCoder:**, reads the encoded class and state data from the file and restores the object to its previous state. There's more on this topic in the following pages.

C **Accessor methods.** The declaration for accessor methods that *return* values is, by convention, the name of the instance variable preceded by the type of the returned value in parentheses. Accessor methods that *set* the value of instance variables begin with "set" prepended to the name of the instance variable (initial letter capitalized). The "set" method's argument takes the type of the instance variable and the method itself returns void.

When a class adopts a protocol, it asserts that it implements the methods the protocol declares. Classes that archive or serialize their data must adopt the NSCoding protocol. See "Objective-C Extensions" in the on-line Programming Languages for more on protocols.

Before You Go On

If you don't want to allow an instance variable's value to be changed by any object other than one of your class, *don't* provide a set method for the instance variable. If you do provide a set method, make sure objects of your own class use it when specifying a value for the instance variables. This has important implications for subclasses of your class.

Exercise: The previous example shows the declarations for only a few accessor methods. Every instance variable of the Country class should have an accessor method that returns a value and one that sets a value. Complete the remaining declarations.

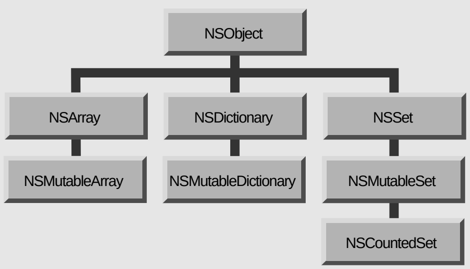
The Collection Classes

Several classes in OpenStep's Foundation framework create objects whose purpose is to hold other objects. These collection classes are very useful. Instances of them can store and locate their contents through a number of mechanisms.

- Arrays (NSArray) store and retrieve objects in an ordered fashion through zero-based indexing.
- Dictionaries (NSDictionary) store and quickly retrieve objects using key/value pairs. For example, the key "red" might be associated with an NSColor object representing red.
- Sets (NSSet) are unordered collections of distinct elements. Counted sets (NSCountedSet) are sets that can contain duplicate (non-distinct) elements; these duplicates are tracked through a counter. Use sets when the speed of membership-testing is important.

The mutable versions of these classes allow you to add and remove objects programmatically after the collection object is created.

Collection objects also provide a valuable way to store data. When you store (or *archive*) a collection object in the file system, its constituent objects are also stored.



Now that you've declared the Country class's accessor methods, implement them.

1 **Implement the accessor methods.**

Select **Country.m** in the project browser.

Write the code that obtains and sets the values of instance variables.

```
- (NSString *)name                                    A
{
    return name;
}

- (void)setName:(NSString *)str                       B
{
    [name autorelease];
    name = [str copy];
}
```

A For "get" accessor methods (at least when the instance variables, like Travel Advisor's, hold immutable objects) simply return the instance variable.

B For accessor methods that set *object* values, first send **autorelease** to the current instance variable, then **copy** (or **retain**) the passed-in value to the variable. The **autorelease** message causes the previously assigned object to be released at the end of the current event loop, keeping current references to the object valid until then.

If the instance variable has a non-object value (such as an integer or float value), you don't need to **autorelease** and **copy**; just assign the new value.

In many situations you can send **retain** instead of **copy** to keep an object around. But for "value" type objects, such as NSStrings and our Country objects, **copy** is better. For the reason why, and for more on **autorelease**, **retain**, **copy**, and related messages for object disposal and object retention, see "Object Ownership, Retention, and Disposal" on page 88.

Before You Go On

Exercise: The example above shows the implementation of the accessor methods for the **name** instance variable. Implement the remaining accessor methods.

Object Ownership, Retention, and Disposal

The problem of object ownership and disposal is a natural concern in object-oriented programming. When an object is created and passed around various "consumer" objects in an application, which object is responsible for disposing of it? And when? If the object is not deallocated when it is no longer needed, memory "leaks." If the object is deallocated too soon, problems may occur in other objects that assume its existence, and the application may crash.

The Foundation framework introduces a mechanism and a policy that helps to ensure that objects are deallocated when—and only when—they are no longer needed.

Who Owns Which Object?

The policy is quite simple: You are responsible for disposing of all objects that you own. You own objects that you create, either by allocating or copying them. You also own (or share ownership in) objects that you retain, since retaining an object increments its reference count (see facing page). The flip side of this rule is: If you don't own an object, you need not worry about releasing it.

But now another question arises. If the owner of an object *must* release the object within its programmatic scope, how can it give that object to other objects? The short answer is: the **autorelease** method, which marks the receiver for later release, enabling it to live beyond the scope of the owning object so that other objects can use it.

The **autorelease** method must be understood in a larger context of the *autorelease mechanism* for object deallocation. Through this programmatic mechanism, you implement the policy of object ownership

How Autorelease Pools Work: An Example

A. **myObj** creates an object:

```
anObj = [[MyClass alloc] init];
```

B. **myObj** returns the object to **yourObj**, autoreleased:

```
return [anObj autorelease];
```

The object is "put" in the autorelease pool; that is, the autorelease pool starts tracking the object.

C. **yourObj** retains the object:

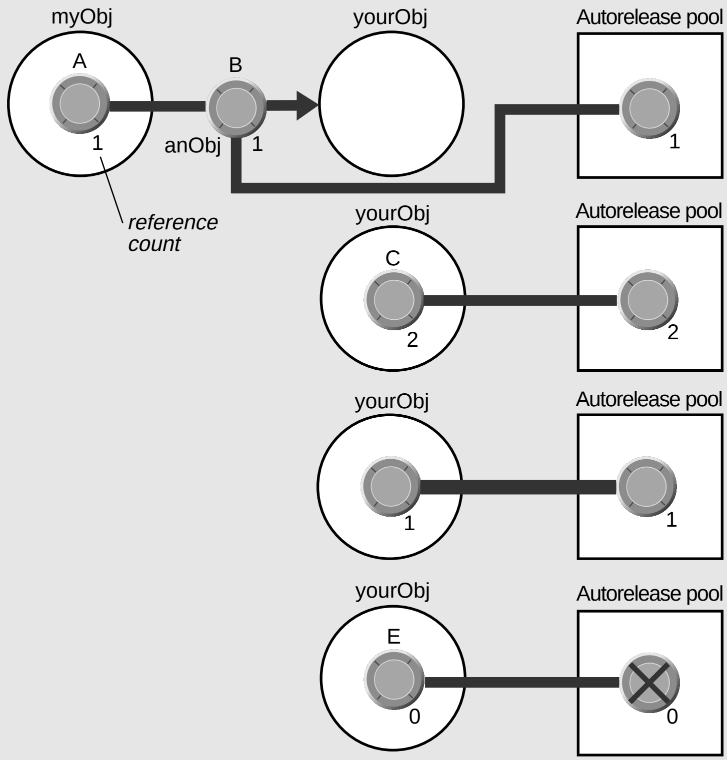
```
[anObj retain];
```

The retain message increments the reference count.(If the object wasn't retained it would be deallocated at the end of the current event cycle.)

D. At the end of the event cycle, the autorelease pool sends **release** to all of its objects, thereby decrementing their reference counts. Objects with reference counts of zero are deallocated. Since **anObj** now has a reference count of one, it is not deallocated.

E. **yourObj** sends **autorelease** to **anObj**, putting it into an autorelease pool again. At the end of the event cycle, the autorelease pool sends **release** to its objects; since **anObj**'s reference count is now zero, it's deallocated.

For a fuller description of object ownership and disposal, see the introduction to the Foundation framework reference documentation.



and disposal.

Reference Counts, Autorelease Pools, and Deallocation

Each object in the Foundation framework has an associated reference count. When you allocate or copy an object, its reference count is set at 1. You send **release** to an object to decrement its reference count. When the reference count reaches zero, NSObject invokes the object's **dealloc** method, after which the object is destroyed. However, successive consumers of the object can delay its destruction by sending it **retain**, which increments the reference count. You retain objects to ensure that they won't be deallocated until you're done with them.

Each application puts in place at least one *autorelease pool* (for the event cycle) and can have many more. An autorelease pool tracks objects marked for eventual release and releases them at the appropriate time. You put an object in the pool by sending the object an **autorelease** message. In the case of an application's event cycle, when code finishes executing and control returns to the application object (typically at the end of the cycle), the application object sends **release** to the autorelease pool, and the pool releases each object it contains. If afterwards the reference count of an object in the pool is zero, the object is deallocated.

Putting the Policy Into Practice

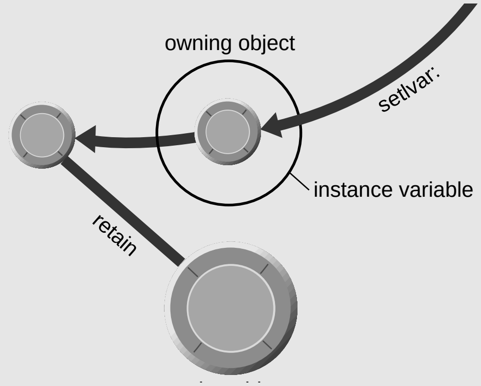
When an object is used solely within the scope of the method that creates it, you can deallocate it immediately by sending it **release**. Otherwise, send **autorelease** to all created objects that you no longer need but will return or pass to other objects.

You shouldn't release objects that you receive from other objects (unless you precede the **release** or **autorelease** with a **retain**). You don't own these objects, and can assume that their owner has seen to their eventual deallocation. You can also assume that (with some exceptions, described below) a received object remains valid within the method it was received in. That method can also safely return the object to its invoker.

You should send **release** or **autorelease** to an object only as many times as are allowed by its creation (one) plus the number of **retain** messages you have sent it. You should never send **free** or **dealloc** to an OpenStep object.

Implications of Retained Objects

When you retain an object, you're sharing it with its owner and other objects that have retained it. While this might be what you want, it can lead to some undesirable consequences. If the owner is released, any object you



A possible side effect of retain: An object that owns an instance variable assigns a new object to it after releasing the previously assigned object. Another object that had retained the prior instance variable is now referencing an invalid object.

received from it and retained can be invalid. If you had retained an instance variable of the owning object, and that instance variable is reassigned, your code could be referencing something it does not expect.

copy Versus retain

When deciding whether to retain or copy objects, it helps to categorize them as *value objects* or *entity objects*. Value objects are objects such as NSNumbers or NSStrings that encapsulate a discrete, limited set of data. Entity objects, such as NSViews and NSWindows, tend to be larger objects that manage and coordinate subordinate objects. For value objects, use **copy** when you want your own "snapshot" of the object (the object must conform to the NSCopying protocol); use **retain** when you intend to share the object. Always retain entity objects.

In accessor methods that set value-object instance variables, you usually (but not always) want to make your own copy of the object and not share it. (Otherwise it might change without your knowing.) Send **autorelease** to the old object and then send **copy**—not **retain**—to the new one:

```
- (void)setTitle:(NSString *)newTitle
{
    [title autorelease];
    title = [newTitle copy];
}
```

OpenStep framework classes can, for reasons of efficiency, return objects cast as immutable when to the owner (the framework class) they are mutable. Thus there is no guarantee that a vended framework object won't change, even if it is of an immutable type. The precaution you should take is evident: copy objects obtained from framework classes if it's important the object shouldn't change from under you.

1 Write the object-initialization and object-deallocation code.

Implement the **init** method, as shown here.

Implement the **dealloc** method, following the suggestions in the Before You Go On section below.

```
- (id)init
{
    [super init];                                   A

    name = @"";                                     B
    airports = @"";
    airlines = @"";
    transportation = @"";
    hotels = @"";
    languages = @"";
    currencyName = @"";
    comments = @"";

    return self;                                    C
}
```

A Invokes **super**'s (the superclass's) **init** method to have inherited instance variables initialized. Always do this first in an **init** method.

B Initializes an NSString instance variable to an empty string. @"" is a compiler-supported construction that creates an immutable NSString object from the text enclosed by the quotes.

You don't need to initialize instance variables to null values (**nil**, zero, NULL, and so on) because the run-time system does it for you. But you should initialize instance variables that take other starting values. Also, don't substitute **nil** when empty objects are expected, and vice versa.The Objective-C keyword **nil** represents a null "object" with an **id** (value) of zero. An empty object (such as @"") is a true object; it just has no "real" content.

C By returning **self** you're returning a true instance of your object; up until this point, the instance is considered undefined.

Before You Go On

Implement the **dealloc** method. In this method you release (that is, send **release** or **autorelease** to) objects that you've created, copied, or retained (which don't have an impending **autorelease**). For the Country class, release all objects held as instance variables. If you had other retained objects, you would release them, and if you had dynamically allocated data, you would free it. When this method completes, the Country object is deallocated. The **dealloc** method should send **dealloc** to **super** as the *last* thing it does, so that the Country object isn't released by its superclass before it's had the chance to release all objects it owns.

Note that **release** itself doesn't deallocate objects, but it leads to their deallocation. For more on **release** and **autorelease**, see "Object Ownership, Retention, and Disposal" on page 88.

You want the Country objects created by the Travel Advisor application to be *persistent*. That is, you want them to “remember” their state between sessions. Archiving lets you do this by encoding the state of application objects in a file along with their class memberships. The NSCoding protocol defines two methods that enable archiving for a class: **encodeWithCoder:** and **initWithCoder:**.

1 **Implement the methods that archive and unarchive the object.**

Implement the **encodeWithCoder:** method as shown at right.

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [coder encodeObject:name];
    [coder encodeObject:airports];
    [coder encodeObject:airlines];
    [coder encodeObject:transportation];
    [coder encodeObject:hotels];
    [coder encodeObject:languages];
    [coder encodeValueOfObjCType:"s" at:&englishSpoken];
    [coder encodeObject:currencyName];
    [coder encodeValueOfObjCType:"f" at:&currencyRate];
    [coder encodeObject:comments];
}
```

The **encodeObject:** method encodes a single object in the archive file. For both object and non-object types, you can use **encodeValueOfObjCType:at:** (shown in this example encoding a string and a float). NSCoder provides other encoding methods.

Implement the **initWithCoder:** method as shown at right.

```
- (id)initWithCoder:(NSCoder *)coder
{
    name = [[coder decodeObject] copy];            A
    airports = [[coder decodeObject] copy];
    airlines = [[coder decodeObject] copy];
    transportation = [[coder decodeObject] copy];
    hotels = [[coder decodeObject] copy];
    languages = [[coder decodeObject] copy];
    [coder decodeValueOfObjCType:"s" at:&englishSpoken];
    currencyName = [[coder decodeObject] copy];
    [coder decodeValueOfObjCType:"f" at:&currencyRate];
    comments = [[coder decodeObject] copy];

    return self;                                   B
 }
```

A The order of decoding should be the same as the order of encoding; since **name** is encoded first it should be decoded first. Use **copy** when you assign value-type objects to instance variables (see “Object Ownership, Retention, and Disposal” on page 88). NSCoder defines **decode...** methods that correspond to the **encode...** methods, which you should use.

B As in any **init...** method, end by returning **self**—an initialized instance.

Implementing the TAController Class

After describing what other instance variables you must add to TAController, this section covers the following implementation tasks:

- Getting the data from Country objects to the interface and back
- Getting the table view to work, including updating Country records
- Adding and deleting "records" (Country objects)
- Formatting and validating field values
- "Housekeeping" tasks (application management)

1 **Update TAController.h.**

Import **Country.h**.

Add the instance-variable declarations shown at right.

```
    NSMutableDictionary *countryDict;
    NSMutableArray      *countryKeys;
    BOOL                recordNeedsSaving;
```

The variables **countryDict** and **countryKeys** identify the array and the dictionary discussed on "Travel Advisor — An Overview" on page 62. The boolean **recordNeedsSaving** flags that record if the user modifies the information in any field.

Add the **enum** declaration shown at right between the last **#import** directive and the **@interface** directive.

```
enum LogisticsFormTags {
    LGairports=0,
    LGairlines,
    LGtransportation,
    LGhotels
};
```

This declaration is not essential, but the **enum** constants provide a clear and convenient way to identify the cells in the Logistics form. Methods such as **cellAtIndex:** identify the editable cells in a form through zero-based indexing. This declaration gives each cell in the Logistics form a meaningful designation.

Turbo Coding With Project Builder

When you write code with Project Builder you have a set of "workbench" tools at your disposal, among them:

Indentation

In Preferences you can set the characters at which indentation automatically occurs, the number of spaces per indentation, and other global indentation characteristics. The Edit menu includes the Indentation submenu, which allows you to indent lines or blocks of code on a case-by-case basis.

Delimiter Checking

Double-click a brace (left or right, it doesn't matter) to locate the matching brace; the code between the braces is highlighted. In a similar fashion, double-click a square bracket in a message expression to locate the matching bracket and double-click a parenthesis character to highlight the code enclosed by the parentheses. If there is no matching delimiter, Project Builder emits a warning beep.

Name Completion

Name completion is a facility that, given a partial name, completes it from all symbols known by the project. You activate it by pressing Escape. You can use name completion in the code editor *and* in all panels where you are finding information or searching for files to open.

As an example: you know there's a certain constant to use with fonts, but you cannot remember it. In your code, type **NSFont**. Then press the Escape key several times. These symbols appear in succession (the found portion is underlined):

```
NSFontIdentityMatrix
NSFontManager
NSFontPanel
```

Emacs Bindings

You can use the most common Emacs commands in Project Builder's code editor. (Emacs is a popular editor for writing code.) For example, there are the commands page-forward (Control-v), word-forward (Meta-f), delete-word (Meta-d), kill-forward (Control-k), and yank from kill ring (Control-y).

Some Emacs commands may conflict with some of the standard Windows key bindings. You can modify the key bindings the code editor uses to substitute other "command" keys—such as the Alternate key or Shift-Control— for Emacs' Control or Meta keys. For instructions on custom key bindings, see "Text Defaults and Key Bindings" in the **Programming Topics** section of **/System/Documentation/Developer/TasksAndConcepts**.

Data Mediation

TAController acts as the mediator of data exchanged between a source of data and the display of that data. Data mediation involves taking data from fields, storing it somewhere, and putting it back into the fields later. TAController has two methods related to data mediation: **populateFields:** puts Country instance data into the fields of Travel Advisor and **extractFields:** updates a Country object with the information in the fields.

1 **Implement the methods that transfer data to and from the application's fields.**

Implement the **populateFields:** method as shown at right.

```
- (void)populateFields:(Country *)aRec
{
    [countryField setStringValue:[aRec name]];

    [[logisticsForm cellAtIndex:LGairports] setStringValue:
        [aRec airports]];
    [[logisticsForm cellAtIndex:LGairlines] setStringValue:
        [aRec airlines]];
    [[logisticsForm cellAtIndex:LGtransportation] setStringValue:
        [aRec transportation]];
    [[logisticsForm cellAtIndex:LGhotels] setStringValue:
        [aRec hotels]];

    [currencyNameField setStringValue:[aRec currencyName]];
    [currencyRateField setFloatValue:[aRec currencyRate]];
    [languagesField setStringValue:[aRec languages]];
    [englishSpokenSwitch setState:[aRec englishSpoken]];

    [commentsField setString:[aRec comments]];

    [countryField selectText:self];
}
```

A Causes the Country field to display the value of the **name** instance variable of the Country record (**aRec**) passed into the method. Since **[aRec name]** is nested, the object it returns is used as the argument of **setStringValue:**, which sets the textual content of the receiver (in this case, an NSFormCell).

B The **cellAtIndex:** message is sent to the form and returns the cell identified by the **enum** constant LGairports.

C Sets the state of the switch according to the boolean value held by the Country instance variable; if the state is YES, the X appears in the switch box.

D Selects the text in the Country field or, if there is no text, inserts the cursor.

Although it doesn't do anything with data, the **blankFields:** method is similar in structure to **populateFields:**. The **blankFields:** method clears whatever appears in Travel Advisor's fields by inserting empty string objects and zeros.

Implement the **blankFields:** method as shown at right.

```
- (void)blankFields:(id)sender
{
    [countryField setStringValue:@""];

    [[logisticsForm cellAtIndex:LGairports] setStringValue:@""];
    [[logisticsForm cellAtIndex:LGairlines] setStringValue:@""];
    [[logisticsForm cellAtIndex:LGtransportation] setStringValue:@""];
    [[logisticsForm cellAtIndex:LGhotels] setStringValue:@""];

    [currencyNameField setStringValue:@""];
    [currencyRateField setFloatValue:0.000];
    [languagesField setStringValue:@""];
    [englishSpokenSwitch setState:NO];          A

    [currencyDollarsField setFloatValue:0.00];
    [currencyLocalField setFloatValue:0.00];
    [celsius setIntValue:0];

    [commentsField setString:@""];              B
    [countryField selectText:self];
}
```

A The **setState:** message affects the appearance of two-state toggled controls, such as a switch button. With an argument of YES, the checkmark appears; with an argument of NO, the checkmark is removed.

B The **setString:** message sets the textual contents of NSText objects (such as the one enclosed by the scroll view).

Before You Go On

Exercise: Implement the **extractFields:** method. In this method set the values of the passed-in Country record's instance variables with the contents of the associated fields.

Here's a little tip for you: This implementation is **extractFields:** in reverse. Use the **stringValue** method to get field contents and use Country's accessor methods to set the values of instance variables.

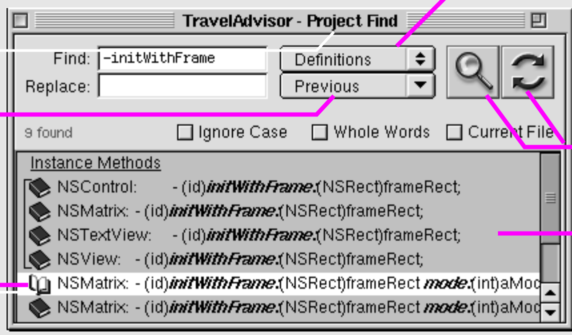
Finding Information Within Your Project

The Project Find Panel

The Project Find panel lets you find any symbol defined or referenced in your project. It also allows you to look up related reference documentation, search for text project-wide using regular expressions, and replace symbols or strings of text. To use the full power of Project Find, your project must be indexed; once it is, you have access to all symbols that the project references, including symbols defined in the frameworks and libraries linked into the project.

Lists the targets of recent find operations; selecting one re-displays the results in the browser.

Click a book icon to see the related reference documentation.



Search for: symbol definition, symbol reference, textual strings (with or without regular expressions).

Find and replace buttons.

Search results.

Click an item to display the relevant code.

Symbol Definition Search Syntax

You can narrow your search for definitions of symbols by indicating type in the Find field of the Project Find panel along with the symbol name. Once the symbol items are listed in the browser, you can click an item to navigate to the definition in the header file, or click a book icon to display the relevant reference documentation.

The following table lists examples of searching for symbol definitions by type:

Example	Finds Definition For
@NSArray	NSArray class
<NSCoding>	NSCoding protocol
-objectAtIndex:	Instance method
+stringWithFormat:	Class method
[NSBox controlView]	Method specific to class
NSRunAlertPanel()	Function
NSApp	Type or constant

Other Ways of Finding Information

Project Builder includes other facilities for finding information:

- **Incremental search**. Control-s brings up the incremental-search panel for the currently edited file. As you type, the cursor advances to the next sequence of characters in the file that match what you type. Click Next (or press Control-s) to go to the next occurrence; click Prev (or press Control-r) to go to the previous occurrence.

  Note thatControl-s might not invoke incremental search on all systems because of different native key bindings on those systems. However, you can customize your key bindings, both generally and specific to Project Builder, and thus get the incremental-search (and other) functionality. See "Turbo Coding With Project Builder" on page 93 for more information.

- **Help**. Project Builder and Interface Builder also feature tool tips, context-sensitive help, and task-related help. See page 56 for details.

Getting the Table View to Work

Table views are objects that display data as records (rows) with attributes (columns). The table view in Travel Advisor displays the simplest kind of record, with each record having only one attribute: a country name.

Table views get the data they display from a *data source*. A data source is an object that implements the informal NSTableDataSource protocol to respond to NSTableView requests for data. Since the NSTableView organizes records by zero-based indexing, it is essential that the data source organizes the data it provides to the NSTableView similarly: in an array.

1 **Implement the behavior of the table view's data source.**

In TAController's a**wakeFromNib** method, create and sort the array of country names.

In the same method, designate **self** as the data source.

```
- (void)awakeFromNib
{
    NSArray *tmpArray = [[countryDict allKeys]                    (A)
        sortedArrayUsingSelector:@selector(compare:)];
    countryKeys = [[NSMutableArray alloc] initWithArray:tmpArray];

    [tableView setDataSource:self];                                (B)
    [tableView sizeLastColumnToFit];
}
```

(A) The **[countryDict allKeys]** message returns an array of keys (country names) from **countryDict**, the unarchived dictionary that contains Country objects as values. The **sortedArrayUsingSelector:** message sorts the items in this "raw" array using the **compare:** method defined by the class of the objects in the array, in this case NSString (this is an example of polymorphism and dynamic binding). The sorted names go into a temporary NSArray—since that is the type of the returned value—and this temporary array is used to create a mutable array, which is then assigned to **countryKeys**. A mutable array is necessary because users may add or delete countries.

(B) The **[tableView setDataSource:self]** message identifies the TAController object as the table view's data source. The table view will commence sending NSTableDataSource messages to TAController. (You can effect the same thing by setting the NSTableView's **dataSource** outlet in Interface Builder.)

If users are supposed to edit the cells of the table view, you could make TAController the delegate of the table view at this point (with **setDelegate:**).The delegate receives messages relating to the editing and validation of cell contents. For details, see the specification on NSTableView in the Application Kit reference documentation.

To fulfill its role as data source, TAController must implement two methods of the NSTableDataSource informal protocol.

Implement two methods of the NSTableDataSource informal protocol:

– **numberOfRowsInTableView:**
– **tableView: objectValueForTableColumn: row:**

```
- (int)numberOfRowsInTableView:(NSTableView *)theTableView      A
{
    return [countryKeys count];
}

- (id)tableView:(NSTableView *)theTableView                     B
      objectValueForTableColumn:(NSTableColumn *)theColumn
            row:(int)rowIndex
{
    if ([[theColumn identifier] isEqualToString:@"Countries"])
        return [countryKeys objectAtIndex:rowIndex];
    else
        return nil;
}
```

A Returns the number of country names in the **countryKeys** array. The table view uses this information to determine how many rows to create.

If you had an application with multiple table views, each table view would invoke this NSTableView delegation method (as well as the others). By evaluating the **theTableView** argument, you could distinguish which table view was involved.

B This method first evaluates the column identifier to determine if it's the right column (it *should* always be "Countries"). If it is, the method returns the country name from the **countryKeys** array that is associated with **rowIndex**. This name is then displayed at **rowIndex** of the column. (Remember, the array and the cells of the column are synchronized in terms of their indices.)

The NSTableDataSource informal protocol has another method, **tableView:setObjectValue:forTableColumn:row:**, that you won't implement in this tutorial. This method allows the data source to extract data entered by users into table-view cells; since Travel Advisor's table view is read-only, there is no need to implement it.

Finally, you have to have the table view respond to mouse clicks in it, which indicate a request that a new record be displayed. As you recall, you defined in Interface Builder the **handleTVClick:** action for this purpose. This method must do a number of things:

- Save the current Country object or create a new one.
- If there's a new record, re-sort the array providing data to the table view.
- Display the selected record.

1 Update records.

Implement the method that responds to user selections in the table view.

```
- (void)handleTVClick:(id)sender
{
    Country *aRec, *newRec, *newerRec;
    int index;

    /* does current obj need to be saved? */
    if (recordNeedsSaving) {                                         A
    /* is current object already in dictionary? */
        if (aRec=[countryDict objectForKey:[countryField stringValue]]) {
            /* remove if it's been changed */
            if (aRec) {
                NSString *country = [aRec name];
                [countryDict removeObjectForKey:country];
                [countryKeys removeObject:country];
            }
        }
        /* Create Country obj, add to dict, add name to keys array */
        newRec = [[Country alloc] init];
        [self extractFields:newRec];
        [countryDict setObject:newRec forKey:[countryField stringValue]];
        [newRec release];
        [countryKeys addObject:[countryField stringValue]];

        /* sort array here */
        [countryKeys sortUsingSelector:@selector(compare:)];
        [tableView reloadData];
    }
    index = [sender selectedRow];
    if (index >= 0 && index < [countryKeys count]) {                 B
        newerRec = [countryDict objectForKey:
            [countryKeys objectAtIndex:index]];
        [self populateFields:newerRec];
        [commentsLabel setStringValue:[NSString stringWithFormat:
            @"Notes and Itinerary for %@", [countryField stringValue]]];
        recordNeedsSaving=NO;
    }
}
```

This method has two major sections, each introduced by an **if** statement.

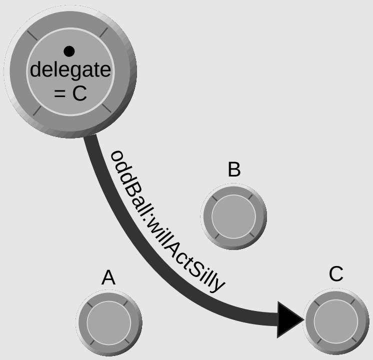
Getting in on the Action: Delegation and Notification

A lot goes on in a running application: events are being interpreted, files are being read, views are being drawn. Because your custom objects might be interested in any of these activities, OpenStep offers two mechanisms through which your objects can participate in or be kept informed of events going on in the application: delegation and notification.

Delegation

Many OpenStep framework objects hold a *delegate* as an instance variable. A delegate is an object that receives messages from the framework object when specific events occur. Delegation messages are of several types, depending on the expected role of the delegate:

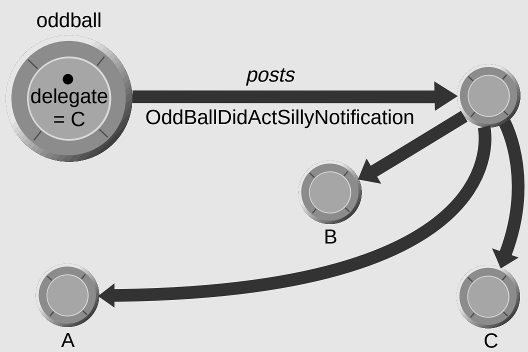
- Some messages are purely informational, occurring after an event has happened. They allow a delegate to coordinate its actions with the other object.
- Some messages are sent before an action will occur, allowing the delegate to veto or permit the action.
- Other delegation messages assign a specific task to a delegate, like filling a browser with cells.



You can set your custom object to be the delegate of a framework object programmatically or in Interface Builder. Your custom classes can also define their own delegate variables and delegation protocols for client objects.

Notification

A notification is a message that is broadcast to all objects in an application that are interested in the event the notification represents. As does the informational delegation message, the notification informs these *observers* that this event took place. It can also pass along relevant data about the event.



Here's the way the notification process works:

- Objects interested in an event that happens elsewhere in the application — say the addition of a record to a database — register themselves with a *notification center* (an instance of NSNotificationCenter) as observers of that event. Delegates of an object that posts notifications are automatically registered as observers of those notifications.
- The object that adds the object to the database (or some such event) *posts* a notification (an instance of NSNotification) to a notification center. The notification contains a tag identifying the notification, the **id** of the associated object, and, optionally, a dictionary of supplemental data.
- The notification center then sends a message to each observer, invoking the method specified by each, and passing in the notification.

Notifications hold some advantages over delegation messages as a means of inter-application communication. They allow an object to synchronize its behavior and state with *multiple* objects in an application, and without having to know the identity of those objects. With *notification queues*, it is also possible to post notifications asynchronously and coalesce similar notifications.

A When any Country-object data is added or altered, Travel Advisor sets the **recordNeedsSaving** flag to YES (you'll learn how to do this later on). If **recordNeedsSaving** is YES, the code first deletes any existing Country record for that country from the dictionary and also removes the country name from the table view's array. (Upon removal, the objects are automatically released by the array.) Then it creates a new Country instance, initializes it with the values currently on the screen, adds the instance to the dictionary, and releases the instance (the dictionary has retained it). For the table view's array, it adds the country name to it, sorts it, and invokes the **reload** method, which causes the table view to request data from its data source.

B The **selectedRow** message queries the table view for the row index of the cell that was clicked. If this index is within the array's bounds, the code uses it to get the country name from the array, and then uses the country name as the key to get the associated Country instance. It writes the instance-variable values of this instance to the fields of the application, updates the "Notes and Itinerary for" label, and resets the **recordNeedsSaving** flag.

Optional Exercise

Users often like to have key alternatives to mouse actions such as clicking a table view. One way of acquiring a key alternative is to add a menu command in Interface Builder, specify a key as an attribute of the command, define an action method that the command will invoke, and then implement that method.

The methods **nextRecord:** and **prevRecord:** should be invoked when users choose Next Record and Prev Record or type the key equivalents Command-n and Command-r. In **TAController.m**, implement these methods, keeping the following hints in mind:

1. Get the index of the selected row (**selectedRow**).

2. Increment or decrement this index, according to which key is pressed (or which command is clicked).

3. If the start or end of the table view is encountered, "wrap" the selection. (Hint: Use the index of the last object in the **countryKeys** array.)

4. Using the index, select the new row, but don't extend the selection.

5. Simulate a mouse click on the new row by sending **handleTVClick:** to **self**.

Breaktime: Build the Project

Now is a good time to take a break and build Travel Advisor. See if there are any errors in your code or in the nib file you've created with Interface Builder. Remember, if you unsure about any of the code discussed so far, especially code that you're encouraged to write on your own as part of an "exercise," refer to the example project in **/System/Developer/Examples/AppKit.** You may also want to take this time to test drive Project Builder's graphical debugger, discussed on the following two pages.

Adding and Deleting Records

When users click Add Record to enter a Country "record," the **addRecord:** method is invoked. You want this method to do a few things besides adding a Country object to the application's dictionary:

- Ensure that a country name has been entered.
- Make the table view reflect the new record.
- If the record already exists, update it (but only if it's been modified).

1 **Implement the method that adds a Country object to the NSDictionary "database."**

```
- (void)addRecord:(id)sender
{
    Country *aCountry;
    NSString *countryName = [countryField stringValue];

   if (countryName && (![countryName isEqualToString:@""])) {  A
        aCountry = [countryDict objectForKey:countryName];
        if (aCountry && recordNeedsSaving) {
           /* remove old Country object from dictionary */
           [countryDict removeObjectForKey:countryName];
           [countryKeys removeObject:countryName];
           aCountry = nil;
         }
        if (!aCountry)  /* record is new or has been removed */
            aCountry = [[Country alloc] init];
        else /* record already exists and hasn't changed */
            return;

        [self extractFields:aCountry];                         B
        [countryDict setObject:aCountry forKey:[aCountry name]];
        [countryKeys addObject:[aCountry name]];
        [countryKeys sortUsingSelector:@selector(compare:)];

        recordNeedsSaving=NO;                                  C
        [commentsLabel setStringValue:[NSString stringWithFormat:
            @"Notes and Itinerary for %@",[countryField stringValue]]];
        [countryField selectText:self];

        [tableView reloadData];                                D
        [tableView selectRow:[countryKeys indexOfObject:
            [aCountry name]] byExtendingSelection:NO];
    }
}
```

A This section of code verifies that a country name has been entered and sees if there is a Country object in the dictionary. If there's no object for the key, **objectForKey:** returns **nil**. If the object exists and it's flagged as modified, the code removes it from the dictionary and removes the country name from the **countryKeys** array. Note that removing an object from a dictionary or array also releases it, so the code sets **aCountry** to **nil**. It then tests **aCountry** and, if it's **nil**, creates a new object; otherwise it just returns, because an object already exists for this country and it hasn't been modified.

B After updating the new Country object with the information on the application's fields (**extractFields:**), this code adds the Country object to the dictionary and the country name to the **countryKeys** array.

C This section of code performs some things that have to be done, such as resetting the **recordNeedsSaving** flag and updating the label over the scroll view to reflect the just-added country.

D The **reloadData** message forces the table view to update its contents. The **selectRow:byExtendingSelection:** message highlights the new record in the table view.

Note: In the code example on the previous page, note the expression "if (!aCountry)". For objects, this is shorthand for "if (aCountry == nil)"; in the same vein, "if (aCountry)" is equivalent to "if (aCountry != nil)".

Before You Go On

Exercise: Implement the **deleteRecord:** method. Although similar in structure to **addRecord:** this method is much simpler, because you don't need to worry about whether a Country record has been modified. Once you've deleted the record, remember to update the table view and clear the fields of the application.

Flattening the Object Network: Coding and Archiving

Coding, as implemented by NSCoder, takes a network of objects such as exist in an application and serializes that data, capturing the state, structure, relationships, and class memberships of the objects. As a subclass of NSCoder, NSArchiver extends this behavior by storing the serialized data in a file.

When you archive a root object, you archive not only that object but all other objects the root object references, all objects those second-level objects reference, and so on. To be archived, however, objects must conform to the NSCoding protocol. This conformance requires that they implement the **encodeWithCoder:** and **initWithCoder:** methods.

Thus sending **archiveRootObject:toFile:** to NSArchiver leads to the invocation of **encodeWithCoder:** in the root object and in all referenced objects that implement it. Similarly, sending **unarchiveObjectWithFile:** to NSUnarchiver results in **initWithCoder:** being invoked in those objects referenced in the archive file. These objects reconstitute themselves from the instance data in the file. In this way, the network of objects, three-dimensional in abstraction, is converted to a two-dimensional stream of data and back again.

Field Validation

The NSControl class gives you an API for validating the contents of cells. Validation verifies that the values of cells fall within certain limits or meet certain criteria. In Travel Advisor, we want to make sure that the user does not enter a negative value in the Rate field.

The request for validation is a message—**control:isValidObject:**—that a control sends to its delegate. The control, in this case, is the Rate field.

1 **Validate the values entered in a field.**

In **awakeFromNib**, make TAController a delegate of the field to be validated: the Rate field.

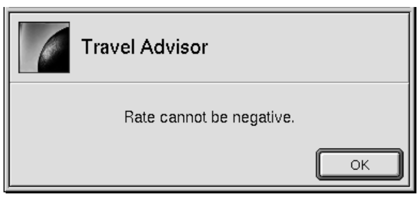
Implement the **control:isValidObject:** method to validate the value of the field.

```
[currencyRateField setDelegate:self];
```

```
- (BOOL)control:(NSControl *)control isValidObject:(id)obj
{
    if (control == currencyRateField) {
        if ([obj floatValue] < 0.0) {
            NSRunAlertPanel(@"Travel Advisor",
                @"Rate cannot be negative.", nil, nil, nil);
            return NO;
        }
    }
    return YES;
}
```

Ⓐ Because you might have more than one field's value to validate, this example first determines which field is sending the message. It then checks the field's value (passed in as the second object); if it is negative, it displays a message box and returns NO, blocking the entry of the value. Otherwise, it returns YES and the field accepts the value.

Ⓑ The **NSRunAlertPanel()** function allows you to display an attention panel from any point in your code. The above example calls this function simply to inform the user why the value cannot be accepted.



Although Travel Advisor doesn't evaluate it, the **NSRunAlertPanel()** function returns a constant indicating which button the user clicks on the message box. The logic of your code could therefore branch according to user input. In addition, the function allows you to insert variable information (using **printf()**-style conversion specifiers) into the body of the message.

Application Management

By now you've finished the major coding tasks for Travel Advisor. All that remains to implement are a half dozen or so methods. Some of these methods perform tasks that every application should do. Others provide bits of functionality that Travel Advisor requires. In this section you'll:

- Archive and unarchive the TAController object.
- Implement TAController's **init** and **dealloc** methods.
- Save data when the application terminates.
- Mark the current record when users make a change.
- Obtain and display converted currency values.

The data that users enter into Travel Advisor should be saved in the file system, or *archived*. The best time to initiate archiving in Travel Advisor is when the application is about to terminate. Earlier you made TAController the delegate of the application object (NSApp). Now respond to the delegate message **applicationShouldTerminate:**, which is sent just before the application terminates.

1 **Archive the application's objects when it terminates.**

Implement the delegate method **applicationShouldTerminate:**, as shown at right.

```
- (BOOL)applicationShouldTerminate:(id)sender
{                                                              A
   NSString *storePath = [[[NSBundle mainBundle] resourcePath]
        stringByAppendingPathComponent:@"TravelData"];
   /* save current record if it is new or changed */
   [self addRecord:self];
                                                               B
   if (countryDict && [countryDict count])
        [NSArchiver archiveRootObject:countryDict toFile:storePath];

   return YES;
}
```

A Constructs a pathname for the archive file, "TravelData." This file is stored in the resource directory of the application's main bundle. The application wrapper—the directory holding the application executable and the resource directory—is a bundle (the *main bundle*), so NSBundle methods are used to get the path to this directory.

This technique of storing application data in the main bundle is for the purposes of demonstrating NSBundle APIs and is not recommended for most applications. See the following chapter, "To Do Tutorial—The Basics," for examples and explanations of storing user-specific document data in the file system.

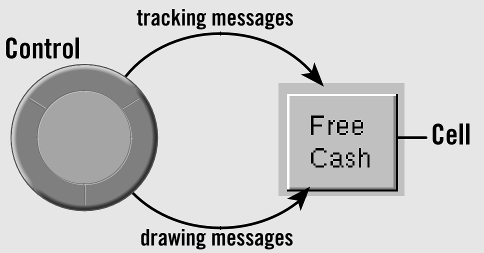
B If the **countryDict** dictionary holds Country objects, TAController archives it with the NSArchiver class method **archiveRootObject:toFile:**. Since the dictionary is designated as the root object for archiving, all objects that the dictionary references (that is, the Country objects it contains) will be archived too.

Behind 'Click Here': Controls, Cells, and Formatters

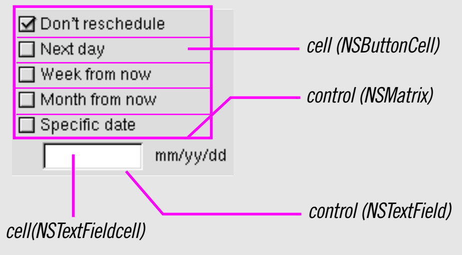
Controls and cells lie behind the appearance and behavior of most user-interface objects in OpenStep, including buttons, text fields, sliders, and browsers. Although they are quite different types of objects—controls inherit from NSControl while cells inherit from NSCell—they interact closely.

Controls enable users to signal their intentions to an application, and thus to *control* what is happening. By interpreting mouse and keyboard events and asking another object to respond to them, controls implement the target/action paradigm described in "Paths for Object Communication: Outlets, Targets, and Actions" on page 40. Controls themselves can hold targets and actions as instance variables, but usually they get this data from the affected cell (which must inherit from NSActionCell).

Cells are rectangular areas "embedded" within a control. A control can hold multiple cells as a way to partition its surface into active areas. Cells can draw their own contents either as text or image (and sometimes as both), and they can respond individually to user actions. Since cells are typically more frugal consumers of memory than controls, they help applications be more efficient.



Controls act as managers of their cells, telling them when and where to draw, and notifying them when a user event (mouse clicks, keystrokes) occurs in their areas. This division of labor, given the relative "weight" of cells and controls, provides a great boost to application performance.

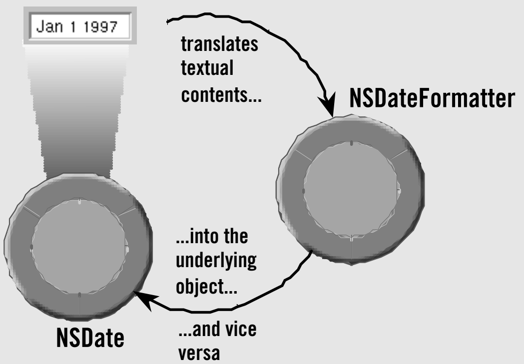


A control does not have to have a cell associated with it, but most user-interface objects available on Interface Builder's standard palettes are cell-control combinations. Even a simple button—from Interface Builder or programmatically created—is a control (an NSButton instance) associated with an NSButtonCell. The cells in a control such as a matrix must be the same size, but they can be of different classes. More complex controls, such as table views and browsers, can incorporate various types of cells.

Cells and Formatters

When one thinks of the contents of cells, it's natural to consider only text (NSString) and images (NSImage). The content seems to be whatever is displayed. However, cells can hold other kinds of objects, such as dates (NSDate), numbers (NSNumber), and custom objects (say, phone-number objects).

Formatter objects handle the textual representation of the objects associated with cells and translate what is typed into a cell into the underlying object. Using NSCell's **setFormatter:**, you must programmatically associate a formatter with a cell to get this behavior.



The Foundation framework provides the NSDateFormatter and NSNumberFormatter classes to generate date formatters and currency and number formatters. You can make a custom subclass of NSFormatter to derive your own formatters.

1 **Implement TAController's methods for initializing and deallocating itself.**

Implement the **init** method, as shown at right.

Implement the **dealloc** method to release object instance variables.

```
- (id)init
{
    NSString *storePath = [[NSBundle mainBundle]                A
        pathForResource:@"TravelData" ofType:nil];
    [super init];
    countryDict =
       [NSUnarchiver unarchiveObjectWithFile:storePath];        B

    if (!countryDict) {                                         C
        countryDict = [[NSMutableDictionary alloc] init];
        countryKeys = [[NSMutableArray alloc] initWithCapacity:10];
    } else
        countryDict = [countryDict retain];
    recordNeedsSaving=NO;

    return self;
}
```

A Using NSBundle methods, locates the archive file "TravelData" in the application wrapper and returns the path to it.

B The **unarchiveObjectWithFile:** message *unarchives* (that is, restores) the object whose attributes are encoded in the specified file. The object that is unarchived and returned is the NSDictionary of Country objects (**countryDict**).

C If no NSDictionary is unarchived, the **countryDict** instance variable remains **nil**. If this is the case, TAController creates an empty **countryDict** dictionary and an empty **countryKeys** array. Otherwise, it retains the instance variable.

When users modify data in fields of Travel Advisor, you want to mark the current record as modified so later you'll know to save it. The Application Kit broadcasts a notification whenever text in the application is altered. To receive this notification, add TAController to the list of the notification's observers.

1 **Write the code that marks records as modified.**

In the **awakeFromNib** method, make TAController an observer of NSControlTextDidChangeNotification.

Implement **textDidChange:** to set the **recordNeedsSaving** flag.

```
    [[NSNotificationCenter defaultCenter] addObserver:self
         selector:@selector(textDidChange:)
         name:NSControlTextDidChangeNotification object:nil];
```

Next, implement the method that you indicated would respond to the notification; this method sets a flag, thereby marking the record as changed.

```
- (void)textDidChange:(NSNotification *)notification
{
    if ([notification object] == currencyDollarsField ||
        [notification object] == celsius) return;

    recordNeedsSaving=YES;
}
```

You post notifications and add objects as observers of notifications with methods defined in the NSNotificationCenter class. NSNotification defines methods for creating notification objects and for accessing their attributes. See the specifications of these classes in the Foundation framework reference documentation.

Two of the editable fields of Travel Advisor hold temporary values used in conversions and so are not saved. This statement checks if these fields are the ones originating the notification and, if they are, returns without setting the flag. (The **object** message obtains the object associated with the notification.)

The final method to implement is almost identical to the one you wrote for Currency Converter to display the results of a currency conversion when the user clicks the Convert button for currency conversion.

1 **Implement the method that responds to a request for a currency conversion.**

```
- (void)convertCurrency:(id)sender
{
    [currencyLocalField setFloatValue:
        [converter convertAmount:[currencyDollarsField floatValue]
        byRate:[currencyRateField floatValue]]];
}
```

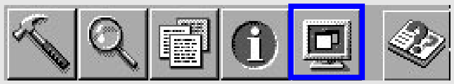
Optional Exercise

Convert Celsius to Fahrenheit: Implement the **convertCelsius:** method. You've already specified and connected the necessary outlets (**celsius**, **fahrenheit**) and action (**convertCelsius:**), so all that remains is the method implementation. The formula you'll need is:

F° = 9/5C° + 32

Using the Graphical Debugger

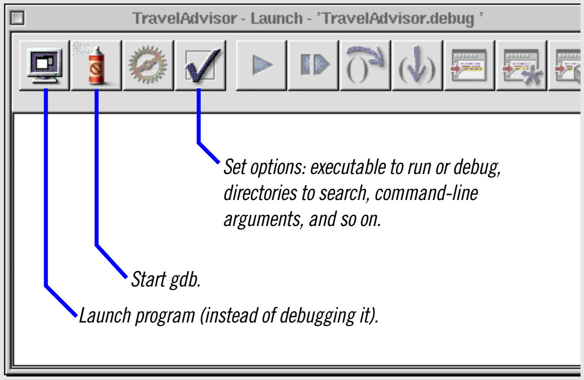
To smooth the task of debugging, Project Builder puts a graphical user interface over the GNU debugger, **gdb**. To access the Launch panel that serves as this graphical debugger, click the button outlined at right.



1. Run the debugger.

The Launch panel allows you to run programs or debug them. If you want to debug a program, start up gdb by clicking this button:

Before you run **gdb** you should first build your project with a target of "debug" to get an executable with full debugging information. You should also verify that the proper executable is being debugged. To select the "debug" executable for debugging, click the checkmark button and, in the Executables display of the Launch Options panel, choose the file with an extension of **debug**.

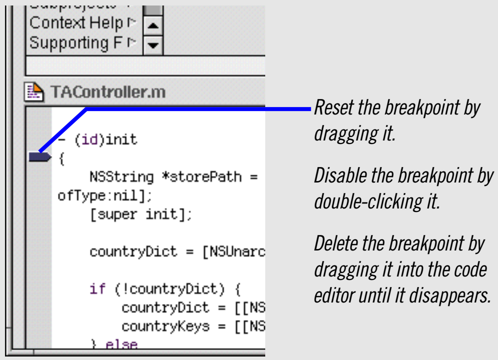


2. Set a breakpoint.

When you start the debugger, a narrow gray band appears along the left margin of the code editor. You set a breakpoint by double-clicking in the gray band next to a line of code.

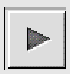
You can see which breakpoints are set in the Breakpoints display of the Task Inspector, which you access by clicking this button:

In this inspector, you can disable and re-enable breakpoints by double-clicking under the "Use?" column.



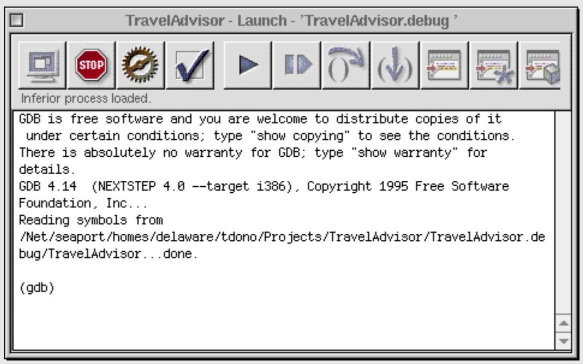
3. Start debugging the application.

To begin debugging an application click the right-triangle button:



The application starts up. If necessary, use the application until the first breakpoint is encountered. When that happens, the "(gdb)" prompt appears in the command-line section of the panel.

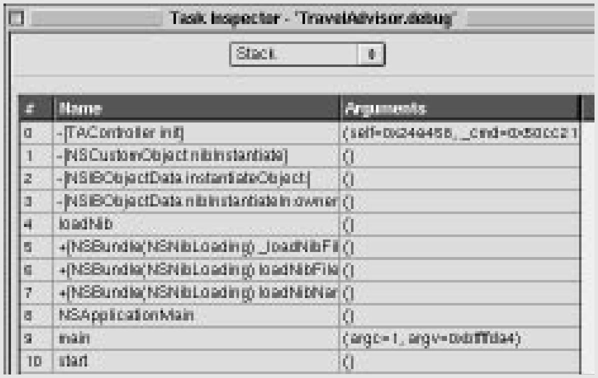
You can type **gdb** commands at this prompt. There are many **gdb** commands not represented in the user interface. For on-line information on these commands, enter "help" at the prompt. You can also find more about commands in the on-line **gdb** reference.



4. Inspect the stack trace.

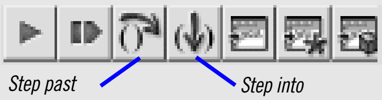
When a program running under the debugger hits a breakpoint, the graphical debugger displays a trace of the call stack. You can see the sequence of calls leading up to the breakpoint as well as the values of arguments of methods or functions implemented by your project.

The Stack display is part of the Task Inspector, which you open by clicking the following button on the Launch panel:

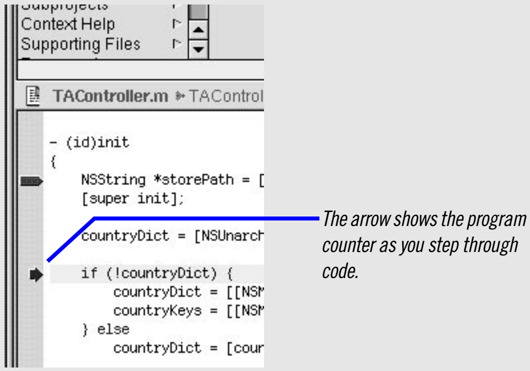


5. Step through code.

When the program you're debugging hits a breakpoint, you usually want to step through a section of the code and see what happens (in terms of the stack and the values of variables). The Launch panel gives you two buttons for stepping through code.

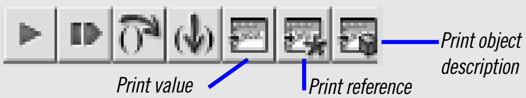


You can step *into* code (going from a call site to an invoked method or called function) only with code that your project implements.

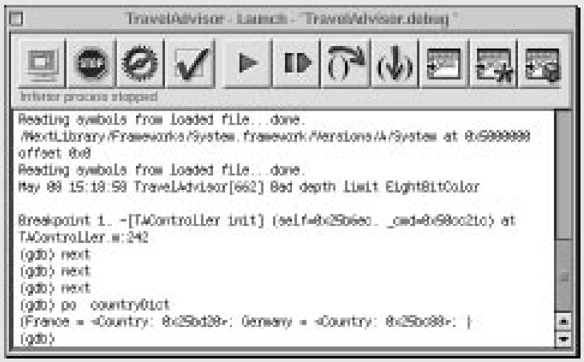


6. Examine data values.

With the graphical debugger, you can inspect the values of variables, pointers, and objects as you step through code. First select a symbol in the code *after* the statement in which it appears has been executed. Then click one of the "print" buttons to learn about its present value:



The **gdb** command-line section of the Launch panel then displays the requested value. When you click the rightmost button and an object is selected, that object's **description** method is invoked. If you are debugging your own objects, it might be worthwhile to implement the **description** method to yield information as precise and detailed as is required (see page 124 for an example of this).



For more information on debugging, see the on-line Help for Project Builder.

Building and Running Travel Advisor

When Travel Advisor is built, start it up by double-clicking the icon in the File Manager. Then put the application through the following tests:

- Enter a few records. Make up geographical information if you have to—you're not trusting your future travels to this application. Not yet, anyway.
- Click the items in the table view and notice how the selected records are displayed. Press Command-n and Command-r and observe what happens.
- Enter values in the conversion fields to see how they're automatically formatted. Try to enter a negative value in the Rate field.
- Quit the application and then start it up again. Notice how the application displays the same records that you entered.

Tips for Eliminating Deallocation Bugs

Problems in object deallocation are not unusual in OpenStep applications under development. You might release an object too many times or you might not release an object as many times as is needed to deallocate it. Both situations lead to nasty problems —in the first case, to run-time errors when your code references non-existent objects; the second case leads to memory leaks.

If you're releasing an object too many times, you'll get run-time error messages telling you that a message was sent to a freed object. To find which methods were releasing the object, in **gdb** or the graphical debugger:

1 Set a breakpoint on **main()** and run the program.

2 When you hit the breakpoint, send **enableFreedObjectCheck:** to NSAutoreleasePool with an argument of YES.

3 Set a breakpoint on **_NSAutoreleaseFreedObject**.

4 Continue running the program.

5 When the program hits the breakpoint, do a backtrace and check the stack to find the method releasing the object.

Avoiding Deallocation Errors

Here's a few things to remember that might help you avoid deallocation bugs in OpenStep code:

- Make sure there's an **alloc**, **copy**, **mutableCopy**, or **retain** message sent to an object for each **release** or **autorelease** sent to it.
- When you release a collection object (such as an NSArray), you release all objects stored in it as well. When you add an object to a collection, it's retained; when you remove an object from a collection, it's released.
- Superviews retain subviews as you add them to the view hierarchy and release subviews as you release them. If you want to keep swapped-out views, you should retain them. Similarly, when you replace a window's or box's content view, the old view is released and the new view is retained.
- To avoid retain cycles, objects should not retain their delegates. Objects also should not retain their outlets, since they do not own them.

To Do Tutorial - The Basics Chapter 4

4

What You'll Learn

Designing a multi-document application

Managing documents

Extending an Application Kit class

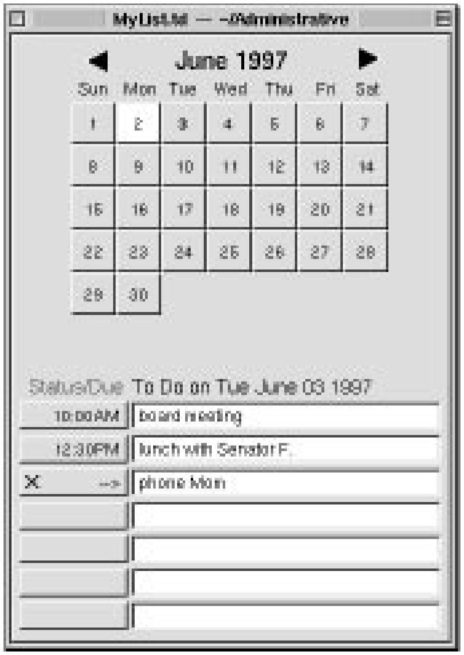
Loading code and resources dynamically

Opening and saving files

Manipulating times and dates

Reading and setting user defaults

The core program framework



*You can find the To Do project in the **AppKit** subdirectory of **/System/Developer/Examples.***

Chapter 4
A Multi-Document Application

Many kinds of applications—word processors and spreadsheets, to name a couple—are designed with the notion of a document *in mind. A document is a body of information, usually contained by a window, that is self-contained and repeatable. Users can create, modify, store, and access a document as a discrete unit. Multi-document applications (as these programs are called) can generate an almost unlimited number of documents.*

The To Do application presented in this chapter is a multi-document application. It is a fairly simple personal information manager. Each To Do document captures the daily "must-do" items for a particular purpose. For instance, one could have a To Do list for work and another one for home.

This chapter guides you through the steps needed to make To Do a multi-document application. When you finish this tutorial, the completed application will allow users to go to specific dates on a calendar and enter a list appointments or tasks for a particular days.

The Design of To Do

The To Do application vaults past Travel Advisor in terms of complexity. Instead of Travel Advisor's one nib file, To Do has three nib files. Instead of three custom classes, To Do has seven. The diagram at the bottom of this page shows the interrelationships among instances of some of those classes and the nib files that they load.

Some of the objects in this diagram are familiar, fitting as they do into the Model-View-Controller paradigm. The ToDoItem class provides the model objects for the application; instances of this class encapsulate the data associated with the items appearing in documents. They also offer functions for computing subsets of that data. And then there's the controller object—actually, there is more than one controller object.

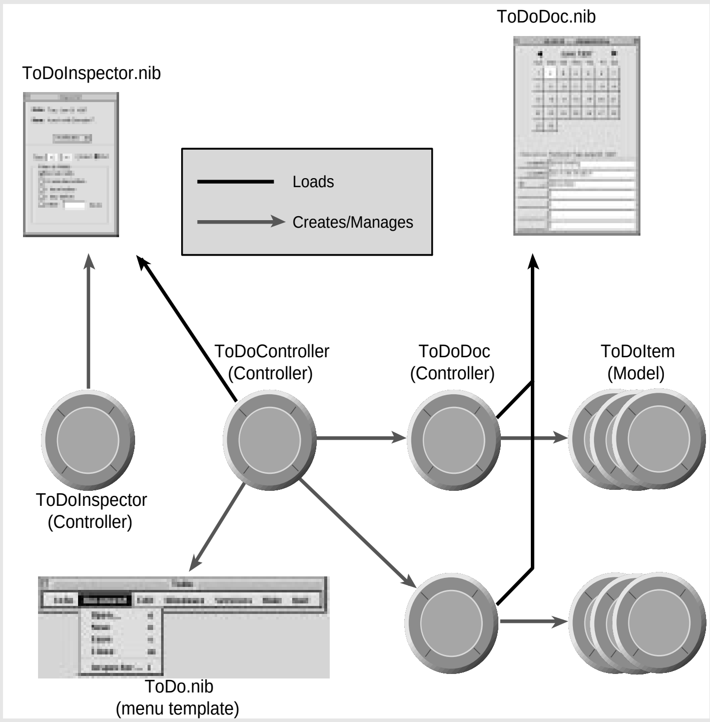
To Do's Multi-Document Design

Two types of controller objects are at the heart of multi-document application design. They claim different areas of responsibility within an application. ToDoController is the *application controller*; it manages events that affect the application as a whole. Each ToDoDoc object is a *document controller*, and manages a single document, including all the ToDoItems that belong to the document. Naturally, it's essential that the application controller be able to communicate with its (potentially) numerous document controllers, and they with it.

The File menu, which Interface Builder includes by default on the menu bar, contains the commands that multi-document applications typically need. When users choose New from the File menu, the application controller allocates and initializes an instance of the ToDoDoc class. When the ToDoDoc instance initializes itself, it loads the **ToDoDoc.nib** file. When the user has finished entering items into the document and chooses Save from the File menu, a Save dialog box appears and the user saves the document in the file system under an assigned name. Later, the user can open the document using the Open menu command, which causes the Open dialog box to be displayed.

The controller objects of To Do respond to a variety of delegation messages sent when certain events occur—primarily from windows and the application object (NSApp)—in order to save and store object state. One example of such an event is when the user closes a document window; another is when data is entered into a document. Often when these events happen, one controller sends a message or notification to the other controller to keep it informed.

The ToDoInspector instance in this diagram takes on some of the work that the application controller, ToDoController, could do. By breaking down a problem domain into distinct areas of responsibility, and assigning certain types of objects to each area, you increase the modularity and reusability of the object, and make maintenance and troubleshooting easier. See "Object-Oriented Programming" in the appendix for more on this.

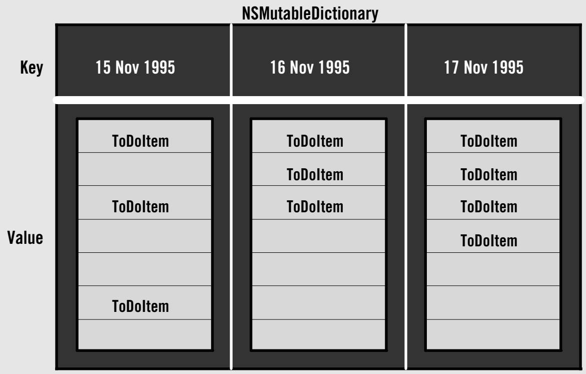


How To Do Stores and Accesses its Data

The data elements of a To Do document (ToDoDoc) are ToDoItems. When a user enters an item in a document's list, the ToDoDoc creates a ToDoItem and inserts that object into a mutable array (NSMutableArray); the ToDoItem occupies the same position in the array as the item in the matrix's text field. This positional correspondence of objects in the array and items in the matrix is an essential part of the design. For instance, when users delete the first entry in the document's list, the document removes the corresponding ToDoItem (at index 0) from the array.

The array of ToDoItems is associated with a particular day. Thus the data for a document consists of a (mutable) dictionary with arrays of ToDoItems for values and dates for keys.

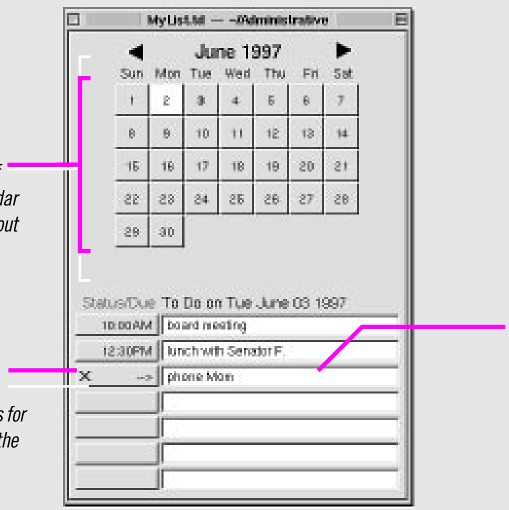
When users select a day in the calendar, the application computes the date, which it then uses as the key to locate an array of ToDoItems in the dictionary.



For further discussion of the architecture of multi-document applications, *see page 139.*

To Do's Custom Views

The discussion so far has touched on model objects and controller objects, but has said nothing about the second member of the Model-View-Controller triad: view objects. Unlike Travel Advisor, which uses only "off-the-shelf" views, To Do's final interface features objects from three custom Application Kit subclasses. (You'll create only CalendarMatrix in this chapter.)



CalendarMatrix (subclass of NSMatrix): A dynamic calendar that notifies its delegate about selected dates.

ToDoCell (subclass of NSButtonCell): A tri-state control with different images for each state. It also displays the times when items are due.

SelectionNotifMatrix (subclass of NSMatrix): Notifies observing objects when a selection in a text field occurs.

Setting Up the To Do Project

1 **Create the application project.**

Start Project Builder.

Choose New from the Project menu.

Set the project type to Application.

Name the application “ToDo.”

Click OK.

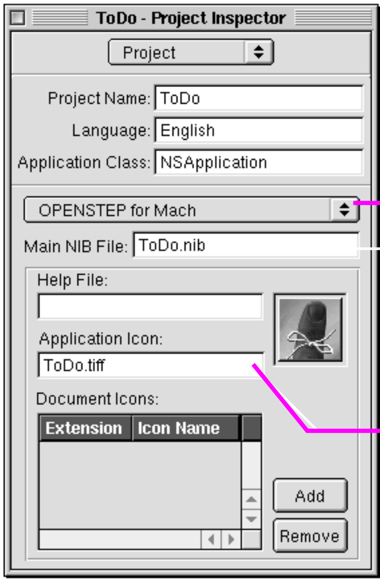
Create the To Do project almost in the same way you created the Travel Advisor application. There are a few differences; each, of course, has a different name and icon. But the most important difference is that To Do has its own document type.

1 **Add the application icon.**

In the Project Attributes display of the project inspector, drag the application icon (**ToDo.tiff**) into the icon well.

Confirm that you want the image added to the project.

(The icon is in the ToDo project in **/System/Developer/Examples/AppKit**.)



You can have different icons and other project attributes for Rhapsody and Yellow Box for Windows.

Instead of dragging the image file into the well, you can add the image file to the project and then just type the name of the file here.

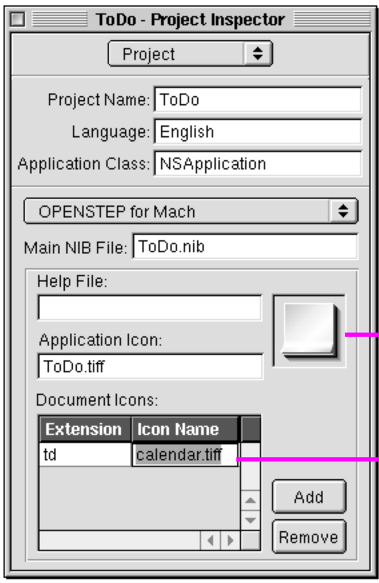
1 **Specify the To Do document type.**

Click Add.

Double-click the new cell under the Extension column.

Type the extension of To Do documents: “td”.

Drag into the icon well the file **calendar.tiff** from the ToDo project in **/System/Developer/Examples/AppKit**.



Document types specify the kinds of files the application can open and “understand.” Documents appear in the desktop with the assigned icon. Double-clicking the icon opens the document.

As with the application icon, when you drag the document icon into the image well (with the document row selected in Document Icons), the image file is added to the project.

Before Project Builder accepts the document icon, you must assign the extension (if the type is new) and select the row.

If the document type is well-known (for example, “.c”) just drag a document of that type into the well.

Creating the Model Class (ToDoItem)

The ToDoItem class provides the model objects for the To Do application. Its instance variables hold the data that defines tasks that should be done or appointments that have to be kept. Its methods allow access to this data. In addition, it provides functions that perform helpful calculations with that data. ToDoItem thus encapsulates both data *and* behavior that goes beyond accessing data.

Since ToDoItem is a model class, it has no user-interface duties and so the expedient course is to create the class without using Interface Builder. We first add the class to the project; Project Builder helps out by generating template source-code files.

1 **Add the ToDoItem class to the project.**

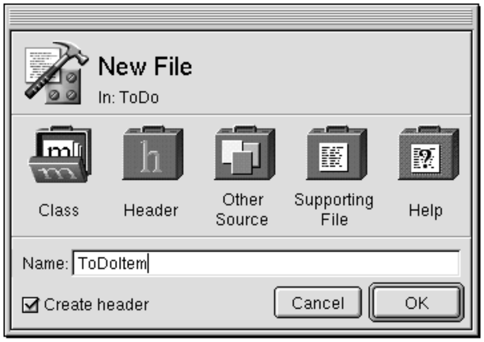
Select Classes in the project browser.

Choose New In Project from the File menu.

In the New File In ToDo panel, type "ToDoItem" in the Name field.

Make sure the "Create header" switch is checked.

Click the OK button.



Setting Up the Programmatic Interface

As you've done before with Travel Advisor, start by declaring instance variables and methods in the header file, **ToDoItem.h**.

1 **Declare ToDoItem's instance variables and methods.**

Type the instance variables as shown at right.

Indicate the protocols adopted by this class.

```
@interface ToDoItem:NSObject<NSCoding, NSCopying>
{
    NSCalendarDate *day;
    NSString *itemName;
    NSString *notes;
    NSTimer *itemTimer;
    long secsUntilDue;
    long secsUntilNotif;
    ToDoItemStatus itemStatus;
}
```

You are adopting the NSCopying protocol in addition to the NSCoding protocol because you are going to implement a method that makes "snapshot" copies of ToDoItem instances.

Instance Variable	What it Holds
day	The day (a date resolved to 12:00 AM) of the to-do item
itemName	The name of the to-do item (the content's of a document text field)
notes	The contents of the inspector's Notes display; this could be any information related to the to-do item, such as an agenda to discuss at a meeting
itemTimer	A timer for notification messages
secsUntilDue	The seconds after day at which the item comes due
secsUntilNotif	The seconds after day at which a notification is sent (before secsUntilDue)
itemStatus	Either "incomplete," "complete," or "deferToNextDay"

Type the method declarations shown at right.

```
- (id)initWithName:(NSString *)name andDate:(NSCalendarDate *)date;
- (void)dealloc;
- (BOOL)isEqual:(id)anObject;
- (id)copyWithZone:(NSZone *)zone;
- (id)initWithCoder:(NSCoder *)coder;
- (void)encodeWithCoder:(NSCoder *)coder;
- (void)setDay:(NSCalendarDate *)newDay;
- (NSCalendarDate *)day;
- (void)setItemName:(NSString *)newName;
- (NSString *)itemName;
- (void)setNotes:(NSString *)notes;
- (NSString *)notes;
- (void)setItemTimer:(NSTimer *)aTimer;
- (NSTimer *)itemTimer;
- (void)setSecsUntilDue:(long)secs;
- (long)secsUntilDue;
- (void)setSecsUntilNotif:(long)secs;
- (long)secsUntilNotif;
- (void)setItemStatus:(ToDoItemStatus)newStatus;
- (ToDoItemStatus)itemStatus;
```

1 **Define enum constants for use in ToDoItem's methods.**

Define these constants before the **@interface** directive.

```
typedef enum ToDoItemStatus {
    incomplete=0,
    complete,
    deferToNextDay
} ToDoItemStatus;

enum {
    minInSecs = 60,
    hrInSecs = (minInSecs * 60),
    dayInSecs = (hrInSecs * 24),
    weekInSecs = (dayInSecs * 7)
};
```

The first set of constants are values for the **itemStatus** instance variable. The second set of constants are for convenience and clarity in the methods that deal with temporal values.

1 **Declare two time-conversion functions.**

```
BOOL ConvertSecondsToTime(long secs, int *hour, int *minute);
long ConvertTimeToSeconds(int hr, int min, BOOL flag);
```

These functions provide computational services to clients of this class, converting time in seconds to hours and minutes (as required by the user interface), and back again to seconds (as stored by ToDoItem).

Before You Go On

Remember, build the project frequently to catch any errors quickly, to get a sense of how the application is developing, and (just as important) to give yourself a break from coding.

Specifying Basic Object Behavior

Most of the method declarations of this class are for accessor methods. You know from past experience what you must do to implement them.

1 **Implement accessor methods.**

Open **ToDoItem.m** in the code editor.

Implement methods that get and set the values of ToDoItem's instance variables.

Implement the **setItemTimer**: method as shown at right.

```
- (void)setItemTimer:(NSTimer *)aTimer
{
    if (itemTimer) {
        [itemTimer invalidate];
        [itemTimer autorelease];
    }
    itemTimer = [aTimer retain];
}
```

The **setItemTimer**: method is slightly different from the other "set" accessor methods. It sends **invalidate** to **itemTimer** to disable the timer before it autoreleases it.

Timers (instances of NSTimer) are always associated with a run loop (an instance of NSRunLoop). See "Tick Tock Brrrring: Run Loops and Timers" on page 198 for more on timers and run loops.

In this application, you want client objects to be able to copy your ToDoItem objects and test them for equality. You must define this behavior yourself.

Starting Up — What Happens in NSApplicationMain()

Every Rhapsody application project created through Project Builder has the same **main()** function (in the file *ApplicationName*_**main.m**). When users double-click an application or document icon in the File Manager or Explorer, **main()** (the entry point) is called first; **main**(), in turn, calls **NSApplicationMain()**—and that's all it does.

The **NSApplicationMain()** function does what's necessary to get an Rhapsody application up and running—responding to events, coordinating the activity of its objects, and so on. The function starts the network of objects in the application sending messages to each other. Specifically, **NSApplicationMain()**:

1 Gets the application's attributes, which are stored in the application wrapper as a property list. From this property list, it gets the names of the main nib file and the principal class (for applications, this is NSApplication or a custom subclass of NSApplication).

2 Gets the Class object for NSApplication and invokes its **sharedApplication** class method, creating an instance of NSApplication, which is stored in the global variable, NSApp. Creating the NSApplication object connects the application to the window system and the Display PostScript server, and initializes its PostScript environment.

3 Loads the main nib file, specifying NSApp as the owner. Loading unarchives and re-creates application objects and restores the connections between objects.

4 Runs the application by starting the main event loop. Each time through the loop, the application object gets the next available event and dispatches it to the most appropriate object in the application. The loop continues until the application object receives a **stop:** or **terminate:** message, after which the application is released and the program exits.

You can add your own code to **main()** to customize application start-up or termination behavior.

1 **Implement copying and comparing object behavior.**

Implement the **isEqual:** method.

```
- (BOOL)isEqual:(id)anObj
{
    if ([anObj isKindOfClass:[ToDoItem class]] &&
        [itemName isEqualToString:[anObj itemName]] &&
        [day isEqualToDate:[anObj day]])
        return YES;
    else
        return NO;
}
```

The default implementation of **isEqual:** (in NSObject) is based on pointer equality. However, ToDoItem has a different basis for equality; any two ToDoItem objects for the same calendar day and having the same item name are considered equal. The implementation of **isEqual:** overrides NSObject to make these tests. (Note that it invokes NSString's and NSDate's own **isEqual...** methods for the specific tests.)

Before You Go On

There is a specific as well as a general need for the **isEqual:** override. In the To Do application, an NSArray contains a day's ToDoItems. To access them, other objects in the application invoke several NSArray methods that, in turn, invoke the **isEqual:** method of each object in the array.

Implement the **copyWithZone:** method.

```
- (id)copyWithZone:(NSZone *)zone
{
    ToDoItem *newobj = [[ToDoItem allocWithZone:zone]
        initWithName:itemName andDate:day];
    [newobj setNotes:notes];
    [newobj setItemStatus:itemStatus];
    [newobj setSecsUntilDue:secsUntilDue];
    [newobj setSecsUntilNotif:secsUntilNotif];

    return newobj;
}
```

This implementation of the **copyWithZone:** protocol method makes a copy of a ToDoItem instance that is an independent replicate of the original (**self**). It does this by allocating a new ToDoItem and initializing it with instance variables held by **self**. Copying is often implemented for *value* objects—objects that represent attributes such as numbers, dates, and to-do items.

Copies of objects can be either deep or shallow. In deep copies (like ToDoItem's) every copied instance variable is an independent replicate, including the values referenced by pointers. In shallow copies, pointers are copied but the referenced objects are the same. For more on this topic, see the description of the NSCopying protocol in the Foundation reference documentation.

The next method you'll implement—**description**—assists you and other developers in debugging the To Do application with **gdb**. When you enter the **po** (print object) command in **gdb** with a ToDoItem as the argument, this **description** method is invoked and essential debugging information is printed.

1 **Have the object describe itself during debugging.**

Implement the **description** method.

```
- (NSString *)description
{
    NSString *desc = [NSString stringWithFormat:@"%@\n\tName:
        %@\n\tDate: %@\n\tNotes: %@\n\tCompleted: %@\n\tSecs Until Due:
        %d\n\tSecs Until Notif: %d",
        [super description],
        [self itemName],
        [self day],
        [self notes],
        (([self itemStatus]==complete)?@"Yes":@"No"),
        [self secsUntilDue],
        [self secsUntilNotif]];

    return (desc);
}
```

1 **Implement ToDoItem's initialization and deallocation methods.**

Here are some things to remember as you implement **initWithName:andDate:** and **dealloc**:

- If the first argument of **initWithName:andDate:** (the item name) is not a valid string, return **nil**. If the second argument (the date) is **nil**, set the related instance variable to some reasonable value (such as today's date). Also, be sure to invoke **super**'s **init** method.
- The instance variables to initialize are **day**, **itemName**, **notes**, and **itemStatus** (to "incomplete").
- In **dealloc**, release those object instance variables initialized in **initWithName:andDate:** plus any object instance variables that were initialized later. Also invalidate any timer before you release it.

1 **Implement ToDoItem's archiving and unarchiving methods.**

When you implement **encodeWithCoder:** and **initWithCoder:**, keep the following in mind:

- Encode and decode instance variables in the same order.
- Copy the object instance variables after you decode them.
- You don't need to archive the **itemTimer** instance variable since timers are re-set when a document is opened.

The final step in creating the ToDoItem class is to implement the functions that furnish "value-added" behavior.

1 **Implement ToDoItem's time-conversion functions.**

```
long ConvertTimeToSeconds(int hr, int min, BOOL flag)          A
{
    if (flag) { /* PM */
        if (hr >= 1 && hr < 12)
            hr += 12;
    } else {
        if (hr == 12)
            hr = 0;
    }
    return ((hr * hrInSecs) + (min * minInSecs));
}

BOOL ConvertSecondsToTime(long secs, int *hour, int *minute)   B
{
    int hr=0;
    BOOL pm=NO;

    if (secs) {
        hr = secs / hrInSecs;
        if (hr > 12) {
            *hour = (hr -= 12);
            pm = YES;
        } else {
            pm = NO;
            if (hr == 0)
                hr = 12;
            *hour = hr;
        }
        *minute = ((secs%hrInSecs) / minInSecs);
    }
    return pm;
}
```

A This expression, as well as others in these two methods, uses the **enum** constants for time-values-as seconds that you defined earlier.

B The **ConvertSecondsToTime()** function uses indirection as a means for returning multiple values and directly returns a Boolean to indicate AM or PM.

Breaktime!

Take a break from coding and build the project as it now stands. Go get a coffee, soda, or other beverage while the project is building. When you return, fix any errors that have insinuated themselves into the code. You can stop and build at anytime— a good thing to do because it will help you locate mistakes more easily.

Only When Needed: Dynamically Loading Resources and Code

As any developer knows well, performance is a key consideration in program design. One factor is the timing of resource allocation. If an application loads all code and resources that it *might* use when it starts up, it will probably be a sluggish, bloated application—and one that takes awhile to launch.

You can strategically store the resources of an application (including user-interface objects) in several nib files. You can also put code that might be used among one or more *loadable bundles*. When the application needs a resource or piece of code, it loads the nib file or loadable bundle that contains it. This technique of deferred allocation benefits an application greatly. By conserving memory, it improves program efficiency. It also speeds up the time it takes to launch the application.

Auxiliary Nib Files

When more sophisticated applications start up, they load only a minimum of resources in the main nib file—the application's menus and perhaps a window. They display other windows (and load other nib files) only when users request it or when conditions warrant it.

Nib files other than an application's main nib file are sometimes called *auxiliary nib files*. There are two general types of auxiliary nib files: special-use and document.

Special-use nib files contain objects (and other resources) that *might* be used in the normal operation of the application. Examples of special-use nib files are those containing inspector panels and Info (or About) panels.

Document nib files contain objects that represent some repeatable entity, such as a word-processor document. A document nib file is a template for documents: it contains the UI objects and other resources needed to make a document.

The Owner of an Auxiliary Nib File

The object that loads a nib file is usually the object that owns it. A nib file's owner must be external to the file. Objects unarchived from the nib file communicate with other objects in the application only through the owner.

In Interface Builder, the File's Owner icon represents this external object. The File's Owner is typically the application controller for special-use nib files, and the document controller for document nib files. The File's Owner object is not really appearing twice; it's created in your application and referenced in your nib file.

The File's Owner object dynamically loads a nib file and makes itself the owner of that file by sending **loadNibNamed:owner**: to NSBundle, specifying **self** as the second argument.

NSBundle and Bundles

A bundle is a location in the file system (a folder) that stores code and the resources that go with that code, including images and archived objects. A bundle is also identified as an instance of NSBundle; this object makes the contents of the bundle available to other objects that request it.

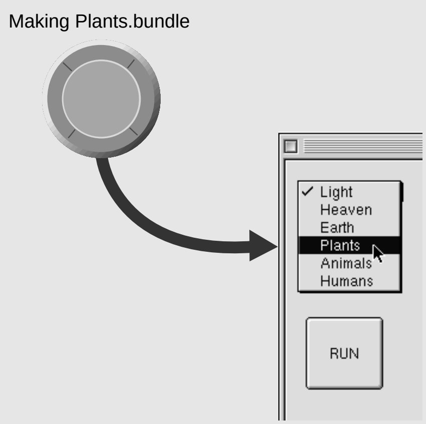
The generic notion of bundles is pervasive throughout Rhapsody. Applications are bundles, as are frameworks and palettes. Every application has at least one bundle—its main bundle—which is the ".app" directory (or *application wrapper*) where its executable file is located. This file is loaded into memory when the application is launched.

Loadable Bundles

You can organize an application into any number of other bundles in addition to the main bundle and the bundles of linked-in frameworks. Although these loadable bundles usually reside inside the application wrapper, they can be anywhere in the file system. Project Builder allows you to build Loadable Bundle projects.

Loadable bundles differ from nib files in that they don't require you to use Interface Builder to build them. Instead of containing mostly archived objects, they usually contain mostly code. Loadable bundles are especially useful for incorporating extra behavior into an application upon demand. An economic-forecast application, for example, might load a bundle containing the code defining an economic model, but only when users request that model. You could also use loadable bundles to integrate "plug and play" components into an existing framework.

Loadable bundles usually have an extension of ".bundle" (although that's a convention, not a requirement). Each loadable bundle must have a principal class that mediates between bundle objects and external objects.



Extending an Application Kit Class: An Example

The calendar on To Do's interface is an instance of a custom subclass of NSMatrix. CalendarMatrix dynamically updates itself as users select new months, notifies a delegate when users select a day, and reflects the current day (today) and the current selection by setting button-cell attributes.

Creating a subclass of a class that is farther down the inheritance tree poses more of a challenge for a developer than a simple subclass of NSObject. A class such as NSMatrix is more specialized than NSObject and carries with it more baggage: It inherits from NSResponder, NSView, and NSControl, all fairly complex Application Kit classes. And since CalendarMatrix inherits from NSView, it appears on the user interface; it is an example of a view object in the Model-View-Controller paradigm, and as such it is highly reusable.

Why NSMatrix as Superclass?

When you select a specialized superclass as the basis for your subclass, it is important to consider what your requirements are and to understand what the superclass has to offer. To Do's dynamic calendar should:

- Arrange numbers (days) sequentially in rows and columns.
- Respond to and communicate selections of days.
- Understand dates.
- Enable navigation between months.

If you then started to peruse the reference documentation on Application Kit classes, and looked at the section on NSMatrix, you'd read this:

NSMatrix is a class used for creating groups of NSCells that work together in various ways. It includes methods for arranging NSCells in rows and columns.... An NSMatrix adds to NSControl's target/action paradigm by allowing a separate target and action for each of its NSCells in addition to its own target and action.

So NSMatrix has an inherent capability for the first of the requirements listed above, and part of the second (responding to selections). Our CalendarMatrix subclass thus does not need to alter anything in its superclass. It just needs to supplement NSMatrix with additional data and behavior so it can understand dates (and update itself appropriately), navigate between months, and notify a delegate that a selection was made.

Composing the Interface

1 **Define the CalendarMatrix class in Interface Builder.**

From Project Builder, open **ToDo.nib**.

In Interface Builder, choose File ► New Module ► New Empty to create a new nib file.

Save the nib file as **ToDoDoc.nib**.

Respond Yes when asked if you want the nib file added to the project.

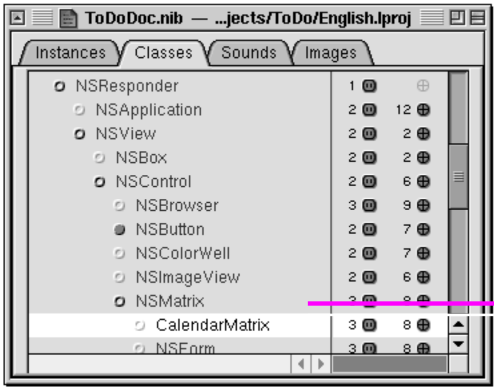
In the Classes display of the nib file window, select NSMatrix.

Choose Subclass from the Classes menu.

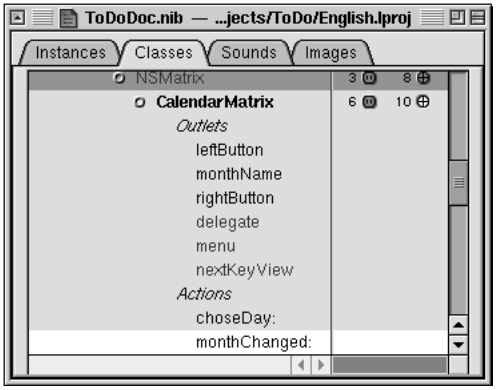
Name the new class "CalendarMatrix".

Select the new class.

Add the outlets and actions shown in the example at right.



Locate NSMatrix several levels down in the class hierarchy.



Outlets and actions already defined by the superclass (or by its superclasses) appear in gray text. Add the outlets and actions shown in black text.

When you created subclasses of NSObject in the previous two tutorials, the next step was to instantiate the subclass. Because CalendarMatrix is a view (that is, it inherits from NSView), the procedure for generating an instance for making connections is different.

1 Put a custom NSView object (CalendarMatrix) on the user interface.

Drag a window from the Windows palette.

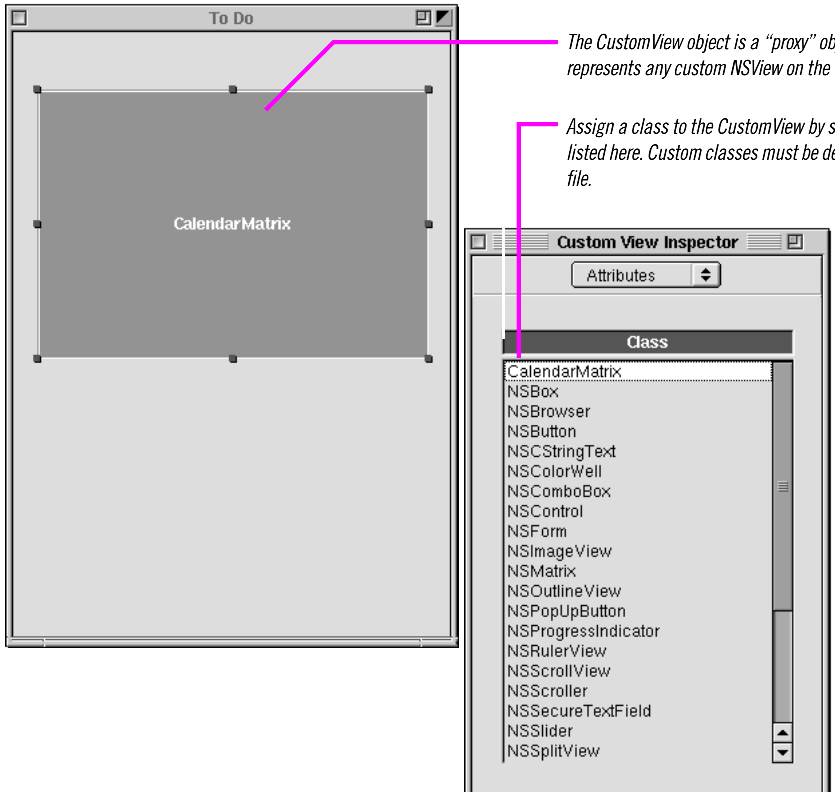
Resize the window, using the example at right as a guide.

Turn off the window's resize bar.

Drag a CustomView from the Views palette onto the window.

Resize and position the CustomView, using the example at right as a guide.

In the Attributes display of the inspector, select CalendarMatrix from the list of available classes.



The CustomView object is a "proxy" object that represents any custom NSView on the interface.

Assign a class to the CustomView by selecting a class listed here. Custom classes must be defined in the nib file.

The selection of the class for the CustomView creates an instance of it that you can connect to other objects in the nib file. Now put the controls and fields associated with CalendarMatrix on the window.

1 **Put the objects related to CalendarMatrix on the window.**

Drag a label object for the month-year from the Views palette and put it over the CalendarMatrix.

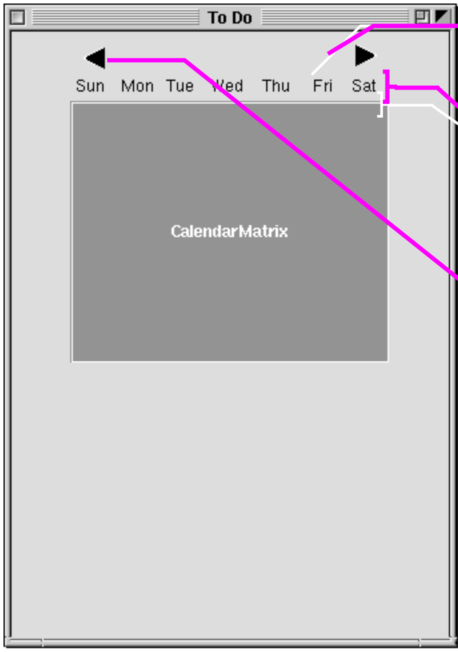
Make a small label for each day of the week.

Drag a button onto the interface and set its attributes to unbordered and image only.

Drag left_arrow.tiff from the ToDo project in **/System/Developer/Examples/AppKit** and drop it over the button.

To the attention panel that asks "Insert image left_arrow in project?" click Yes.

Repeat the same button procedure for right_arrow.tiff.



This empty label will display the month and year. Initialize it by typing "September 9999" (the longest possible string). Set the text to Helvetica 18 points, center it, and then delete the text.

Type the days of the week as individual labels, arrange them as a row, and then center the labels over the columns of days. (This latter task could take some trial and error.)

To have the button surround the image as tightly as possible, select the button and choose Format ► Size ► Size To Fit.

Next connect CalendarMatrix to its satellite objects.

1 **Connect CalendarMatrix to its outlet and to the controls sending action messages.**

Name	Connection	Type
monthName	From CalendarMatrix to the label field above it	outlet
leftButton	From CalendarMatrix to the left-pointing arrow	outlet
rightButton	From CalendarMatrix to the right-pointing arrow	outlet
monthChanged:	From both arrows to CalendarMatrix	action

1 **Finish up in Interface Builder.**

Save ToDoDoc.nib.

Select CalendarMatrix and in the Classes display and choose Create Files from the Operations pull-down menu.

Confirm that you want the source-code files added to the project.

You might have noticed that there's an action message left unconnected: **choseDay:**. Because it is impossible in Interface Builder to connect an object with itself, you will make this connection programmatically.

1 **Add declarations to the header file CalendarMatrix.h.**

(Existing declarations are indicated by ellipsis.)

```
@interface CalendarMatrix : NSMatrix
{
     /* ... */
    NSCalendarDate *selectedDay;
    short startOffset;                                              A
}
    /* ... */
- (void)refreshCalendar;
- (id)initWithFrame:(NSRect)frameRect;
- (void)dealloc;
- (void)setSelectedDay:(NSCalendarDate *)newDay;
- (NSCalendarDate *)selectedDay;
@end

@interface NSObject(CalendarMatrixDelegate)                         B
 - (void)calendarMatrix:(CalendarMatrix *)object
       didChangeToDate:(NSDate *)date;
 - (void)calendarMatrix:(CalendarMatrix *)object
       didChangeToMonth:(int)month year:(int)year;
@end
```

There are a couple of interesting things to note about these declarations:

A The cells in CalendarMatrix are sequentially ordered by tag number, left to right, going downward. **startOffset** marks the cell (by its tag) on which the first day of the month falls.

B CalendarMatrixDelegate is a category on NSObject that declares the methods to be implemented by the delegate. This technique creates what is called an *informal protocol*, which is commonly used for delegation methods.

Defining the New Behavior

1 **Implement CalendarMatrix's initialization methods.**

Select **CalendarMatrix.m** in the project browser.

Write the implementation of **initWithFrame:** (at right).

Implement **dealloc**.

```
- (id)initWithFrame:(NSRect)frameRect
{
    int i, j, cnt=0;
    id cell = [[NSButtonCell alloc] initTextCell:@""];
    NSCalendarDate *now = [NSCalendarDate date];                    A

    [super initWithFrame:frameRect                                  B
                    mode:NSRadioModeMatrix
               prototype:cell
            numberOfRows:6
         numberOfColumns:7];
    // set cell tags                                                C
    for (i=0; i<6; i++) {
        for (j=0; j<7; j++) {
            [[self cellAtRow:i column:j] setTag:cnt++];
        }
    }
    [cell release];
    selectedDay = [[NSCalendarDate dateWithYear:[now yearOfCommonEra]
        month:[now monthOfYear]                                     D
          day:[now dayOfMonth]
         hour:0 minute:0 second:0
     timeZone:[NSTimeZone localTimeZone]] copy];

    return self;
}
```

The **initWithFrame:** method is an initializer of NSMatrix, NSControl and NSView.

A This invocation of **date**, a class method declared by NSDate, returns the current date ("today") as an NSCalendarDate. (NSCalendarDate is a subclass of NSDate.)

B This message to **super** (NSMatrix) sets the physical and cell dimensions of the matrix, identifies the type of cell using a prototype (an NSButtonCell), and specifies the general behavior of the matrix: radio mode, which means that only one button can be selected at any time.

C Set the tag number of each cell sequentially left to right and down. Tags are the mechanism by which CalendarMatrix sets and retrieves the day numbers of cells.

D This NSCalendarDate class method initializes the **selectedDay** instance variable to midnight of the current day, using the year, month, and day elements of the current date. The **localTimeZone** message obtains an NSTimeZone object with a suitable offset from Greenwich Mean Time.

Implement **awakeFromNib** as shown at right.

```
- (void)awakeFromNib
{
    [monthName setAlignment:NSCenterTextAlignment];
    [self setTarget:self];
    [self setAction:@selector(choseDay:)];
    [self setAutosizesCells:YES];
    [self refreshCalendar];
}
```

The **awakeFromNib** method performs additional initializations (some of which could just have easily been done in **initWithFrame:**). Most importantly, it sets **self** as its own target object and specifies an action method for this target, **choseDay:**, something that couldn't be done in Interface Builder. Other methods to note:

- **•setAutosizesCells:** causes the matrix to resize its cells on every redraw.
- **•refreshCalendar** (which you'll write next) updates the calendar.

The **refreshCalendar** method is fairly long and complex—it is the workhorse of the class—so you'll approach it in sections.

Dates and Times in Rhapsody

In Rhapsody you represent dates and times as objects that inherit from NSDate. The major advantage of dates and times as objects is common to all objects that represent basic values: they yield functionality that, although commonly found in most operating systems, is not tied to the internals of any particular operating-system.

NSDates hold dates and times as values of type NSTimeInterval and express these values as seconds. The NSTimeInterval type makes possible a wide and fine-grained range of date and time values, giving accuracy within milliseconds for dates 10,000 years apart.

NSDate and its subclasses compute time as seconds relative to an absolute reference date (the first instant of January 1, 2001). NSDate converts all date and time representations to and from NSTimeInterval values that are relative to this reference date.

NSDate provides methods for obtaining NSDate objects (including **date**, which returns the current date and time as an NSDate), for comparing dates, for computing relative time values, and for representing dates as strings.

The NSCalendarDate class, which inherits from NSDate, generates objects that represent dates conforming to western calendrical systems. NSCalendarDate objects also adjust the representations of dates to reflect their associated time zones. Because of this, you can track an NSCalendarDate object across different time zones. You can also present date information from time-zone viewpoints other than the one for the current locale.

Each NSCalendarDate object also has a calendar format string bound to it. This format string contains date-conversion specifiers that are very similar to those used in the standard C library function **strftime()**. NSCalendarDate can interpret user-entered dates that conform to this format string.

NSCalendarDate has methods for creating NSCalendarDate objects from formatted strings and from component time values (such as minutes, hours, day of week, and year). It also supplements NSDate with methods for accessing component time values and for representing dates in various formats, locales, and time zones.

1 **Implement the code that updates the calendar.**

Initialize the **MonthDays[]** array and write the **isLeap()** macro.

Determine the day of the week at the start of the month and the number of days in the month.

```
static short MonthDays[] =
    {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
#define isLeap(year) (((((year) % 4) == 0 && (((year) % 100) != 0))
    || ((year) % 400) == 0))
```

```
- (void)refreshCalendar
{
    NSCalendarDate *firstOfMonth, *selDate = [self selectedDay],
        *now = [NSCalendarDate date];
    int i, j, currentMonth = [selDate monthOfYear];
    unsigned int currentYear = [selDate yearOfCommonEra];
    short daysInMonth;
    id cell;

    firstOfMonth = [NSCalendarDate dateWithYear:currentYear        A
                   month:currentMonth
                     day:1 hour:0 minute:0 second:0
                timeZone:[NSTimeZone localTimeZone]];
    [monthName setStringValue:[firstOfMonth                        B
        descriptionWithCalendarFormat:@"%B %Y"]];
    daysInMonth = MonthDays[currentMonth-1]+1;                     C
    /* correct Feb for leap year */
    if ((currentMonth == 2) && (isLeap(currentYear))) daysInMonth++;
    startOffset = [firstOfMonth dayOfWeek];                        D
```

Before it can start writing day numbers to the calendar for a given month, CalendarMatrix must know what cell to start with and how many cells to fill with numbers. The **refreshCalendar** method begins by calculating these values.

A Creates an NSCalendarDate for the first day of the currently selected month and year (computed from the **selectedDay** instance variable).

B Writes the month and year (for example, “February 1997”) to the label above the calendar.

C Gets from the **MonthDays** static array the number of days for that month; if the month is February and it is a leap year, this number is adjusted.

D Gets the day of the week for the first day of the month and stores this in the **startOffset** instance variable.

Write the **refreshCalendar** code that writes day numbers to the cells and sets cell attributes.

```
    for (i=0; i<startOffset; i++) {
        cell = [self cellWithTag:i];
        [cell setBordered:NO];
        [cell setEnabled:NO];
        [cell setTitle:@""];
        [cell setCellAttribute:NSCellHighlighted to:NO];
    }
    for (j=1; j < daysInMonth; i++, j++) {
        cell = [self cellWithTag:i];
        [cell setBordered:YES];
        [cell setEnabled:YES];
        [cell setFont:[NSFont systemFontOfSize:12]];
        [cell setTitle:[NSString stringWithFormat:@"%d", j]];
        [cell setCellAttribute:NSCellHighlighted to:NO];
    }
    for (;i<42;i++) {
        cell = [self cellWithTag:i];
        [cell setBordered:NO];
        [cell setEnabled:NO];
        [cell setTitle:@""];
        [cell setCellAttribute:NSCellHighlighted to:NO];
    }
```

The first and third for-loops in this section of code clear the leading and trailing cells that aren't part of the month's days. Because the current day is indicated by highlighting, they also turn off the highlighted attribute. The second for-loop writes the day numbers of the month, starting at **startOffset** and continuing until **daysInMonth**, and resets the font (since the selected day is in bold face) and other cell attributes.

Complete the **refreshCalendar** method implementation by resetting the "today" cell attribute.

```
       if ((currentYear == [now yearOfCommonEra])
           && (currentMonth == [now monthOfYear])) {
        [[self cellWithTag:([now dayOfMonth]+startOffset)-1]
            setCellAttribute:NSCellHighlighted to:YES];
        [[self cellWithTag:([now dayOfMonth]+startOffset)-1]
            setHighlightsBy:NSMomentaryChangeButton];
       }
}
```

This final section of **refreshCalendar** determines if the newly selected month and year are the same as today's, and if so highlights the cell corresponding to today.

1 **Specify the behavior that occurs when users select a new month.**

Implement the **monthChanged:** action method.

```
- (void)monthChanged:sender
{
    NSCalendarDate *thisDate = [self selectedDay];
    int currentYear = [thisDate yearOfCommonEra];
    unsigned int currentMonth = [thisDate monthOfYear];

    if (sender == rightButton) {                                    A
        if (currentMonth == 12) {
            currentMonth = 1;
            currentYear++;
        } else {
            currentMonth++;
        }
    } else {
        if (currentMonth == 1) {
            currentMonth = 12;
            currentYear--;
        } else {
            currentMonth--;
        }
    }                                                               B
    [self setSelectedDay:[NSCalendarDate dateWithYear:currentYear
                   month:currentMonth
                     day:1 hour:0 minute:0 second:0
                timeZone:[NSTimeZone localTimeZone]]];
    [self refreshCalendar];
    [[self delegate] calendarMatrix:self                            C
        didChangeToMonth:currentMonth year:currentYear];
}
```

The arrow buttons above CalendarMatrix send it the **monthChanged:** message when they are clicked. This method causes the calendar to go forward or backward a month.

A Determines which button is sending the message, then increments or decrements the month accordingly. If it goes past the end or beginning of the year, it increments or decrements the year and adjusts the month.

B Resets the **selectedDay** instance variable with the new month (and perhaps year) numbers and invokes **refreshCalendar** to display the new month.

C Sends the **calendarMatrix:didChangeToMonth:year:** message to its delegate (which in this application, as you'll soon see, is a ToDoDoc controller object).

1 **Specify the behavior that occurs when users select a day on the calendar.**

Implement the **choseDay:** action method.

```
- (void)choseDay:sender
{
    NSCalendarDate *selDate, *thisDate = [self selectedDay];
    unsigned int selDay = [[self selectedCell] tag]-startOffset+1;        A

    selDate = [NSCalendarDate dateWithYear:[thisDate yearOfCommonEra]     B
                                 month:[thisDate monthOfYear]
                                   day:selDay
                                  hour:0
                                minute:0
                                second:0
                              timeZone:[NSTimeZone localTimeZone]];
    [[self cellWithTag:[thisDate dayOfMonth]+startOffset-1]               C
        setFont:[NSFont systemFontOfSize:12]];
    [[self cellWithTag:selDay+startOffset-1] setFont:
        [NSFont boldSystemFontOfSize:12]];

    [self setSelectedDay:selDate];                                        D
    [[self delegate] calendarMatrix:self didChangeToDate:selDate];
}
```

This method is invoked when users click a day of the calendar.

A Gets the tag number of the selected cell and subtracts the offset from it (plus one to adjust for zero-based indexing) to find the number of the selected day.

B Derives an NSCalendarDate that represents the selected date.

C Sets the font of the previously selected cell to the normal system font (removing the bold attribute) and puts the number of the currently selected cell in bold face.

D Sets the **selectedDay** instance variable to the new date and sends the **calendarMatrix:didChangeToDate:** message to the delegate.

1 **Implement accessor methods for the selectedDay instance variable.**

You are finished with CalendarMatrix. If you loaded **ToDoDoc.nib** right now, the calendar would work, up to a point. If you clicked the arrow buttons, CalendarMatrix would display the next or previous months. The days of the month would be properly set out on the window, and the current day would be highlighted.

But not much else would happen. That's because CalendarMatrix has not yet been hooked up to its delegate.

The Basics of a Multi-Document Application

A multi-document application, as described on page 139, has at least one application controller and a document controller for each document opened. The application controller also responds to user commands relating to documents and either creates, opens, closes, or saves a document.

1 Customize the application's menu.

In Interface Builder:

Open **ToDo.nib**.

Rename the File menu "Document"

Remove unused menu items.

Put a generic menu item ("Item") in the Document menu and rename it "Inspector".

Make sure separator lines are above and below the Inspector command.

Give the Inspector command the key equivalent of Command-i.

Remove all file- or document-related commands from the File menu except for the ones shown here.

The three dots after Inspector indicate that the command displays a modal panel.

Interface Builder gives each new Rhapsody application the following default menus: Apple, File, Edit, Window, Services, and Help. The Windows menu lists windows of the application that are open and allows you to bring them to the top window tier. The Services menu lists other Rhapsody applications on a system and allows you to pass data to, or get data from, those applications.

Note: Disable the Preferences command in the Apple menu. This tutorial does not specifically cover Preferences panels, but it does give you enough information so that you can implement Preferences on your own.

The Structure of Multi-Document Applications

From a user's perspective, a document is a unique body of information usually contained by its own window. Users can create an unlimited number of documents and save each to a file. Common documents are word-processing documents and spreadsheets.

From a programming perspective, a document comprises the objects and resources unarchived from an auxiliary nib file and the controller object that loads and manages these things. This *document controller* is the owner of the auxiliary nib file containing the document interface and related resources.To manage a document, the document controller makes itself the delegate of its window and its "content" objects. It tracks edited status, handles window-close events, and responds to other conditions.

When users choose the New (or equivalent) command, a method is invoked in the application's controller object. In this method, the application controller creates a document-controller object, which loads the document nib file in the course of initializing itself. A document thus remains independent of the application's "core" objects, storing state data in the document controller. If the application needs information about a document's state, it can query the document controller.

When users choose the Save command, the application displays a Save panel and enables users to save the document in the file system. When users choose the Open command, the application displays an Open panel, allowing users to select a document file and open it.

Document Management Techniques

When you make the application controller the delegate of the application (NSApp) and the document controller the delegate of each document window, they can receive messages sent at critical moments of a running application.

These moments include the closure of windows (**windowShouldClose**:), window selection (**windowDidResignMain**:), application start-up (**applicationWillFinishLaunching**:) and application termination (**applicationShouldTerminate**:). In the methods handling these messages, the controllers can then do the appropriate thing, such as saving a document's data or displaying an empty document.

Several NSViews also have delegation messages that facilitate document management, particularly text fields, forms, and other controls with editable text (**controlText**...) and NSText objects (**text**...). One important such message is **textDidChange**: (or **controlTextDidChange**:), which signals that the document's textual content was modified. In responding to this message, controllers can mark a document window as having unsaved data with the **setDocumentEdited**: message (the close button of edited documents is a "broken" X). Later, they can determine whether the document needs to be saved by sending **isDocumentEdited** to the window.

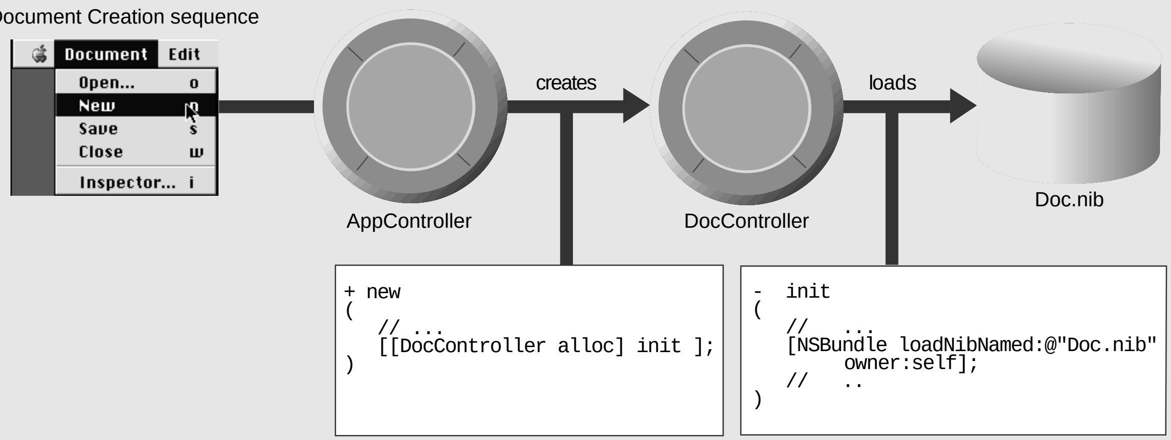
Document controllers often need to communicate with the application controller or other objects in the application. One way to do this is by posting notifications. Another way is to use the key relationships within the core program framework (see page 152) to find the other object (assuming it's a delegate of an Application Kit object). For example, the application controller can send the following message to locate the current document controller:

```
[[NSApp mainWindow] delegate]
```

The document controller can find the application controller with:

```
[NSApp delegate]
```

Document Creation sequence



Defining the Controller and User Interfaces

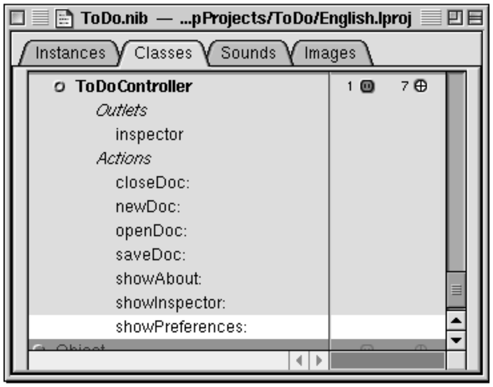
Begin by defining in Interface Builder the object controlling the To Do application.

1 **Define the application-controller class.**

Create ToDoController as a subclass of NSObject.

Add the outlet and actions shown in the example.

Make the action connections from the appropriate File menu commands.



Now that you've defined the application-controller class, define the document-controller class, ToDoDoc. Remember, since the ToDoDoc controller must own the nib file containing the document, it must be external to it; although it is referenced in the main nib file (**ToDo.nib**) and in **ToDoDoc.nib**, it's instantiated before its nib file is loaded.

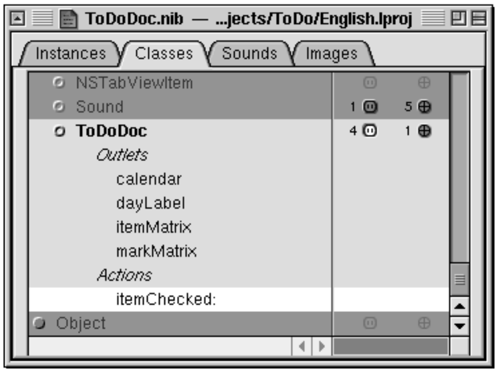
1 **Define the document-controller class.**

Create ToDoDoc as a subclass of NSObject.

Add to the class the outlets and action listed at right.

Instantiate ToDoController and ToDoDoc.

Save **ToDo.nib**.



Now add the remaining objects to the document interface.

1 Complete the document interface.

Open **ToDoDoc.nib**.

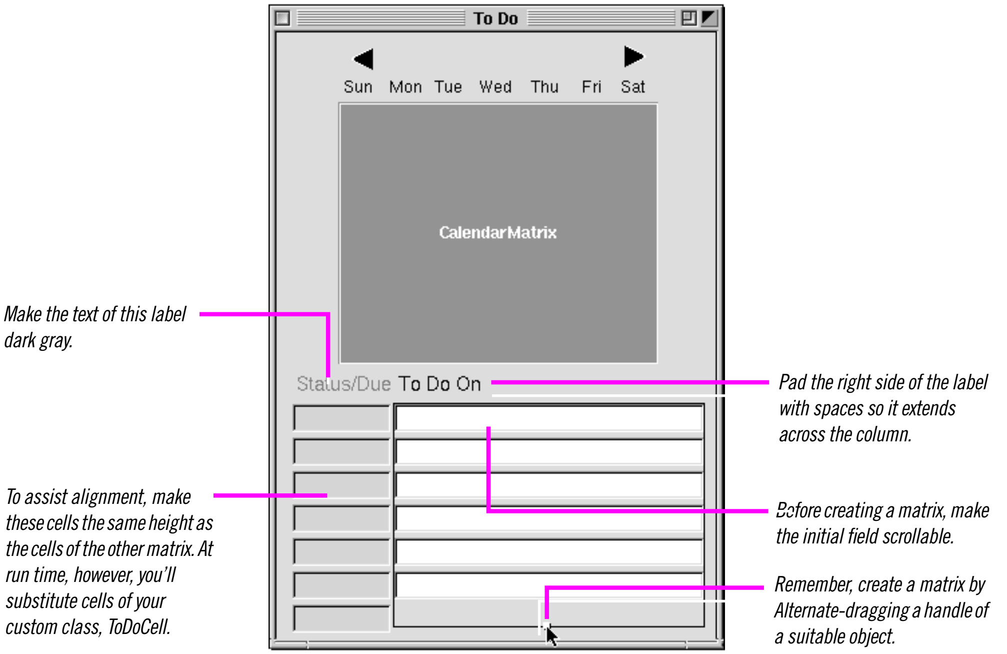
Add the matrices of text fields.

Add the labels above the matrices.

Make the labels 14 points in the user's application font.

Make the item text 12 points in the user's application font.

Save **ToDoDoc.nib**.



1 Connect the outlets and actions of ToDoDoc.

Select File's Owner in the Instances display of **ToDoDoc.nib**.

Choose ToDoDoc from the list of classes in the Attributes display of the inspector.

Make the connections described in the table at right.

Name	Connection	Type
calendar	From File's Owner to the CalendarMatrix object	outlet
dayLabel	From File's Owner to label "To Do on"	outlet
itemMatrix	From File's Owner (ToDoDoc) to matrix of long text fields	outlet
markMatrix	From File's Owner to matrix of short text fields	outlet
itemChecked:	From matrix of short text fields to File's Owner	action

Text fields in a matrix, just like a form's cells, are connected for inter-field tabbing when you create the matrix. But you must also connect ToDoDoc and ToDoController to the delegate outlets of other objects in the application—this step is critical to the multi-document design.

Connect ToDoDoc and ToDoController to other objects as their delegates.

Name	Connection
textDelegate	From the CalendarMatrix object to File's Owner (ToDoDoc)
delegate	From the document window's title bar (or the window icon in the nib file window) to File's Owner (ToDoDoc)
delegate	In **ToDo.nib**, from File's Owner (NSApp) to the ToDoController instance

The ToDoDoc class needs supplemental data and behavior to get the multi-document mechanism working right.

1 **Create source-code files for ToDoDoc and ToDoController.**

In Project Builder:

1 **Add declarations of methods and instance variables to the ToDoDoc class.**

Select **ToDoDoc**.h in the project browser.

Add the declarations at right.

(Ellipses indicate existing declarations.)

```
@interface ToDoDoc:NSObject
{
    /* ... */
    NSMutableDictionary *activeDays;
    NSMutableArray *currentItems;
}
/* ... */
- (NSMutableArray *)currentItems;
- (void)setCurrentItems:(NSMutableArray *)newItems;
- (NSMatrix *)itemMatrix;
- (NSMatrix *)markMatrix;
- (NSMutableDictionary *)activeDays;
- (void)saveDoc;
- (id)initWithFile:(NSString *)aFile;
- (void)dealloc;
- (void)activateDoc;
- (void)selectItem:(int)item;
@end
```

The **activeDays** and **currentItems** instance variables hold the collection objects that store and organize the data of the application. (You'll deal with these instance variables much more in the next section of this tutorial.) Many of the methods declared are accessor methods that set or return these instance variables or one of the matrices of the document.

Creating, Opening, Saving, and Closing Documents

You'll be switching between **ToDoDoc.m** and **ToDoController.m** in the next few tasks. The intent is not to confuse, but to show the close interaction between these two classes.

1 **Write the code that creates documents.**

Select ToDoController.m in the project browser.

Implement ToDoController's **newDoc:** method.

```
- (void)newDoc:(id)sender
{
    id currentDoc = [[ToDoDoc alloc] initWithFile:nil];
    [currentDoc activateDoc];
}
```

The **newDoc:** method is invoked when the user chooses New from the Document menu. The method allocates and initializes an instance of the document controller, ToDoDoc, thereby creating a document. (See the implementation of **initWithFile:** on the following page to see what happens in this process.) It then updates the document interface by invoking **activateDoc**.

Select ToDoDoc.m in the project browser.

Implement ToDoDoc's **initWithFile:** method.

```
- initWithFile:(NSString *)aFile
{
    NSEnumerator *dayenum;
    NSDate *itemDate;

    [super init];
    if (aFile) {                                                    A
        activeDays = [NSUnarchiver unarchiveObjectWithFile:aFile];
        if (activeDays)
            activeDays = [activeDays retain];
        else
            NSRunAlertPanel(@"To Do", @"Couldn't unarchive file %@",
                nil, nil, nil, aFile);
    } else {                                                        B
        activeDays = [[NSMutableDictionary alloc] init];
        [self setCurrentItems:nil];
    }
    if (![NSBundle loadNibNamed:@"ToDoDoc.nib" owner:self] )        C
          return nil;
    if (aFile)                                                      D
        [[itemMatrix window] setTitleWithRepresentedFilename:aFile];
    else
        [[itemMatrix window] setTitle:@"UNTITLED"];
    [[itemMatrix window] makeKeyAndOrderFront:self];
    return self;
}
```

This method, which initializes and loads the document, has the following steps:

A Restores the document's archived objects if the **aFile** argument is the pathname of a file containing the archived objects (that is, the document is opened). If objects are unarchived, it retains the **activeDays** dictionary; otherwise it displays an attention panel.

B Initializes the **activeDays** and **currentItems** instance variables. An **aFile** argument with a **nil** value indicates that the user is requesting a new document.

C Loads the nib file containing the document interface, specifying **self** as owner.

D Sets the title of the window; this is either the file name on the left of the title bar and the pathname on the right, or "UNTITLED" if the document is new.

Note the **[itemMatrix window]** message nested in the last message. Every object that inherits from NSView "knows" its window and will return that NSWindow object if you send it a **window** message.

1 **Implement the document-opening method.**

Select **ToDoController.m** in the project browser.

Write the code for **openDoc:**.

```
- (void)openDoc:(id)sender
{
    int result;
    NSString *selected, *startDir;
    NSArray *fileTypes = [NSArray arrayWithObject:@"td"];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];          A

    [oPanel setAllowsMultipleSelection:YES];
    if ([[[NSApp keyWindow] delegate] isKindOfClass:[ToDoDoc class]])
        startDir = [[[NSApp keyWindow] representedFilename]  B
            stringByDeletingLastPathComponent];
    else
        startDir = NSHomeDirectory();
    result = [oPanel runModalForDirectory:startDir file:nil  C
        types:fileTypes];
    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        int i, count = [filesToOpen count];
        for (i=0; i<count; i++) {                            D
              NSString *aFile = [filesToOpen objectAtIndex:i];
              id currentDoc = [[ToDoDoc alloc] initWithFile:aFile];
              [currentDoc activateDoc];
        }
    }
}
```

The **openDoc:** method displays the modal Open panel, gets the user's response (which can be multiple selections) and opens the file (or files) selected.

A Creates or gets the NSOpenPanel instance (an instance shared among objects of an application). The previous message specifies the file types (that is, the extensions) of the files that will appear in the Open panel browser. The next message enables selection of multiple files in the panel's browser.

B Sets the directory at which the NSOpenPanel starts displaying files either to the directory of any document window that is currently key or, if there is none, to the user's home directory.

C Runs the NSOpenPanel and obtains the key clicked.

D If the key is NSOKButton, cycles through the selected files and, for each, creates a document by allocating and initializing a ToDoDoc instance, passing in a file name.

The methods invoked by the Document menu's Close and Save commands both simply send a message to another object. How they locate these objects exemplify important techniques using the core program framework.

1 **Write the code that closes documents.**

In **ToDoController.m,** implement the **closeDoc:** method.

```
- (void)closeDoc:(id)sender
{
    [[NSApp mainWindow] performClose:self];
}
```

NSApp, the global NSApplication instance, keeps track of the application's windows, including their status. Because only one window can have main status, the **mainWindow** message returns that NSWindow object— which is, of course, the one the user chose the Close command for. The **closeDoc:** method sends **performClose:** to that window to simulate a mouse click in the window's close button. (See the following section, "Managing Documents Through Delegation," to learn how the document handles this user event.)

1 **Write the code that saves documents.**

In **ToDoController.m**, implement the **saveDoc:** method.

```
- (void)saveDoc:(id)sender
{
    id currentDoc = [[NSApp mainWindow] delegate];
    if (currentDoc)
        [currentDoc saveDoc];
}
```

As did **closeDoc:**, this method sends **mainWindow** to NSApp to get the main window, but then it sends **delegate** to the returned window to get its delegate, the ToDoDoc instance that is managing the document. It then sends the ToDoDoc-defined message **saveDoc** to this instance.

Note: You could implement **closeDoc:** and **saveDoc:** in the ToDoDoc class, but the ToDoController approach was chosen to make the division of responsibility clearer.

Select **ToDoDoc.m** in the project browser.

Implement the **saveDoc:** method.

```
- (void)saveDoc
{
    NSString *fn;

    if (![[[itemMatrix window] title] hasPrefix:@"UNTITLED"]) {
        fn = [[itemMatrix window] representedFilename];
    } else {
        int result;
        NSSavePanel *sPanel = [NSSavePanel savePanel];
        [sPanel setRequiredFileType:@"td"];
        result = [sPanel runModalForDirectory:NSHomeDirectory()
file:nil];
        if (result == NSOKButton) {
            fn = [sPanel filename];
            [[itemMatrix window] setTitleWithRepresentedFilename:fn];
        } else
            return;
    }

    if (![NSArchiver archiveRootObject:activeDays toFile:fn])
        NSRunAlertPanel(@"To Do", @"Couldn't archive file %@",
            nil, nil, nil, fn);
    else
        [[itemMatrix window] setDocumentEdited:NO];
}
```

ToDoDoc's **saveDoc** method complements ToDoController's **openDoc:** method in that it runs the modal Save panel for users.

A The **title** method returns the text that appears in the window's title bar. If the title doesn't begin with "UNTITLED" (what new document windows are initialized with), then a file name and directory location has already been chosen, and is stored as the **representedFilename**.

B If the window title begins with "UNTITLED" then the document needs to be saved under a user-specified file name and directory location. This part of the code creates or gets the shared NSSavePanel instance and sets the file type, which is the extension that's automatically appended. Then it runs the Save panel, specifying the user's home directory as the starting location.

C Archives the document under the chosen directory path and file name and, with the **setDocumentEdited:** message, puts an asterisk next to the window's title (more on this in the next section).

1 **Implement the accessor methods for ToDoController and ToDoDoc.**

Don't implement **setCurrentItems**: yet. This method does something special for the application that will be covered in "Managing ToDo's Data and Coordinating its Display" on page 158.

Coordinate Systems in Rhapsody

The screen's coordinate system is the basis for all other coordinate systems used for positioning, sizing, drawing, and event handling. You can think of the entire screen as occupying the upper-right quadrant of a two-dimensional coordinate grid. The other three quadrants, which are invisible to users, take negative values along their x-axis, their y-axis, or both axes. The screen's quadrant has its origin in the lower left corner; the positive x-axis extends horizontally to the right and the positive y-axis extends vertically upward. A unit along either axis is expressed as a pixel.

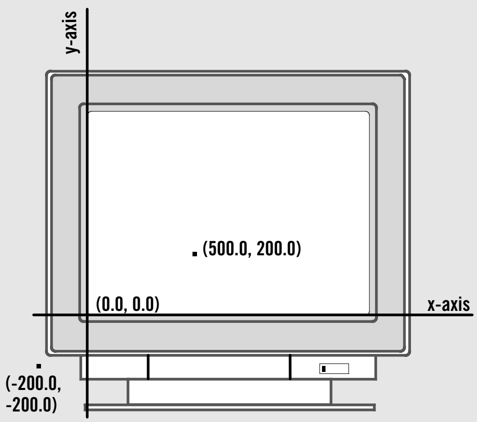
The screen coordinate system has just one function: to position windows on the screen. When your application creates a new window, it must specify the window's initial size and location in screen coordinates.You can "hide" windows by specifying their origin points well within one of the invisible quadrants. This technique is often used in off-screen rendering in buffered windows.

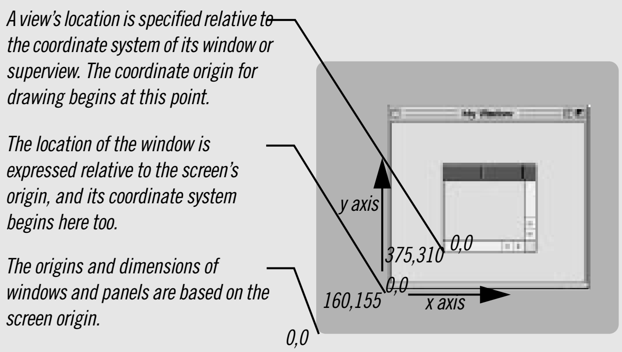
The reference coordinate system for a window is known as the *base* coordinate system. It differs from the screen coordinate system in only two ways:

- It applies only to a particular window; each window has its own base coordinate system.
- Its origin is at the lower left corner of the window, rather than the lower left corner of the screen. If the window moves, the origin and the entire coordinate system move with it.

For drawing, each NSView uses a coordinate system transformed from the base coordinate system or from the coordinate system of its superview. This coordinate system also has it origin point at the lower-left corner of the NSView, making it more convenient for drawing operations. NSView has several methods for converting between base and local coordinate systems. When you draw, coordinates are expressed in the application's *current* coordinate system, the system reflecting the last coordinate transformations to have taken place within the current window.

These coordinate systems are the inverse of several other operating systems, which put the origin point at the upper left of the window or screen and extend dimensions downward and to the right. NSView provides means for "flipping" coordinate systems to conform to those other systems.





Managing Documents Through Delegation

At certain points while an application is running you want to ensure that a document's data is preserved, that a document's edited status is tracked, or that the application otherwise does "the right thing" for a given circumstance. These events occur when users:

- Edit a document.
- Close a window.
- Launch the application.
- Quit the application by choosing the Exit command.
- Quit the application by closing the last window.
- Switch to another application or window.

Several classes of the Application Kit send messages to their delegates when these events occur, giving the delegate the opportunity to do the appropriate thing, whether that be saving a document to the file system or marking a document as edited.

1 **Mark a document as edited.**

Open **ToDoDoc.m**.

Implement the **controlTextDidChange:** method to mark the document.

```
- (void)controlTextDidChange:(NSNotification *)notif
{
    [[itemMatrix window] setDocumentEdited:YES];
}
```

When a control that contains editable text—such as a text field or a matrix of text fields—detects editing in a field, it posts the **controlTextDidChange:** notification which, like all notifications, is sent to the control's delegate as well as to all observers. The **setDocumentEdited:** message (with an argument of YES) inserts an asterisk to the right of the window's title, thereby marking it as "dirty" (containing modified, unsaved data).


```
[window setDocumentEdited:NO];
```


```
[window setDocumentEdited:YES];
```

Note: The object that, by notification, invokes the **controlTextDidChange:** method is **itemMatrix**, the matrix of to-do items (text fields). You will programmatically set ToDoDoc to be the delegate of this object later in this tutorial.

Assuming that you've completed certain steps (see "Opening Documents by Double-Clicking," below), when users select or double-click a To Do document icon in the Start menu, in Explorer, or elsewhere on the desktop, To Do will launch itself and open the document. But what happens when users simply launch the application, without specifying a document? Rhapsody applications have several alternatives (see side bar on page 155). To Do lends itself well to the user-defaults technique:

- At first, open an "UNTITLED" document.
- When the user saves a document, save the document path in user defaults.
- Thereafter open the last-saved document when the user launches To Do.

1 **Customize the launch behavior for your multi-document application.**

Initialize the ToDoController class.

```
+ (void)initialize
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSDictionary *regdom = [NSDictionary dictionaryWithObject:@"UNTITLED"
            forKey:@"ToDoDocumentLastSaved"];
    [defaults registerDefaults:regdom];
 }
```

The **initialize** message is sent to each class before it receives any other message, giving it an opportunity to do something having global effect on all future instances. In ToDoController's case, the **initialize** method specifies a "catch-all" default in the registration domain of user defaults. To Do applications that are launched the first time on a system will take this default.

Opening Documents by Double-Clicking

To let users of your application open documents by double-clicking the document icon in the file system, you must complete the following steps:

1 Specify an icon and a type (file extension) for your document in the Project Attributes display of Project Builder's Project Inspector (see page 118 for an example).

2 Implement the NSApplication delegation method **application:openFile:**. This method is invoked when users double-click or select a document in the file system (for instance, using File Manager or Explorer). In your implementation, you should attempt to create your document using the path given in the second argument. If you succeed, return YES; otherwise, return NO.

3 After building the application, install it in the conventional file-system locations for applications, such as **/LocalApps** and **~/Apps**.

In ToDoController.m, implement the delegation method **applicationOpenUntitledFile:**.

```
- (BOOL)applicationOpenUntitledFile:(NSApplication *)sender
{                                                                    A
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSString *docToOpen = [defaults stringForKey:
        @"ToDoDocumentLastSaved"];
                                                                     B
    if (![docToOpen isEqualToString:@"UNTITLED"] &&
       [[NSFileManager defaultManager] fileExistsAtPath:docToOpen] &&
        [[docToOpen pathExtension] isEqualToString:@"td"]) {
        ToDoDoc *thisDoc = [[ToDoDoc alloc] initWithFile:docToOpen];
        [thisDoc activateDoc];
        return YES;
    }
    [self newDoc:self];                                              C
    return YES;
}
```

An NSApp's delegate can implement the **applicationOpenUntitledFile:** method to display an appropriate starting document when an Yellow Boxfor Windows application is launched. This specific implementation does the following:

A The class method **standardUserDefaults** returns the NSUserDefaults representing the current user's defaults. From this object, it gets the path of the To Do document that was last saved (more soon on how this was done).

B If the default is not the registration-domain one and the path references a real To Do document, it re-creates and activates the document.

C Otherwise, it creates a new document, which has a title of "UNTITLED."

User Defaults and the Defaults System

User defaults denotes information about a user's preferences that an Rhapsody program keeps between sessions. Also recorded in user defaults are initial values for applications (such as the position of windows), default values that apply globally, and defaults specific to a language (for example, the way in which time is expressed). An application typically allows its users to enter their choices into users defaults through a Preferences panel.

User defaults belong to *domains*. The most common domain consists of individual applications, but there are other domains. For example, NSGlobalDomain holds values common to all applications; there is also a language-specific domain and NSRegistrationDomain (temporary default values).

Each domain has a dictionary of keys and values representing its defaults. Keys are always strings, but values can be property lists: complex data structures comprising arrays, dictionaries, strings, and binary data. Searches for a default proceed through a search list, in which the application's domain typically comes before the global, language-specific, and registration domains.

The defaults system, which implements user defaults, includes a framework component and a command-line component. You can specify, read, and manage user defaults with the methods of NSUserDefaults and with the **defaults** utility.

The Application Quartet: NSResponder, NSApplication, NSWindow, and NSView

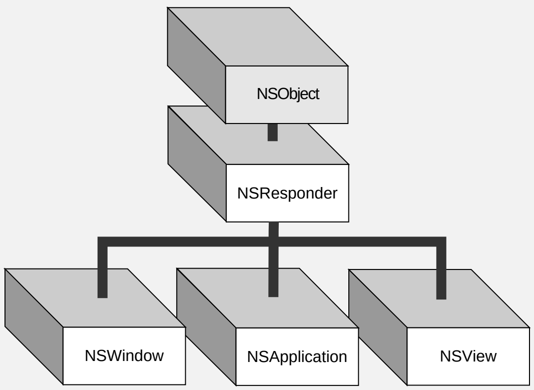
Many classes of the Application Kit stand out in terms of relative importance. NSControl, for example, is the superclass of all user-interface devices, NSText underlies all text operations, and NSMenu has obvious significance. But four classes are at the core of a running application: NSResponder, NSApplication, NSWindow, and NSView. Each of these classes plays a critical role in the two primary activities of an application: drawing the user interface and responding to events. The structure of their interaction is sometimes called the core program framework.

NSWindow

An NSWindow object manages each physical window on the screen. It draws the window's content area and responds to user actions that close, move, resize, and otherwise manipulate the window.

The main purpose of an NSWindow is to display an application's user interface (or part of it) in its *content area*: that space below the title bar and menu bar and within the window frame. A window's content is the NSViews it encloses, and at the root of this *view hierarchy* is the *content view*, which fills the content area. Based on the location of a user event, NSWindows assigns an NSView in its content area to act as *first responder*.

An NSWindow allows you to assign a custom object as its delegate and so participate in its activities.



NSResponder

NSResponder is an abstract class, but it enables event handling in all classes that inherit from it. It defines the set of messages invoked when different mouse and keyboard events occur. It also defines the mechanics of event processing among objects in an application, especially the passing of events up the *responder chain* to each *next responder* until the event is handled. See the "First Responder and the Responder Chain" on page 169 for more on the responder chain and a description of *first responder*.

NSApplication

Every application must have one NSApplication object to supervise and coordinate the overall behavior of the application. This object dispatches events to the appropriate NSWindows (which, in turn, distribute them to their NSViews). The NSApplication object manages its windows and detects and handles changes in their status as well as in its own active and inactive status. The NSApplication object is represented in each application by the global variable NSApp. To coordinate your own code with NSApp, you can assign your own custom object as its delegate.

NSView

Any object you see in a window's content area is an NSView. (Actually, since NSView is an abstract class, these objects are instances of NSView subclasses.) NSView objects are responsible for drawing and for responding to mouse and keyboard events. Each NSView owns a rectangular region associated with a particular window; it produces images within this region and responds to events occurring within the rectangle.

NSViews in a window are logically arranged in a *view hierarchy*, with the *content view* at the top of the hierarchy (see next page for more information). An NSView references its window, its superview, and its subviews. It can be the first responder for events or the next responder in the responder chain. An NSView's frame and bounds are rectangles that define its location on the screen, its dimension, and its coordinate system for drawing.

'The NSEvent class is also involved in event processing. For more about NSEvent and the event cycle, see "Events and the Event Cycle" on page 168.

The View Hierarchy

Just inside each window's content area—the area enclosed by the title bar and the other three sides of the frame—lies the content view. The content view is the root (or top) NSView in the window's view hierarchy. Conceptually like a tree, one or more NSViews may branch from the content view, one or more other NSViews may branch from these subordinate NSViews, and so on. Except for the content view, each NSView has one (and only one) NSView above it in the hierarchy. An NSView's subordinate views are called its subviews; its superior view is known as the superview.

On the screen *enclosure* determines the relationship between superview and subview: a superview encloses its subviews. This relationship has several implications for drawing:

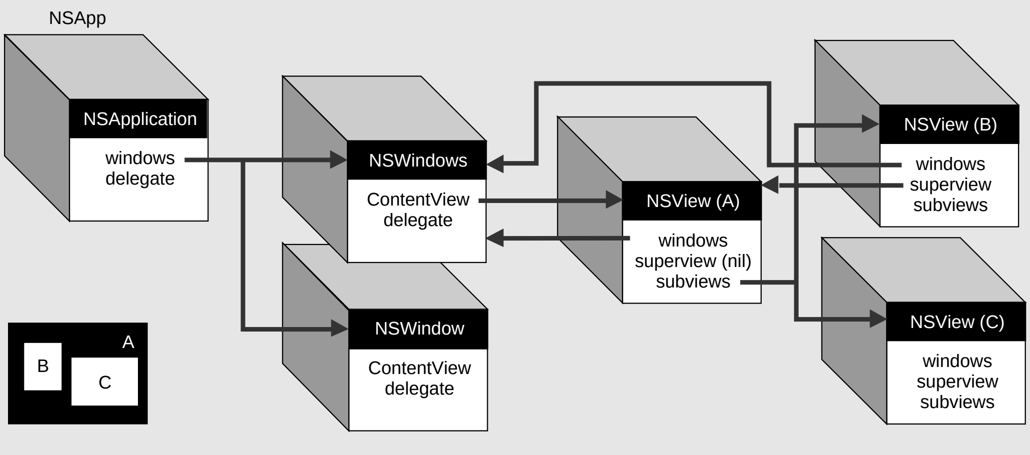
- It permits construction of a superview simply by arrangement of subviews. (An NSBrowser is an instance of a compound NSView.)
- Subviews are positioned in the coordinates of their superview, so when you move an NSView or transform its coordinate system, all subviews are moved and transformed in concert.
- Because an NSView has its own coordinate system for drawing, its drawing instructions remain constant regardless of any change in position in itself or of its superview.

Fitting Your Application In

The core program framework provides ways for your application to access the participating objects and so to enter into the action.

- The global variable NSApp identifies the NSApplication object. By sending the appropriate message to NSApp, you can obtain the application's NSWindow objects (**windows**), the key and main windows (**keyWindow** and **mainWindow**), the current event (**currentEvent**), the main menu (**mainMenu**), and the application's delegate (**delegate**).
- Once you've identified an NSWindow object, you can get its content view (by sending it **contentView**) and from that you can get all subviews of the window. By sending messages to the NSWindow object you can also get the current event (**currentEvent**), the current first responder (**firstResponder**), and the delegate (**delegate**).
- You can obtain from an NSView most objects it references. You can discover its **window**, its **superview**, and its **subviews**. Some NSView subclasses can also have delegates, which you can access with **delegate**.

By making your custom objects delegates of the NSApplication object, your application's NSWindows, and NSViews that have delegates, you can integrate your application into the core program framework and participate in what's going on.



For To Do, we want the last-saved document to be opened when the user launches the application. Accordingly, in the method that saves documents, we store the document's path in user defaults.

In **ToDoDoc.m**'s **saveDoc** method, add code to write the path of the saved document to user defaults.

(See comments in example for code to add.)

```
/* ... */
        if (result == NSOKButton) {
            fn = [sPanel filename];
            [[itemMatrix window] setTitleWithRepresentedFilename:fn];
            /* add the code below ==========> */
            if (fn && ![fn isEqualToString:@""]) {
                NSUserDefaults *defaults =
                    [NSUserDefaults standardUserDefaults];
                [defaults setObject:fn forKey:@"ToDoDocumentLastSaved"];
                [defaults synchronize];
            }
            /* <========== add the code above */
        }
/* ... */
```

The new section of code gets the NSUserDefaults object for the current user and stores the document path (**fn**) in user defaults for that user under the key ToDoDocumentLastSaved. The **synchronize** method saves this default to disk.

1 **Save edited documents when windows are closed.**

Implement the delegation method **windowShouldClose:**.

```
- (BOOL)windowShouldClose:(id)sender
{
    int result;
                                                                    A
   if (![[itemMatrix window] isDocumentEdited]) return YES;
                                                                    B
    [[itemMatrix window] makeFirstResponder:[itemMatrix window]];
    result = NSRunAlertPanel(@"Close", @"Document has been edited.
        Save changes before closing?", @"Save", @"Don't Save",
        @"Cancel");
                                                                    C
    switch(result) {
      case NSAlertDefaultReturn: {
          [self saveDocItems];
          [self saveDoc];
          return YES;
      }
        case NSAlertAlternateReturn: {
          return YES;
        }
        case NSAlertOtherReturn: {
          return NO;
        }
    }
    return NO;
}
```

When an Rhapsody Application Is Launched

When the user launches an application, the default behavior is to display the contents of the main nib file. This initial presentation could be one or more windows, but often it is just the application's menu. Often with document-centric applications, this behavior is what you want. But you aren't restricted to this behavior.

With Rhapsody applications you have a number of alternatives. The alternative that is best for an application depends on that application's nature and purpose.

Put up an untitled document. The application displays a content-less document with a window title of "UNTITLED" (or something similar). The user can start adding content immediately or can open an existing document. This is the course adopted by the TextEdit application. A variation of this approach always displays an initial window with some standard content, such as a product logo (see the Preview application).

How: The application's delegate must implement the **applicationOpenUntitledFile:** method and, in that method, create a new document or open a standard document.

Display the document that the user last saved. The first time a user launches an application, the application creates and displays an untitled document. When the user saves that document, the application stores the full path of the saved file in user defaults. The next time the user launches the application the application restores the document from the file. This is the approach taken by the To Do application.

How: Implement **applicationOpenUntitledFile**:, as before, but this time first check user defaults to see if it contains a path for a document file. If it does, verify that the file exists (it could have been moved or deleted since the last session) before opening and displaying it. Otherwise, display an untitled document. When the user closes a document or terminates the application, store the full path of the last-saved document file in user defaults.

Display an opened-document window. The opened-document window (typically small) contains a list of documents that the user currently has created or opened. Users can get a document to appear by clicking an item in the list. When users choose the Exit command, the application can terminate after closing (and, if necessary, saving) all listed documents. As a variation, the application can, when it's next launched, restore to the project window (via user defaults) the documents opened when the last session was terminated.

How: In the application's main nib file create a small window that contains a table view or browser. The project window's menu bar can contain the complete set of menus or an appropriate subset. When the application is launched, the project window is automatically displayed. When users open or create a document, create and insert an appropriate entry in the table view or browser. When users click (or double-click) on an entry, display the document.

When users click a window's close button, the window sends **windowShouldClose:** to the window's delegate. The window expects a response directing it either to close the window or leave it open. This implementation does the following:

Ⓐ Returns YES (close the window) if the document hasn't been edited.

Ⓑ Makes the window its own first responder. This has the effect of forcing the validation of cells, flushing currently entered text to the method that handles it (more on this in the next section).

Ⓒ Identifies the clicked button by evaluating the constant returned from **NSRunAlertPanel()** and returns the appropriate boolean value. If the user clicks the Save button, this method also updates internal storage with the currently displayed items (**saveDocItems**, described in the following section) and then sends **saveDoc** to itself to archive application data to a file.

Note: Do you recall the **performClose:** method that ToDoController sends the document window when the user chooses the Close command? This method simulates a mouse click on the window's close button, causing **windowShouldClose:** to be invoked.

The NSApplication object sends **applicationShouldTerminate:** to its delegate to give it notice that the application is about to terminate. In this method you should first let the user save any edited document.

1 **Save edited documents when the user quits the application.**

In **ToDoController.m**, implement the delegation method **applicationShouldTerminate:**.

```
- (BOOL)applicationShouldTerminate:(id)sender
{
    NSString *repfile = nil;
    NSArray *appWindows = [NSApp windows];                        A
    NSEnumerator *enumerator = [appWindows objectEnumerator];
    id object;

    while (object = [enumerator nextObject]) {                    B
        int result;
        id doc;
        if ((doc = [object delegate]) &&
            [doc isKindOfClass:[ToDoDoc class]] &&
            [object isDocumentEdited] ) {
            repfile = [[NSApp keyWindow] representedFilename];
            result = NSRunAlertPanel(@"To Do", @"Save %@?", @"Save",
                @"Don't Save", @"Cancel",
                ([repfile isEqualToString:@""]?@"UNTITLED":repfile));
            switch(result) {
              case NSAlertDefaultReturn:
                [doc saveDocItems];
                [doc saveDoc];
                break;
              case NSAlertAlternateReturn:
                [[NSApp keyWindow] close];                        C
                break;
              case NSAlertOtherReturn:
                return NO;
            }
        }
    }
    return YES;
}
```

Much of the code in this method is similar to that for **windowShouldClose:**; if a window is managed by ToDoDoc, the **applicationShouldTerminate:** method puts up an attention panel and responds according to the user's choice. However, there are some significant differences:

A Returns all open windows of the application in an NSArray. Remember, one of the jobs of an NSApplication is to track and manage all windows.

B Enumerates and processes the NSWindow objects in this NSArray, as noted.

C If the user clicks "Don't Save," the **close** message forces the window to close (without sending the **windowShouldClose:** delegate message).

Managing ToDo's Data and Coordinating its Display

If you recall the discussion on To Do's design earlier in this chapter ("How To Do Stores and Accesses its Data" on page 117), you'll remember that the application's real data consists of instances of the model class, ToDoItem. To Do stores these objects in arrays and stores the arrays in a dictionary; it uses dates as the keys for accessing specific arrays. (Both the dictionary and its arrays are mutable, of course.) You might also recall that this design depends on a positional correspondence between the text fields of the document interface and the "slots" of the arrays.

To lend clarity to this design's implementation, this section follows the process from start to finish through which the ToDoDoc class handles entered data, and organizes, displays, and stores it. It also shows how the display and manipulation of data is driven by the selections made in the CalendarMatrix object.

Start by revisiting a portion of code you wrote earlier for ToDoDoc's **initWithFile:** method.

```
- initWithFile:(NSString *)aFile
{
    /* ... */
    if (aFile) {
        activeDays = [NSUnarchiver unarchiveObjectWithFile:aFile];
        if (activeDays)
            activeDays = [activeDays retain];
        else
           NSRunAlertPanel(@"To Do", @"Couldn't unarchive file %@",
                nil, nil, nil, aFile);
    } else {
        activeDays = [[NSMutableDictionary alloc] init];
        [self setCurrentItems:nil];
    }
/* ... */
}
```

Assume the user has chosen the New command from the Document menu. Since there is no archive file (**aFile** is **nil**), the **activeDays** dictionary is created but is left empty. Then **initWithFile:** invokes its own **setCurrentItems:** method, passing in **nil**.

1 **Set the current items or, if necessary, create and prepare the array that holds them.**

Implement **setCurrentItems:**.

```
- (void)setCurrentItems:(NSMutableArray *)newItems
{
    if (currentItems) [currentItems autorelease];

    if (newItems)
        currentItems = [newItems mutableCopy];
    else {
        int numRows, numCols;
        [itemMatrix getNumberOfRows:&numRows columns:&numCols];
        currentItems = [[NSMutableArray alloc]
            initWithCapacity:numRows];
        while (--numRows >= 0)
            [currentItems addObject:@""];
    }
}
```

This "set" accessor method is like other such methods, except in how it handles a **nil** argument. In this case, **nil** signifies that the array does not exist, and so it must be created. Not only does **setCurrentItems:** create the array, but it "initializes" it with empty string objects. It does this because NSMutableArray's methods cannot tolerate **nil** within the bounds of the array.

So there's now a **currentItems** array ready to accept ToDoItems. Imagine yourself using the application. What are the user events that cause a ToDoItem to be added to the **currentItems** array? To Do allows entry of items "on the fly," and thus does not require the user to click a button to add a ToDoItem to the array. Specifically, items are added when users type something and then:

- Press the Tab key.
- Press the Enter key.
- Click outside the text field.

The **controlTextDidEndEditing:** delegation method makes these scenarios possible. The matrix of editable text fields (**itemMatrix**) invokes this method when the cursor leaves a text field that has been edited.

1 **As items are entered in the interface, add ToDoItems to internal storage, delete them, or modify them, as appropriate.**

Implement **controlTextDidEndEditing:**. as shown.

```
- (void)controlTextDidEndEditing:(NSNotification *)notif
{
    id curItem, newItem;
    int row = [itemMatrix selectedRow];
    NSString *selName = [[itemMatrix selectedCell] stringValue];
                                                                    A
    if (![[itemMatrix window] isDocumentEdited] ||
        (row >= [currentItems count])) return;
    if (!currentItems)
        [self setCurrentItems:nil];
                                                                    B
    if ([selName isEqualToString:@""] &&
        ([[currentItems objectAtIndex:row] isKindOfClass:
            [ToDoItem class]]) &&
        (![[[currentItems objectAtIndex:row] itemName]
            isEqualToString:@""]))
        [currentItems replaceObjectAtIndex:row withObject:@""];
                                                                    C
    else if ([[currentItems objectAtIndex:row] isKindOfClass:
            [ToDoItem class]] &&
            (![[[currentItems objectAtIndex:row] itemName]
            isEqualToString:selName]) )
        [[currentItems objectAtIndex:row] setItemName:selName];
                                                                    D
   else if (![selName isEqualToString:@""]) {
        newItem = [[ToDoItem alloc] initWithName:selName
            andDate:[calendar selectedDay]];
        [currentItems replaceObjectAtIndex:row withObject:newItem];
        [newItem release];
    }
                                                                    E
    [self updateMatrix];
}
```

A control sends **controlTextDidEndEditing:** to its delegate when the insertion point *leaves* a text field. In addition to creating new ToDoItems, this implementation of **controlTextDidEndEditing:** removes ToDoItems from arrays and modifies item text. What it does is appropriate to what the user does.

A If the document hasn't been edited (see **controlTextDidChange:**) or if the selected row exceeds the array bounds, the code returns because there's no reason to proceed. Otherwise, it initializes a **currentItems** array if one doesn't exist.

B If the user deletes the text of an existing item, the code removes the ToDoItem that positionally corresponds to the row of that deleted text.

C It changes the name of an item if the text entered in a field doesn't match the name of the corresponding item in the **currentItems** array.

D If either of the two previous conditions don't apply, and text has been entered, it creates a new ToDoItem and inserts it in the **currentItems** array.

E Updates the list of items in the document interface.

1 **Update the document interface with the current items.**

Implement **updateMatrix:**.

```
- (void)updateMatrix
{
    int i, cnt = [currentItems count],
        rows = [[itemMatrix cells] count];
    ToDoItem *thisItem;

    for (i=0; i<cnt, i<rows; i++) {
        NSDate *due;
        thisItem = [currentItems objectAtIndex:i];
        if ([thisItem isKindOfClass:[ToDoItem class]]) {        A
            if ( [thisItem secsUntilDue] )
                due = [[thisItem day] addTimeInterval:
                          [thisItem secsUntilDue]];
            else
                due = nil;
            [[itemMatrix cellAtRow:i column:0] setStringValue:
                [thisItem itemName]];
            [[markMatrix cellAtRow:i column:0] setTimeDue:due];
            [[markMatrix cellAtRow:i column:0] setTriState:
                [thisItem itemStatus]];
        }
        else  {                                                  B
            [[itemMatrix cellAtRow:i column:0] setStringValue:@""];
            [[markMatrix cellAtRow:i column:0] setTitle:@""];
            [[markMatrix cellAtRow:i column:0] setImage:nil];
        }
    }
}
```

The **updateMatrix** method writes the names of the items (ToDoItems) in the **currentItems** array to the text fields of **itemMatrix**. It also updates the visual appearance of the cells in the matrix (**markMatrix**) next to **itemMatrix**. These cells are instances of a custom subclass of NSButtonCell that you will create later in this tutorial. For now, just type all the code above; later, when you create the cell class (ToDoCell) you can refer back to this example.

Basically, this method cycles through the array of items, doing the following:

A If an object in the array is a ToDoItem, it writes the item name to the text field pegged to the array slot and updates the button cell next to the field.

B If an object isn't a ToDoItem, it blanks the corresponding text field and cell.

1 **Respond to user actions in the calendar.**

Implement CalendarMatrix's delegation methods.

```
- (void)calendarMatrix:(CalendarMatrix *)matrix
        didChangeToDate:(NSDate *)date
{
    [[itemMatrix window] makeFirstResponder:[itemMatrix window]];
    [self saveDocItems];

    [self setCurrentItems:[activeDays objectForKey:date]];
    [dayLabel setStringValue:[date descriptionWithCalendarFormat:
        @"To Do on %a %B %d %Y" timeZone:[NSTimeZone defaultTimeZone]
        locale:nil]];
    [self updateMatrix];
}

- (void)calendarMatrix:(CalendarMatrix *)matrix
        didChangeToMonth:(int)mo year:(int)yr
{
    [self saveDocItems];
    [self setCurrentItems:nil];
    [self updateMatrix];
}
```

As you might recall, CalendarMatrix declared two methods to allow delegates to "hook into" its behavior. Its delegate for this application is ToDoDoc.

A The calendar sends **calendarMatrix:didChangeToDate:** when users click a new day of the month. This implementation saves the current items to the **activeDays** dictionary. It then sets the current items to be those corresponding to the selected date (if there are no items for that date, the **objectForKey:** message returns **nil** and the **currentItems** array is initialized with empty strings). Finally it updates the matrix with the new data.

B The calendar sends **calendarMatrix:didChangeToMonth:year:** when users go to a new month and (possibly) a new year. This implementation responds by saving the current items to internal storage and presenting a blank list of items.

1 **Save the data to internal storage.**

Implement **saveDocItems:**.

1 **Archive and unarchive the document's data.**

Implement **encodeWithCoder:** and **initWithCoder:** to archive and unarchive the dictionary holding the arrays of ToDoItems.

```
- (void)saveDocItems
{
    ToDoItem *anItem;
    int i, cnt = [currentItems count];
    // save day's current items (array) to document dictionary
    for (i=0; i<cnt; i++) {
        if ( (anItem = [currentItems objectAtIndex:i])  &&
             ([anItem isKindOfClass:[ToDoItem class]]) ) {
            [activeDays setObject:currentItems forKey:
                [anItem day]];
            break;
        }
    }
}
```

This method inspects the **currentItems** array and, if it contains at least one ToDoItem, puts the array in the **activeDays** dictionary with a key corresponding to the date.

Now that you've completed the methods for saving and archiving the collection objects holding ToDoItems, assume that the user has saved his or her document and then opens it.

1 **Perform set-up tasks when the document's nib file is unarchived.**

Implement **awakeFromNib** as shown at right.

```
- (void)awakeFromNib
{
    int i;
    NSDate *date;

    date = [calendar selectedDay];
    [self setCurrentItems:[activeDays objectForKey:date]];
    /* set up self as delegates */
    [[itemMatrix window] setDelegate:self];
    [itemMatrix setDelegate:self];
    [[itemMatrix window] makeKeyAndOrderFront:self];
}
```

When the **ToDoDoc.nib** file is completely unarchived, **awakeFromNib** is invoked. It sets the current items for today, sets a couple of delegates, and puts the document window in front of all other windows.

Note: This method sets some delegates programmatically, which is redundant since you set these delegates in Interface Builder. However, this code demonstrates the programmatic route—and no harm done.

1 **Set up the document once it's created or opened.**

Implement **activateDoc** as shown at right.

```
- (void)activateDoc
{
    if ([currentItems count]) [self updateMatrix];
    [dayLabel setStringValue:[[calendar selectedDay]
         descriptionWithCalendarFormat:@"To Do on %a %B %d %Y"
         timeZone:[NSTimeZone defaultTimeZone] locale:nil]];
}
```

The **activateDoc** method is invoked right after a To Do document is created or opened. It starts the ball rolling by updating the list matrices of the document and writing the current date to the "To Do on *<date>*" label.

To Do Tutorial - Extended Chapter 5

5

What You'll Learn

Creating and managing an inspector

Responding to user actions

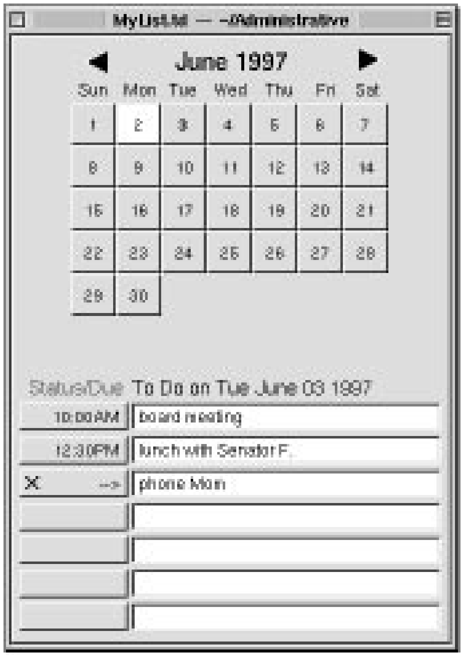
Coordinating events within an application

Overriding behavior of an Application Kit class

Creating a custom NSView subclass

Using timers

Drawing and compositing essentials



You can find the ToDo project in the ***AppKit*** *subdirectory of* ***/System/Developer/Examples.***

Chapter 5
Extending the To Do Application

In this tutorial you will add features and functionality to the To Do application you created in the previous tutorial. The finished application will allow users to do much more than entering to-do items into a daily list. In an inspector they will be able to:

- *Specify the times those items are due.*
- *Request that they be notified at a specified interval before the due time.*
- *Associate notes with items.*
- *Mark items as complete or deferred.*
- *Reschedule uncompleted items.*

Moreover, the document interface will have a custom button for each item. The button will display the item's due time. Users can also click the button to change an item's status. Changes users make in the document will be immediately reflected in the inspector, and vice versa.

Events and the Event Cycle

Conceptually, this chapter focuses primarily on *events*—especially events originating from user actions—and how, as a programmer, you intercept, handle, and coordinate them with Rhapsody APIs. Therefore, it's best to begin with a short overview of this topic.

You can depict the interaction between a user and an Rhapsody application as a cyclical process, with the Window Server playing an intermediary role (see illustration below). This cycle—the *event cycle*—usually starts at launch time when the application (which includes all the frameworks it's linked to) sends a stream of PostScript code to the Window Server to have it draw the application interface.

Then the application begins its main event loop and begins accepting input from the user (see next page). When users click or drag the mouse or type on the keyboard, the Window Server detects these actions and processes them, passing them to the application as events. Often the application, in response to these events, returns another stream of PostScript code to the Window Server to have it redraw the interface.

In addition to events, applications can respond to other kinds of input, particularly timers, data received at a port, and data waiting at a file descriptor. But events are the most important kind of input.

Events

The Window Server treats each user action as an event. It associates the event with a window and reported to the application that created the window. Events are objects: instances of NSEvent composed from information derived from the user action.

All event methods defined in NSResponder (such as **mouseDown:** and **keyDown:**) take an NSEvent as their argument. You can query an NSEvent to discover its window, the location of the event within the window, and the time the event occurred (relative to system start-up). You can also find out which (if any) modifier keys were pressed, such as Command, Option (Alternate), and Control), the codes that identify characters and keys, and various other kinds of information.

An NSEvent also divulges the type of event it represents. There are many event types (NSEventType); they fall into four categories:

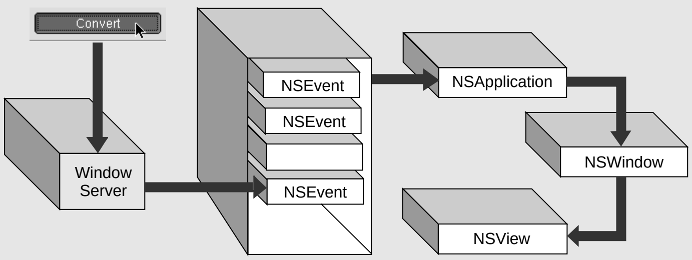
- **Keyboard events**. Generated when a key is pressed down, a pressed key is released, or a modifier key changes. Of these, key-down events are the most useful. When you handle a key-down event, you often determine the character or characters associated with the event by sending the NSEvent a **characters** message.
- **Mouse event**. Mouse events are generated by changes in the state of the mouse buttons (that is, down and up) for both left and right mouse buttons and during mouse dragging. Events are also generated when the mouse simply moves, without any button pressed.
- **Tracking-rectangle events**. If the application has asked the window system to set a tracking rectangle in a window, the window system creates mouse-entered and mouse-exit events when the cursor enters the rectangle or leaves it.
- **Periodic events**. A periodic event notifies an application that a certain time interval has elapsed. An application can request that periodic events be placed in its event queue at a certain frequency. They are usually used during a tracking loop. (These events aren't passed to an NSWindow.)

The Event Queue and Event Dispatching

When an application starts up, the NSApplication object (NSApp) starts the main event loop and begins receiving events from the Window Server. As NSEvents arrive, they're put in the *event queue* in the order they're received. On each cycle of the loop, NSApp gets the topmost event, analyzes it, and sends an *event message* to the appropriate object. (Event messages are defined by NSResponder and correspond to particular events.) When NSApp finishes processing the event, it gets the next event and repeats the process again and again until the application terminates.

The object that is "appropriate" for an event depends on the type of event. NSApp sends most event messages to the NSWindow in which the user action occurred. If the event is a keyboard or mouse event, the NSWindow forwards the message to one of the objects in its view hierarchy: the NSView within which the mouse was clicked or the key was pressed. If the NSView can respond to the event—that is, it accepts first responder status and defines an NSResponder method corresponding to the event message—it handles the event.

If the NSView cannot handle an event, it forwards the message to the next responder in the responder chain (see next section for details). It travels up the responder chain until an object handles it.



NSWindow handles some events itself, and doesn't forward them to an NSView, such as window-moved, window-resized, and window-exposed events. (Since these are handled by NSWindow itself, they are not defined in NSResponder.) NSApp also processes a few kinds of events itself, such as application-activate and application-deactivate events.

First Responder and the Responder Chain

Each NSWindow in an application keeps track of the object in its view hierarchy that has *first responder* status. This is the NSView that currently receives keyboard events for the window. By default, an NSWindow is its own first responder, but any NSView within the window can become first responder when the user clicks it with the mouse.

You can also set the first responder programmatically with the NSWindow's **makeFirstResponder:** method. Moreover, the first-responder object can be a target of an action message sent by an NSControl, such as a button or a matrix. Programmatically, you do this by sending **setTarget:** to the NSControl (or its cell) with an argument of **nil**. You can do the same thing in Interface Builder by making a target/action connection between the NSControl and the First Responder icon in the Instances display of the nib file window.

Recall that all NSViews of the application, as well as all NSWindows and the application object itself, inherit from NSResponder, which defines the default message-handling behavior: events are passed up the responder chain. Many Application Kit objects, of course, override this behavior, so events are passed up the chain until they reach an object that does respond.

The series of next responders in the responder chain is determined by the interrelationships between the application's NSView, NSWindow, and NSApplication objects (see page 152). For an NSView, the next responder is usually its superview; the content view's next responder is the NSWindow. From there, the event is passed to the NSApplication object.

For action messages sent to the first responder, the trail back through possible respondents is even more detailed. The messages are first passed up the responder chain to the NSWindow and then to the NSWindow's delegate. Then, if the previous sequence occurred in the key window, the same path is followed for the main window. Then the NSApplication object tries to respond, and failing that, it goes to NSApp's delegate.

Overriding Behavior of an Application Kit Class: An Example

You can often achieve significant gains in object behavior by making a subclass that adds only a small amount of code to its superclass. Such is the case with the subclass you'll create in this section: SelectionNotifMatrix.

The need for this class is this: An instance of NSMatrix is a control and thus can send action messages to its cell's targets; but when it contains NSTextFieldCells, action messages are sent only when users press the Return key in a cell. You want the inspector (which you'll create in the next section) to synchronize its displays when the user selects a new item by clicking a text field. To do this, you will *override* the method in NSMatrix that is invoked when users click the matrix; in your implementation, you'll invoke the superclass method, detect the selected row, and then post a notification to interested observers.

1 **Create template source-code files and add to the project.**

Choose File ► New In Project.

In the New File In ToDo panel, select the Class suitcase, turn on the Create header switch, and type "SelectionNotifMatrix" after Name.

1 **Add declarations to the header file.**

```
#import <AppKit/AppKit.h>

extern NSString *SelectionInMatrixNotification =          A
         @"SelectionInMatrixNotification";

@interface SelectionNotifMatrix : NSMatrix
{
}

- (void)mouseDown:(NSEvent *)theEvent;                    B

@end
```

A Declares a string constant identifying the notification that will be posted.

B Declares **mouseDown:**, the method implemented by the superclass but overridden by SelectionNotifMatrix.

Before You Go On

Remember, build the project frequently to catch any errors quickly, to get a sense of how the application is developing, and (just as important) to give yourself a break from coding.

1 **Override mouseDown:**

In SelectionNotifMatrix.m, implement **mouseDown:** as shown here.

```
- (void)mouseDown:(NSEvent *)theEvent
{
    int row;
    [super mouseDown:theEvent];                                    A

    row = [self selectedRow];                                      B
    if (row != -1) {
        [[NSNotificationCenter defaultCenter]
           postNotificationName:@"SelectionInMatrixNotification"
           object:self userInfo:[NSDictionary
           dictionaryWithObjectsAndKeys:
           [NSNumber numberWithInt:row], @"ItemIndex", nil]];
    }
}
```

This override of **mouseDown:** does the following:

A Invokes NSMatrix's implementation of **mouseDown:** to allow the normal processing of this event.

B Gets the row of the cell clicked and, if it's a valid row, creates a **userInfo** dictionary containing the index of the clicked row, and posts the SelectionInMatrixNotification.

Now that you've created the SelectionNotifMatrix class, you must re-assign the class membership of the object in the interface. You can do this easily in Interface Builder.

1 **Assign the new class to the matrix of text fields.**

In Interface Builder:

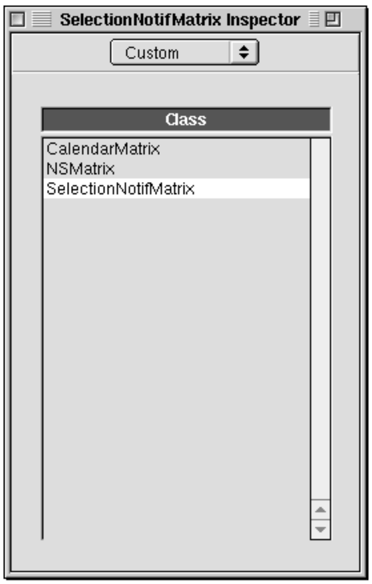
In the Classes display of **ToDoDoc.nib**, select NSMatrix as the superclass.

Choose Read File from the Classes menu.

In the Read File browser, select SelectionNotifMatrix and click OK.

Select the matrix of text cells.

Choose SelectionNotifMatrix in the Custom display of the inspector.



The Custom Classes browser lists the original class of the selected object and all compatible custom subclasses.

Creating and Managing an Inspector (ToDoInspector)

An inspector is a panel of fields and controls that enable users to examine and set an object's attributes. Because objects often have many attributes and because you want to make it easy for users to set those attributes, inspectors usually have more than one display; users typically access these multiple displays using a pop-up list.

The ToDo application has an inspector panel that allows users to inspect and set the attributes of the currently selected ToDoItem. The inspector panel has its own controller: ToDoInspector. While showing you how to create the inspector panel and ToDoInspector, this section focuses on four things:

- Managing displays according to user selections
- Getting the current ToDoItem
- Updating the currently selected display
- Updating the current ToDoItem as users make changes to it

In Interface Builder

1 **Create a new nib file named ToDoInspector.nib and add it to the ToDo project.**

1 **Create the inspector panel.**

Drag a panel object from the Windows palette.

Make the title of the panel "Inspector."

Turn on the panel's sizing border and resize it, using the example as a guide.

Turn off the panel's sizing border.

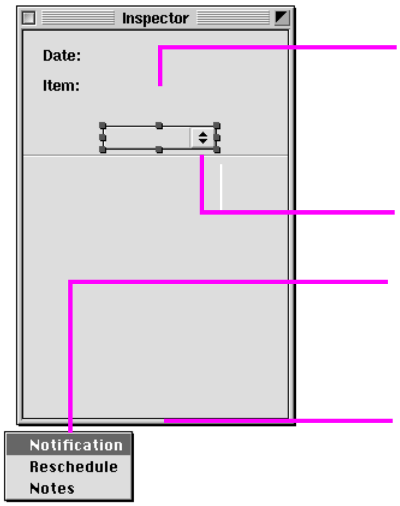
Put labels and fields on the panel and set their attributes (as shown).

Put a pop-up button on the panel and set cell titles (as shown).

Assign tags to the pop-up button cells.

Create a separator line just below the pop-up button.

Put an empty box object in the lower part of the panel.



The text fields should have a gray background and should not be editable nor scrollable. Enlarge the lower field to accommodate long item entries.

Double-click to display in a floating window the three default items (Item1, Item2, Item3)

In the floating window, double-click the title tof each item to select it; type the new title.

Assign tags 0 to 2 to the cells in downward order.

Turn off the box's Title attribute and resize the object so it fits just inside the lower part of the panel. To provide a guide for resizing, this example shows the box having a border; turn the border off after resizing.

Before You Go On

You might be wondering about the empty box object in the lower part of the panel. This box by itself may not seem a promising thing for displaying object attributes, but it is critical to the workings of the inspector panel. A box that you drag from the Views palette contains one subview, called the *content view.* NSBox's content view fits entirely within the bounds of the box. NSBox provides methods for obtaining and changing the content view of boxes. You'll use these methods to change what the inspector panel displays.

1 **Create an off-screen panel holding the inspector's displays.**

Drag a panel object from the Windows palette.

Resize the panel, using the example at right as a guide.

Put the labels, text fields, scroll view, and switch and radio-button matrices on the panel shown in the example at right.

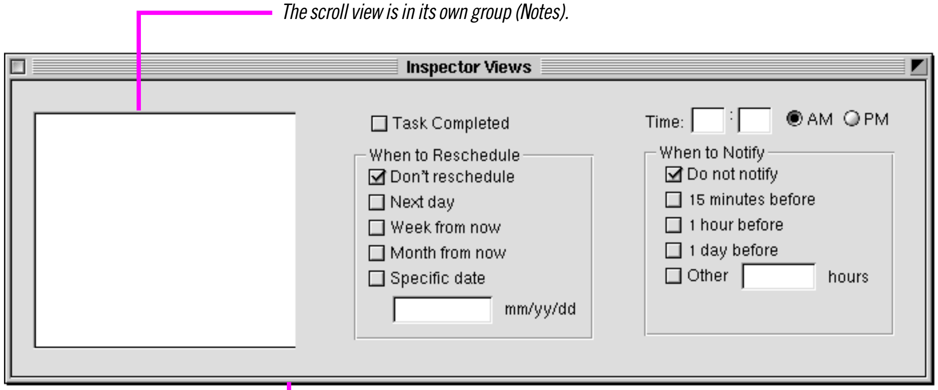
Set the mode attributes of the switch matrices to Radio.

Make the "When to reschedule" and "When to notify" groupings (boxes).

Make three other groupings for the three displays: Notes, Reschedule, and Notification.

Make the resulting outer boxes the same size as the "dummy" view in the inspector panel.

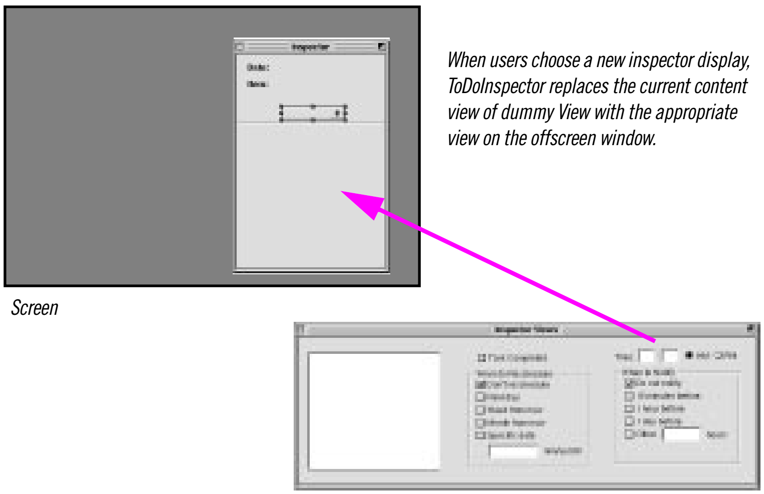
The scroll view is in its own group (Notes).



Turn off the border attribute of each outer box.

Before You Go On

You probably now see where the inspector panel gets its displays and how it puts them in place. When the inspector panel is first opened (and **ToDoInspector.nib** is loaded) the inspector controller, ToDoInspector, replaces the content view of the inspector's empty box (**dummyView**) with the content view of the Notification box in the off-screen panel. Thereafter, every time the user chooses a new pop-up button in the inspector panel, ToDoInspector replaces the currently displayed content view with the content view of the associated off-screen box.



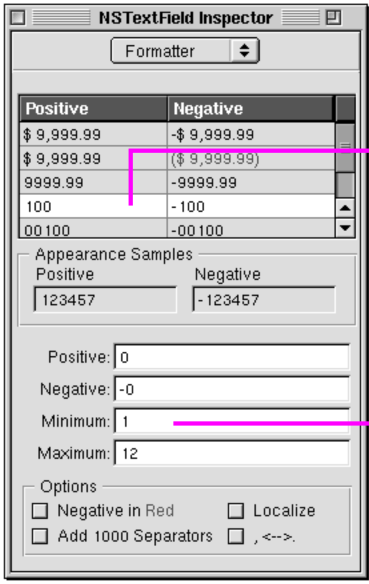
When users choose a new inspector display, ToDoInspector replaces the current content view of dummy View with the appropriate view on the offscreen window.

1 **Apply formatters to fields of the inspector.**

Drag a number-formatter object from the DataViews palette and drop it on the hours field of the Notification display (the first field after “Time:”).

In the inspector’s Formatter display, set the field to have a minimum value of 1 and a maximum value of 12 (see example).

Apply a number formatter to the minutes field (the second field after “Time:”) and set it to have a minimum value of 0 and a maximum value of 59.



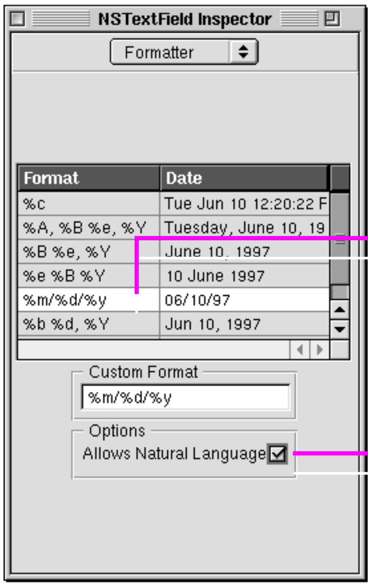
Select a simple integer format for the hour and minute “Time” fields.

Users cannot enter values that are less than this into the field; the cursor will not leave the field until they enter an appropriate value.

Interface Builder provides a palette object that formats dates in addition to the one that formats numbers. You can identify this object on the DataViews palette through its calendar icon.

Drag a date-formatter object from the DataViews palette onto the date field in the Rescheduling display (the “mm/dd/yy” field).

In the inspector’s Formatter display, select the “%m/%d/%y” format from the table.



Select this format for the field.

The formatter rejects dates entered in any other format. It also verifies that the individual fields contain proper values (for instance, “13” is disallowed as a month).

Check if you want the formatter to interpret common temporal expressions such as “tomorrow” or “next month.”

1 **Define the ToDoInspector class.**

Create a subclass of NSObject and name it “ToDoInspector.”

Add the outlets and actions in the tables at right to the new class.

Instantiate ToDoInspector.

Connect the ToDoInspector object to its outlets and as the target of action messages (see tables at right).

Connect ToDoInspector and the inspector panel via the panel’s **delegate** outlet.

Close both panels.

Save **ToDoInspector.nib**.

Create source-code files for ToDoInspector and add them to the project.

Outlet	Connection From ToDoInspector To...
dummyView	The empty box object in the inspector panel
inspectorViews	The title bar of the off-screen panel
notesView	The box in the off-screen panel containing the scroll view
notifView	The box in the off-screen panel containing the fields and controls related to notification of impending items
reschedView	The box in the off-screen panel containing the fields and controls related to rescheduling items
inspPopUp	The pop-up button on the inspector panel
inspDate	The uneditable text field next to the “Date” label
inspItem	The uneditable text field next to the “Item” label
inspNotifHour	The first field after the “Time” label
inspNotifMinute	The second field after the “Time” label
inspNotifAMPM	The matrix holding the “AM” and “PM” radio buttons
inspNotifOtherHours	The text field in the “When to Notify” box
inspNotifSwitchMatrix	The matrix of switches in the “When to Notify” box
inspSchedComplete	The “Task Completed” switch
inspSchedDate	The text field in the “When to Reschedule” box
inspSchedMatrix	The matrix of switches in the “When to Reschedule” box
inspNotes	The text object inside the scroll view

Action	Connection To ToDoInspector From...
newInspectorView:	The pop-up button on the inspector panel
switchChecked:	The matrix of switches in the “When to Notify” box, the AM-PM matrix, the “Task Completed” switch, and the matrix of switches in the “When to Reschedule” switches

In Project Builder

1 Add declarations to ToDoInspector.h.

Open **ToDoInspector.h**.

Type the declarations shown at right (ellipses indicate existing declarations).

Import **ToDoItem.h** and **ToDoDoc.h**.

```
@interface ToDoInspector : NSObject
{
    ToDoItem *currentItem;
    /* ... */
}
/* ... */
- (void)setCurrentItem:(ToDoItem *)newItem;
- (ToDoItem *)currentItem;
- (void)updateInspector:(ToDoItem *)item;
@end
```

The ToDoInspector class has a utility function for clearing switches set in a matrix and defines constants for the tags assigned to the pop-up buttons.

Open **ToDoInspector.m**.

Forward-declare **clearButtonMatrix()** at the beginning of the file.

Define **enum** constants for the pop-up button tags.

```
static void clearButtonMatrix(id matrix);
enum { notifTag = 0, reschedTag, notesTag };
```

Using tags to identify cells rather than cell titles is a better localization strategy.

ToDoInspector has two accessor methods, one that gives out the current item and one that sets the current item.

1 Implement the accessor methods for the class.

Implement **currentItem** to return the instance variables it names.

Implement **setCurrentItem:** as shown at right.

```
- (void)setCurrentItem:(ToDoItem *)newItem
{
    if (currentItem) [currentItem autorelease];
    if (newItem)
        currentItem = [newItem retain];                A
    else
        currentItem = nil;
    [self updateInspector:currentItem];                B
}
```

The implementation of **setCurrentItem:**'s "set" accessor method probably seems familiar to you—except for a couple of things:

A Instead of copying the new value, this implementation retains it. By retaining, it *shares* the current ToDoItem with the document controller (ToDoDoc) that has sent the **setCurrentItem:** message, enabling both objects to update the same ToDoItem simultaneously.

Later in this section, you'll invoke ToDoInspector's **setCurrentItem:** method in various places in **ToDoDoc.m**.

B Updates the current display of the inspector with the appropriate values of the new ToDoItem.

1 **Switch inspector displays based on user selections.**

Implement **newInspectorView:**.

```
- (void)newInspectorView:(id)sender
{
    NSBox *newView=nil;
    NSView *cView = [[inspPopUp window] contentView];          A
    int selected = [[inspPopUp selectedItem] tag];             B
    switch(selected){
      case notifTag:
        newView = notifView;
        break;
      case reschedTag:
        newView = reschedView;
        break;
      case notesTag:
        newView = notesView;
        break;
    }
    if ([[cView subviews] containsObject:newView]) return;     C
    [dummyView setContentView:newView];                        D
    if (newView == notifView) [inspNotifHour selectText:self];
    if (newView == notesView) [inspNotes
        setSelectedRange:NSMakeRange(0,0)];
    [self updateInspector:currentItem];                        E
    [cView setNeedsDisplay:YES];
}
```

This method switches the current inspector display according to the pop-up button users select; it does this switching by replacing the **dummyView**'s content view. Toward this end, the method:

A Gets the panel's content view and the tag of the selected pop-up button.

B Assigns to the **newView** local variable the off-screen box object corresponding to the tag of the selected pop-up button.

C Returns if the selected display is already on the inspector panel. The **subviews** message returns an array of all subviews of the inspector panel's control view, and the **containsObject:** message determines if the chosen display is among these subviews.

D Replaces the content view of the inspector panel's **dummyView.** In **awakeFromNib** (which you'll soon implement) you'll retain each original content view. The **setContentView:** method replaces the new view and releases the old one; because it's retained, the replaced view remains visible.

E Updates the inspector with the current item; this item hasn't changed, but the display is new and so the set of instance variables to be displayed is different. The **setNeedsDisplay:** message forces a re-draw of the inspector panel's views.

1 **Update the current inspector display with the new ToDoItem.**

Write the first part of the **updateInspector:** method shown at right.

```
- (void)updateInspector:(ToDoItem *)newItem
{
    int minute=0, hour=0, selected=0;
    selected = [[inspPopUp selectedItem] tag];                          A
    [[inspPopUp window] orderFront:self];
    if (newItem && [newItem isKindOfClass:[ToDoItem class]]) {          B
        [inspItem setStringValue:[newItem itemName]];
        [inspDate setStringValue:[[newItem day]
            descriptionWithCalendarFormat:@"%a, %b %d %Y"
            timeZone:[NSTimeZone localTimeZone] locale:nil]];
        switch(selected) {
            case notifTag: {                                            C
                long notifSecs, dueSecs = [newItem secsUntilDue];
                BOOL ampm = ConvertSecondsToTime(dueSecs, &hour, &minute);
                [[inspNotifAMPM cellAtRow:0 column:0] setState:!ampm];
                [[inspNotifAMPM cellAtRow:0 column:1] setState:ampm];
                [inspNotifHour setIntValue:hour];
                [inspNotifMinute setIntValue:minute];
                notifSecs = dueSecs - [newItem secsUntilNotif];
                if (notifSecs == dueSecs) notifSecs = 0;
                clearButtonMatrix(inspNotifSwitchMatrix);
                switch(notifSecs) {                                     D
                  case 0:
                    [[inspNotifSwitchMatrix cellAtRow:0 column:0]
                        setState:YES];
                    break;
                  case (hrInSecs/4):
                    [[inspNotifSwitchMatrix cellAtRow:1 column:0]
                        setState:YES];
                    break;
                  case (hrInSecs):
                    [[inspNotifSwitchMatrix cellAtRow:2 column:0]
                        setState:YES];
                    break;
                  case (dayInSecs):
                    [[inspNotifSwitchMatrix cellAtRow:3 column:0]
                        setState:YES];
                    break;
                  default:  /* Other */
                    [[inspNotifSwitchMatrix cellAtRow:4 column:0]
                        setState:YES];
                    [inspNotifOtherHours setIntValue:
                        ((dueSecs-notifSecs)/hrInSecs)];
                    break;
                }
                break;
            }
          case reschedTag:
            break;
```

The **updateInspector:** method is a long one, so we'll approach it in stages. This first part updates the common data elements (item name and date) and, if the selected display is for notifications, updates that display.

A Gets the tag assigned to the selected pop-up button.

B Tests the argument **newItem** to see if it is a ToDoItem. This test is important because if the argument is **nil**, the method clears the display of existing data (next example).

If **newItem** is a ToDoItem, **updateInspector:** first updates the Item and Date fields.

C If the tag of the selected pop-up button is **notifTag**, updates the associated inspector display. This task starts by converting the due time from seconds to hour, minute, and PM boolean values and then setting the appropriate fields and button matrix with these values.

D Sets the appropriate switch in the "When to Notify" matrix. It starts with the difference (in seconds) between the time the item is due and the time the item notification is sent. It calls **clearButtonMatrix()** to turn all switches off and then, in a switch statement, sets the switch corresponding to the difference in value between seconds from midnight before due and before notification.

Before You Go On

Update the Notes display. Add code to update the inspector's Notes display from the information in the ToDoItem passed into **updateInspector:**. (Check the documentation on NSText to see what method is suitable for this.) The selected pop-up button must have **notesTag** assigned to it. Also put the cursor at the start of the text object by selecting a "null" range.

Note that this tutorial omits the rescheduling logic of the ToDo application, including the code in this method that would update the "Reschedule" display. Rescheduling of ToDoItems is reserved as an optional exercise for you at the end of this tutorial.

Finish the implementation of **updateInspector:** by resetting all displays if the argument is **nil**.

```
    }
    else if (!newItem) { /* newItem is nil */
        [inspItem setStringValue:@""];
        [inspDate setStringValue:@""];
        [inspNotifHour setStringValue:@""];
        [inspNotifMinute setStringValue:@""];
        [[inspNotifAMPM cellAtRow:0 column:0] setState:YES];
        [[inspNotifAMPM cellAtRow:0 column:1] setState:NO];
        clearButtonMatrix(inspNotifSwitchMatrix);
        [[inspNotifSwitchMatrix cellAtRow:0 column:0]
            setState:YES];
        [inspNotifOtherHours setStringValue:@""];
        [inspNotes setString:@""];
     }
}
```

As you've most likely noticed, the **updateInspector:** method calls the function **clearButtonMatrix()**, which resets the states of all button cells in a switch matrix to NO.

Implement the **clearButtonMatrix()** utility function.

```
void clearButtonMatrix(id matrix)
{
    int i, rows, cols;
    [matrix getNumberOfRows:&rows columns:&cols];
    for(i=0; i<rows; i++)
        [[matrix cellAtRow:i column:0] setState:NO];
}
```

The **getNumberOfRows:columns:** message returns, by indirection in the first argument, the number of cells in **itemMatrix**.

Making a Custom View

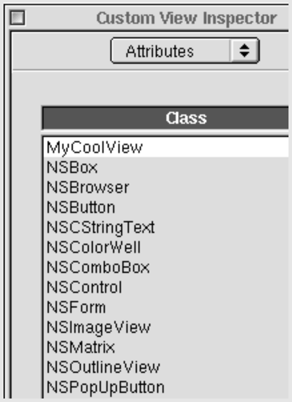
If you want an object that draws itself differently than any other Application Kit object, or responds to events in a special way, you should make a custom subclass of NSView. Your custom subclass should complete at least the steps outlined below.

Note: If you make a custom subclass of any class that inherits from NSView, and you want to do custom drawing or event handling, the basic procedure presented here still applies.

Interface Builder

1 Define a subclass of NSView in Interface Builder. Then generate header and implementation files.

2 Drag a CustomView object from the Views palette onto a window and resize it. Then, with the CustomView object still selected, choose the Custom Class display of the Inspector panel and select the custom class. Connect any outlets and actions.



Initializing Instances

3 Override the designated initializer, **initWithFrame:** to return an initialized instance of **self**. The argument of this method is the frame rectangle of the NSView, usually as set in Interface Builder (see step 2).

Handling Events

In the next section, you'll make a subclass of NSButtonCell that uniquely responds to mouse clicks. The way custom NSViews handle events is different. If you intend your custom NSView to respond to user actions you must do a couple of things:

4 Override **acceptsFirstResponder** to return YES if the NSView is to handle selections. (The default NSView behavior is to return NO.)

5 Override the desired NSResponder event methods (**mouseDown:, mouseDragged:**, **keyDown:**, etc.).

```
- (void)mouseDown:(NSEvent *)event {
    if (([event modifierFlags] &
        NSControlKeyMask){
        doSomething();
}
```

You can query the NSEvent argument for the location of the user action in the window, modifier keys pressed, character and key codes, and other information.

Drawing

When you send **display** to an NSView, its **drawRect:** method and each of its subview's **drawRect:** are invoked. This method is where an NSView renders its appearance.

6 Override **drawRect:**. The argument is usually the frame rectangle in which drawing is to occur. This tells the Window Server where the NSView's coordinate system is located. To draw the NSView, you can do one or more of the following:

- Composite an NSImage.
- Call Application Kit functions such as **NSRectFill()** and **NSFrameRect ()** (NSGraphics.h).
- Call C functions that correspond to single PostScript operations, such as **PSsetgray()** and **PSfill()**.
- Call custom drawing functions created with **pswrap**.

7 When state changes and you need to have the object redraw itself, invoke **setNeedsDisplay:** with an argument of YES.

See "A Short Guide to Drawing and Compositing" on page 188 for more information on drawing techniques and requirements.

1 **Update the current item with new values entered in the inspector.**

Implement **switchChecked:** to apply changes made through switches and other controls.

```
- (void)switchChecked:(id)sender
{
    long tmpSecs=0;
    int idx = 0;
    id doc = [[NSApp mainWindow] delegate];
    if (sender == inspNotifAMPM) {                                    A
        if ([inspNotifHour intValue]) {
            tmpSecs = ConvertTimeToSeconds([inspNotifHour intValue],
                [inspNotifMinute intValue],
                [[sender cellAtRow:0 column:1] state]);
            [currentItem setSecsUntilDue:tmpSecs];
            [[NSApp mainWindow] setDocumentEdited:YES];
            [doc updateMatrix];
        }
    } else if (sender == inspNotifSwitchMatrix) {                     B
        idx = [inspNotifSwitchMatrix selectedRow];
        tmpSecs = [currentItem secsUntilDue];
        switch(idx) {
          case 0:
            [currentItem setSecsUntilNotif:0];
            break;
          case 1:
            [currentItem setSecsUntilNotif:tmpSecs-(hrInSecs/4)];
            break;
          case 2:
            [currentItem setSecsUntilNotif:tmpSecs-hrInSecs];
            break;
          case 3:
            [currentItem setSecsUntilNotif:tmpSecs-dayInSecs];
            break;
          case 4:  // Other
            [currentItem setSecsUntilNotif:([inspNotifOtherHours intValue]
                * hrInSecs)];
            break;
          default:
            NSLog(@"Error in selectedRow");
            break;
        }
        [[NSApp mainWindow] setDocumentEdited:YES];
    } else if (sender == inspSchedComplete) {                         C
        [currentItem setItemStatus:complete];
        [[NSApp mainWindow] setDocumentEdited:YES];
        [doc updateMatrix];

    } else if (sender == inspSchedMatrix) {                           D
    } /* left as an exercise */
}
```

When users click a switch button on any inspector display, or when they click one of the AM-PM radio buttons, the **switchChecked:** method is invoked. This method works by evaluating the **sender** argument: the sending object.

A If **sender** is the radio-button matrix (AM-PM), gets the new time due by calling the utility function **ConvertTimeToSeconds()**, sets the current item to have this new value, marks the document as edited, and then sends **updateMatrix** to the document controller to have it display this new time.

B If **sender** is the "When to Notify" matrix, gets the index of the selected cell and the seconds until the item is due. It evaluates the first value in a switch statement and uses the second value to set the current item's new **secsUntilNotif** value. It also sets the window to indicate an edited document.

C If **sender** is the "Task Completed" switch, sets the status of the current item to "complete," sets the window to indicate an edited document, and has the document controller update its matrices.

D As before, implementation of this rescheduling block is left as a final exercise.

Since text fields are controls that send target/action messages, you could also have **switchChecked:** respond when data is entered in the fields. However, users might not press Return in a text field so you can't assume the action message will be sent. Therefore, it's better to rely upon delegation messages.

Update the current item if changes are made to the contents of text fields or the text object of the inspector panel.

```
- (void)textDidEndEditing:(NSNotification *)notif                    A
{
    if ([notif object] == inspNotes)
        [currentItem setNotes:[inspNotes string]];
        [[NSApp mainWindow] setDocumentEdited:YES];
}

- (void)controlTextDidEndEditing:(NSNotification *)notif
{
    long tmpSecs=0;
    if ([notif object] == inspNotifHour ||                           B
        [notif object] == inspNotifMinute) {
        tmpSecs = ConvertTimeToSeconds([inspNotifHour intValue],
            [inspNotifMinute intValue],
            [[inspNotifAMPM cellAtRow:0 column:1] state]);
        [currentItem setSecsUntilDue:tmpSecs];
        [[[NSApp mainWindow] delegate] updateMatrix];
        [[NSApp mainWindow] setDocumentEdited:YES];
    } else if ([notif object] == inspNotifOtherHours) {              C
        if ([inspNotifSwitchMatrix selectedRow] == 4) {
            [currentItem setSecsUntilNotif:([inspNotifOtherHours
                intValue] * hrInSecs)];
            [[NSApp mainWindow] setDocumentEdited:YES];
}
    } else if ([notif object] == inspSchedDate) {                    D
    } /* left as an exercise */
}
```

The **textDidEndEditing:** and **controlTextDidEndEditing:** notification messages are sent to the delegate (and all other observers) when the cursor leaves a text object or text field (respectively) after editing has occurred.

A After editing takes place in the "Notes" text object, this method is invoked, and it responds by resetting the **notes** instance variable of the ToDoItem with the contents of the text object.

B If the object behind the notification is the hour or minute field of the "Notifications" display, **controlTextDidEndEditing:** computes the new due time, sets the current item to have this new value, and then sends **updateMatrix** to the document controller to have it display this new time. (This code is almost the same as that for the AM-PM matrix in the **switchChecked:** method.)

C If the object behind the notification is the "Other...hours" text field in the "When to Notify" box, the method verifies that the "Other" switch is checked and, if it is, sets the ToDoItem with the new value.

D Here is another empty rescheduling block of code that you can fill out in a later exercise.

Now it's time to address two related problems in synchronizing displays of data. The first is the requirement for the inspector to display the ToDoItem currently selected in the document. In **ToDoDoc.m** write code that communicates this object to ToDoInspector through notification.

1 **Synchronize the items displayed in the document with the inspector.**

Open **ToDoDoc.m**.

Import **ToDoInspector.h**.

Add the code at right to the end of the **controlTextDidEndEditing:** method.

Post identical notifications in the other ToDoDoc methods listed in the table below.

In **ToDoDoc.h** declare as extern the string constant **ToDoItemChangedNotification**.

In **ToDoDoc.m**, declare and initialize the same constant.

```
    id curItem;
/* ... */
        if (curItem = [currentItems objectAtIndex:row]) {
        if (![curItem isKindOfClass:[ToDoItem class]])
            curItem = nil;
        [[NSNotificationCenter defaultCenter] postNotificationName:
            ToDoItemChangedNotification object:curItem
            userInfo:nil];
    }
```

The **controlTextDidEndEditing:** method is where ToDoItems are added, removed, or modified, so it's especially important here to let ToDoInspector know when there's a change in the current ToDoItem. The fragment of code above gets the current item (**row** holds the index of the selected row); if the returned object isn't a ToDoItem, **curItem** is set to **nil**. Then the code posts a ToDoItemChangedNotification, passing in **curItem** as the object related to the notification.

Post an identical notification in other ToDoDoc methods that select a ToDoItem *or* that require the removal of the currently displayed ToDoItem from the inspector's display. In methods of this second type, there is no need to get the current item because the **object** argument of the notification should always be **nil**. This argument is eventually passed to ToDoInspector's **updateInspector:**, to which **nil** means "clear the display."

Other Methods Posting Notifications to ToDoInspector	object: Argument
calendarMatrix:didChangeToDate:	nil
calendarMatrix:didChangeToMonth:year:	nil
windowShouldClose: (for both "Save" and "Close")	nil
selectionInMatrix:	current item or nil

1 **Open the inspector panel when users choose the Inspector command.**

Implement ToDoController's showInspector: method to load ToDoInspector.nib and make the inspector panel the key window.

1 **Update the document and inspector to display initial values.**

In ToDoDoc.m, implement **selectItem:**.

Invoke this method at the appropriate places (see table below).

The second data-synchronization problem involves the selection and display of initial values in the document and the inspector when the user:

- Opens the inspector
- Opens a document
- Selects a new day from the calendar

You must return to **ToDoDoc.m** to write code that implements this behavior.

```
- (void)selectItem:(int)item
{
    id thisItem = [currentItems objectAtIndex:item];
    [itemMatrix selectCellAtRow:item column:0];
    if (thisItem) {
        if (![thisItem isKindOfClass:[ToDoItem class]]) thisItem = nil;
        [[NSNotificationCenter defaultCenter]
            postNotificationName:ToDoItemChangedNotification
                           object:thisItem
                         userInfo:nil];
    }
}
```

The **selectItem:** method selects the text field identified in the argument and posts a notification to the inspector with the associated ToDoItem as argument (or **nil** if the text field is empty). Next, invoke **selectItem:** in these methods:

Method	Comment
calendarMatrix:didChangeToDate:	Make it the final message, with an argument of 0 (**ToDoDoc.m**).
openDoc:	Invoke after opening a document, with an argument of 0 (**ToDoController.m**)
showInspector:	Invoke after opening the inspector panel, passing in the index of the selected row in the document. (**ToDoController.m**). Hint: Get the current document by querying for the delegate of the main window, then obtain the selected row from this object.

Before You Go On

Exercise: Make ToDoInspector respond to the notification. Declare a notification method named **currentItemChanged:** and implement it to set the current item with the **object** value of the notification. Then, in **init** or **awakeFromNib**, add ToDoInspector as an observer of the ToDoItemChangedNotification, identifying **currentItemChanged:** as the method to be invoked.

1 **Set up the inspector when it is unarchived.**

In ToDoInspector.m, implement **awakeFromNib** as shown at right.

```
- (void)awakeFromNib
{
    [inspPopUp selectItemAtIndex:0];                          A
    [inspNotes setDelegate:self];

    [[notifView contentView] removeFromSuperview];            B
    notifView = [[notifView contentView] retain];
    [[reschedView contentView] removeFromSuperview];
    reschedView = [[reschedView contentView] retain];
    [[notesView contentView] removeFromSuperview];
    notesView = [[notesView contentView] retain];
    [inspectorViews release];
    [self newInspectorView:self];
}
```

ToDoInspector's **awakeFromNib** method performs some necessary "housekeeping" tasks for the ToDoInspector instance of the application.

A Makes the Notification pop-up display the start-up default, using the index of the "Notification" cell rather than its title to improve localization. Then it sets **self** to be the delegate of the text object.

B Each of the three inspector displays in the off-screen panel (**inspectorViews**) is the content view of an NSBox. This section of code extracts and retains each of those content views, reassigning each to its original NSBox instance variable in the process. This explicit retaining is necessary because, in **newInspectorView:**, each current content view is released when it's swapped out. Once all content views are retained, the code releases the off-screen window and invokes **newInspectorView:** to put up the default display.

The use of notifications to communicate changes in one object to another object in an application is a good design strategy because it removes the need for the objects to have specific knowledge of each other. It also makes the application more extensible, because any number of objects can also become observers of the changes. However, there is a way for ToDoDoc to locate ToDoInspector reliably using the various relationships established within the program framework. See page 197 to see how this is done.

A Short Guide to Drawing and Compositing

Besides responding to events, all objects that inherit from NSView can render themselves on the screen. They do this rendering through image composition and PostScript drawing.

NSViews draw themselves as an indirect result of receiving the **display** message (or a variant of **display**); this message is sent explicitly or through conditions that cause automatic display. The **display** message leads to the invocation of an NSView's **drawRect:** method and the **drawRect:** methods of all subviews of that NSView. The **drawRect:** method should contain all code needed to redraw the NSView completely.

An NSView can be automatically displayed when:

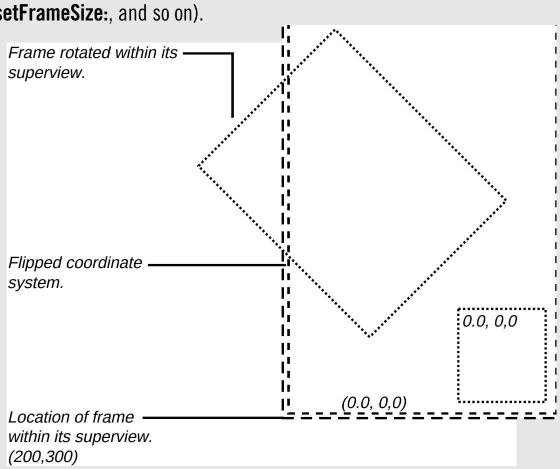
- Users scroll it (assuming it supports scrolling).
- Users resize or expose the NSView's window.
- The window receives a **display** message or is automatically updated.
- For some Application Kit objects, when an attribute changes.

An NSView represents a context within which PostScript drawing can take place. This context has three components:

- A rectangular frame within a window to which drawing is clipped
- A coordinate system
- The current PostScript graphics state

Frame and Bounds

An NSView's *frame* specifies the location and dimensions of the NSView in terms of the coordinate system of the NSView's superview. It is a rectangle that encloses the NSView. You can programmatically move, resize, and rotate the NSView by reference to its frame (**setFrameOrigin:**, **setFrameSize:**, and so on).



To draw efficiently, the NSView must have its frame rectangle translated into its own coordinate system. This translated rectangle, suitable for drawing, is called the *bounds*. The bounds rectangle usually specifies exactly the same area as the frame rectangle, but it specifies that area in a different coordinate system. In the default coordinate system, an NSView's bounds is the same as its frame, except that the point locating the frame becomes the origin of the bounds (x = 0.0, y = 0.0). The x- and y-axes of the default coordinate system run parallel to the sides of the frame so, for example, if you rotate the frame the default coordinate system rotates with it.

This relationship between frame and bounds has several implications important in drawing and compositing.

- Each NSView's coordinate system is a transformation of its superview's.
- Drawing instructions don't have to account for an NSView's location on the screen or its orientation.
- Changes in a superview's coordinate system are propagated to its subviews.

NSView allows you to flip coordinate systems (so the positive y-axis runs downward) and to otherwise alter coordinate systems.

Focusing

Before an NSView can draw it must *lock focus* to ensure that it draws in the correct window, place, and coordinate system. It locks focus by invoking NSView's **lockFocus** method. Focusing modifies the PostScript graphics state by:

- Making the NSView's window the current device
- Creating a clipping path around the NSView's frame
- Making the PostScript coordinate system match the NSView's coordinate system

After drawing, the NSView should unlock focus (**unlockFocus**).

PostScript Drawing

In Rhapsody, NSViews draw themselves by sending binary-encoded PostScript code to the Window Server. The Application Kit and the Display PostScript frameworks provide a number of C-language functions that send PostScript code to perform common drawing tasks. You can use these functions in combinations to accomplish fairly elaborate drawing.

The Application Kit has functions and constants, declared in **NSGraphics.h**, for (among other things):

- Drawing, filling, highlighting, clipping and erasing rectangles
- Drawing buttons, bezels, and bitmaps
- Computing window depth and related display information

You also call Yellow Box-compliant drawing routines defined in **dpsOpenStep.h**. These routines (such as **DPSDoUserPath()**) draw a specified path. In addition, you can call the functions declared in psops.h. These functions correspond to single PostScript operators, such as **PSsetgray()** and **PSfill()**.

You can also write and send your own custom PostScript code. The **pswrap** program converts PostScript code into C-language functions that you can call within your applications. It is an efficient way to send PostScript code to the Window Server. The following **pswrap** functions draw ovals.:

```
defines PDFramedOval(float x, y, w, h)
    matrix currentmatrix
    w h x y oval
    setmatrix stroke
endps

defines PSFilledOval float x, y, w, h)
    w h x y oval fill
endps
```

Compose the function in a file with a **.psw** extension and add it to the Other Source project "suitcase" in Project Builder. When you next build your project, Project Builder runs the **pswrap** program, generating an object file and a header file (matching the file name of the **.psw**) file, and links these into the application. To use the code, import the header file and call the function when you want to do the drawing:

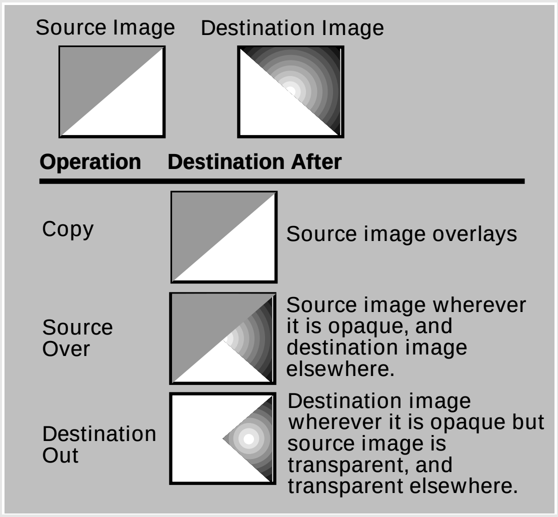
```
PSFilledOval(5.0, 5.0, 1.0, 1.0);
```

Compositing Images

The other technique NSViews use to render their appearance is image compositing. By compositing (with the SOVER operator) NSViews can simply display an image within their frame. You usually composite an image using NSImage's **compositeToPoint:operation:** (or a related method).

NSImage allows you to copy images into your user interface. It uses various subclasses of NSImageRep to store the multiple representations of the same image—color, grayscale, TIFF, EPS, and so on—and choosing the representation appropriate for a given type or display. NSImage can read image data from a bundle (including the application's main bundle), from the pasteboard, or from an NSData object.

Compositing allows you to do more than simply copy images. Compositing builds a new image by overlaying images that were previously drawn. It's like a photographer printing a picture from two negatives, one placed on top of the other. Various compositing operators (NSCompositingOperation, defined in **dpsOpenStep.h**) determine how the source and destination images merge.



You can achieve interesting effects with compositing when the initial images are drawn with partially transparent paint. (Transparency is specified by *coverage*, a indicator of paint opacity.) In a typical compositing operation, paint that's partially transparent won't completely cover the image it's placed on top of; some of the other image will show through. The more transparent the paint is, the more of the other image you'll see.

Overriding and Adding Behavior to a Class: An Example

Buttons in the Application Kit are two-state controls. They have two—and only two—states: 1 and 0 (often expressed as Boolean YES and NO, or ON and OFF). For the To Do application, a three-state button is preferable. You want the button to indicate, with an image, three possible states: not done (no image), done (an "X"), and deferred (a check mark). These states correspond to the possible states of a ToDoItem.

The ToDoCell class, which you will implement in this section, generates cells that behave as three-state buttons. These buttons also display the time an item is due.

1 **Add the cell images to the project to the project.**

Select the Images "suitcase."

Choose Add Files from the Project menu.

In the Add Images panel, navigate to the ToDo project directory of **/System/Developer/Examples/AppKit** and select file **X.tiff**.

Click OK.

Repeat the same steps for file **checkMark.tiff**, which is in the same location.

The superclass of ToDoCell is NSButtonCell. In creating ToDoCell you will add data and behavior to NSButtonCell, and you will override some existing behavior.

Why Choose NSButtonCell as Superclass?

ToDoCell's superclass is NSButtonCell. This choice prompts two questions:

- Why a button cell and not the button itself?
- Why this particular superclass?

NSCell defines state as an instance variable, and thus all cells inherit it. Cells instead of controls hold state information for reasons of efficiency—one control (a matrix) can manage a collection of cells, each cell with its own state setting. NSButton does provide methods for getting and setting state values, but it accesses the state value of the cell (usually NSButtonCell) that it contains.

NSButtonCell is ToDoCell's superclass because button cells already have much of the behavior you want. By virtue of inheritance from NSActionCell, button cells can hold target and action information. Button cells also have the unique capability to display an image and text simultaneously. These are all aspects of behavior needed for ToDoCell.

When you think that you need a specialized subclass of a Yellow Box class, you should first spend some time examining the header files and reference documentation on not only that class, but its superclasses and any "sibling" classes.

1 **Add header and implementation files to the project.**

Choose New in Project from the File menu.

In the New File In ToDo panel, select the Class suitcase, click Create header, type “ToDoCell” after Name, and click OK.

1 **Complete ToDoCell.h.**

Make the superclass NSButtonCell.

Add the instance-variable and method declarations shown at right.

Add the **enum** constants for state values (as shown).

```
enum ToDoButtonState {notDone=0, done, deferred} ToDoButtonState;

@interface ToDoCell : NSButtonCell
{
    ToDoButtonState triState;
    NSImage *doneImage, *deferredImage;
    NSDate *timeDue;
}
- (void)setTriState:(ToDoButtonState)newState;
- (ToDoButtonState)triState;
- (void)setTimeDue:(NSDate *)newTime;
- (NSDate *)timeDue;
@end
```

The **triState** instance variable will be assigned ToDoButtonState constants as values. The NSImage variables hold the “X” and check mark images that represent statuses of completed and deferred (that is, rescheduled for the next day). The **timeDue** instance variable carries the time the item is due as an NSDate; for display, this object will be converted to a string.

1 **Initialize the allocated ToDoCell instance (and deallocate it).**

Select ToDoCell.m in the project browser.

Implement **init** as shown at right.

Implement **dealloc**.

```
- (id)init
{
    NSString *path;
    [super initTextCell:@""];

    triState = notDone;
    [self setType:NSToggleButton];                                   A
    [self setImagePosition:NSImageLeft];
    [self setBezeled:YES];
    [self setFont:[NSFont userFontOfSize:12]];
    [self setAlignment:NSRightTextAlignment];
                                                                     B
    path = [[NSBundle mainBundle] pathForImageResource:@"X.tiff"];
    doneImage = [[NSImage alloc] initByReferencingFile:path];
    path = [[NSBundle mainBundle]
        pathForImageResource:@"checkMark.tiff"];
    deferredImage = [[NSImage alloc] initByReferencingFile:path];

    return self;
}
```

A Sets some superclass (NSButtonCell) attributes, such as button type, image and text position, font of text, and border.

B Through NSBundle’s **pathForImageResource:**, gets the pathname for the cell images and creates and stores the images using the pathname.

1 Implement the accessor methods related to state.

Write the methods that get and set the **triState** instance variable.

Override the superclass methods that get and set state.

```
- (void)setTriState:(ToDoButtonState)newState          A
{
    if (newState == deferred+1)
        triState = notDone;
    else
        triState = newState;
    [self TD_setImage:triState];
 }

- (ToDoButtonState)triState {return triState;}

- (void)setState:(int)val                              B
{
}

- (int)state                                           C
{
    if (triState == deferred)
        return (int)done;
    else
        return (int)triState;
}
```

Accessing state information is a dual-path task in ToDoCell. It involves not only setting and getting the new state instance variable, **triState**, but properly handling the inherited instance variable by overriding the superclass accessor methods for state.

A If the new value for **triState** is one greater than the limit (**deferred**), reset it to zero (**notDone**); otherwise, assign the value. The reason behind this logic is that (as you'll soon learn) when users click a ToDoCell, **setTriState:** is invoked with an argument one more than the current value. This way users can cycle through the three states of ToDoCell.

B Overrides **setState:** to be a null method. The reason for this override is that NSCell intervenes when a button is clicked, resetting state to zero (NO). This override nullifies that effect.

C Overrides **state** to return a reasonable value to client objects that invoke this accessor method.

1 **Set the cell image.**

Declare the private method **TD_setImage:**.

Implement the **TD_setImage:** method.

```
@interface ToDoCell (PrivateMethods)
- (void)TD_setImage:(ToDoButtonState)aState;        A
@end
/* ... */
- (void)TD_setImage:(ToDoButtonState)aState
{
    switch(aState) {                                  B
       case notDone: {
           [self setImage:nil];
           break;
       }
       case done: {
           [self setImage:doneImage];
           break;
       }
       case deferred: {
           [self setImage:deferredImage];
           break;
       }

     [(NSControl *)[self controlView] updateCell:self];   C
}
```

This portion of code handles the display of the cell's image by doing the following:

A In a category of ToDoCell in **ToDoCell.m**, it declares the private method **TD_setImage:**. Private methods are methods that you don't want clients of your object to invoke, and thus you don't "publish" them by declaring them in public header files. In this case, you don't want the image to be set independently from the cell's **triState** value.

B In a switch statement, evaluates the tri-state argument and sets the cell's image appropriately (**setImage:** is an NSButtonCell method).

C Sends **updateCell:** to the control view of the cell's control (a matrix) to force a re-draw of the cell.

1 **Track mouse clicks on a ToDoCell and reset state.**

Override two NSCell mouse-tracking methods as shown in this example.

```
- (BOOL)startTrackingAt:(NSPoint)startPoint inView:        A
  (NSView *)controlView
{
    return YES;
}

- (void)stopTracking:(NSPoint)lastPoint at:(NSPoint)stopPoint
  inView:(NSView *)controlView mouseIsUp:(BOOL)flag         B
{
    if (flag == YES) {
        [self setTriState:([self triState]+1)];
    }
}
```

When you create your own cell subclass, you might want to override some methods that are intrinsic to the behavior of the cell. Mouse-tracking methods, inherited from NSCell, are among these. You can override these methods to incorporate specialized behavior when the mouse clicks the cell or drags over it. ToDoCell overrides these methods to increment the value of **triState**.

A Overrides **startTrackingAt:inView:** to return YES, thus signalling to the control that the ToDoCell will track the mouse.

B Overrides **stopTracking:at:inView:mouseIsUp:** to evaluate flag and, if it's YES, to increment the **triState** instance variable. The **setTriState:** method "wraps" the incremented value to zero (**notDone**) if it is greater than 2 (**deferred**).

1 **Get and set the time due, displaying the time in the process.**

Implement setTimeDue: as shown in this example.

Implement **timeDue** to return the NSDate.

```
- (void)setTimeDue:(NSDate *)newTime
{
    if (timeDue)
        [timeDue autorelease];
    if (newTime) {
        timeDue = [newTime copy];
        [self setTitle:[timeDue descriptionWithCalendarFormat:
            @"%I:%M %p" timeZone:[NSTimeZone localTimeZone]
            locale:nil]];
    }
    else {
        timeDue = nil;
        [self setTitle:@"-->"];
    }
}
```

The **setTimeDue:** method is similar to other "set" accessor methods, except that it handles interpretation and display of the NSDate instance variable it stores. If **newTime** is a valid object, it uses **descriptionWithCalendarFormat:timeZone:locale:**,an NSDate method, to interpret and format the date object before displaying the result with

setTitle:. If **newTime** is **nil**, no due time has been specified, and so the method sets the title to “-->”.

You’ve now completed all code required for ToDoCell. However, you must now “install” instances of this class in the To Do interface.

1 **At launch time, create and install your custom cells in the matrix.**

Select **ToDoDoc.m** in the project browser.

Insert the code at right in **awakeFromNib**.

```
- (void)awakeFromNib
{
    int i;
/* ... */
    i = [[markMatrix cells] count];
    while (i--) {
        ToDoCell *aCell = [[ToDoCell alloc] init];
        [aCell setTarget:self];
        [aCell setAction:@selector(itemChecked:)];
        [markMatrix putCell:aCell atRow:i column:0];
        [aCell release];
    }
}
```

This block of code substitutes a ToDoCell for each cell in the left matrix (**markMatrix**) you created for the To Do interface. It creates a ToDoCell, sets its target and action message, then inserts it into the **markMatrix** by invoking NSMatrix’s **putCell:atRow:column:** method.

Finally, you must implement the action message sent when the matrix of ToDoCells is clicked. (This response to mouse-down is for objects external to ToDoCell, while the mouse-tracking response sets state internally.)

1 **Respond to mouse clicks on the matrix of ToDoCell’s.**

In **ToDoDoc.m**, implement itemChecked:.

```
- (void)itemChecked:sender
{
    int row = [sender selectedRow];
    ToDoCell *cell = [sender cellAtRow:row column:0];
    if (cell && [currentItems count]) {
        id item = [currentItems objectAtIndex:row];
        if (item && [item isKindOfClass:[ToDoItem class]]) {
            [item setItemStatus:[cell triState]];
            [[sender window] setDocumentEdited:YES];
        }
    }
}
```

This method gets the ToDoCell that was clicked and the object in the corresponding text field. If that object is a ToDoItem, the method updates its status to reflect the state of the ToDoCell. It then marks the window as containing an edited document.

Setting Up Timers

One of To Do's features is the capability for notifying users of items with impending due times. Users can specify various intervals before the due time for these notifications, which take the form of a message in an attention panel. In this section you will implement the notification feature of To Do. In the process you'll learn the basics of creating, setting, and responding to timers.

Here's how it works: Each ToDoItem with a "When to Notify" switch (other than "Do not notify") selected in the inspector panel—and hence has a positive **secsUntilNotif** value—has a timer set for it. If a user cancels a notification by selecting "Do not notify," the document controller invalidates the timer. When a timer fires, it invokes a method that displays the attention panel, selects the "Do not notify" switch, and sets **secsUntilNotif** to zero.

Implementing the timer feature takes place entirely in Project Builder, but extends across several classes.

1 **Add the timer as an instance variable to ToDoItem.**

Open **ToDoItem.h**.

Add the instance variable **itemTimer** of class NSTimer.

Write accessor methods to get and set this instance variable.

1 **Create and set the timer, or invalidate it.**

Open **ToDoDoc.m**.

Implement the **setTimerForItem:** method, which is shown at right.

```
- (void)setTimerForItem:(ToDoItem *)anItem
{
    NSDate *notifDate;
    NSTimer *aTimer;
    if ([anItem secsUntilNotif]) {                                  A
        notifDate = [[anItem day] addTimeInterval:[anItem
            secsUntilNotif]];
        aTimer = [NSTimer scheduledTimerWithTimeInterval:           B
                   [notifDate timeIntervalSinceNow]
            target:self
          selector:@selector(itemTimerFired:)
          userInfo:anItem
           repeats:NO];
        [anItem setItemTimer:aTimer];
    } else
        [[anItem itemTimer] invalidate];                            C
}
```

This method sets or invalidates a timer, depending on whether the ToDoItem passed in has a positive **secsUntilNotif** value.

A Tests the ToDoItem to see if it has a positive **secsUntilNotif** value and, if it has, composes the time the notification should be sent.

B Creates a timer and schedules it to fire at the right time, directing it to invoke **itemTimerFired:** when it fires. It also sets the timer in the ToDoItem.

C If the **secsUntilNotif** variable is zero, invalidates the item's timer.

1 **Respond to timers firing.**

Implement **itemTimerFired:** as shown at right.

```
- (void)itemTimerFired:(id)timer
{
    id anItem = [timer userInfo];
    ToDoInspector *inspController = [[[NSApp delegate]      A
        inspector] delegate];
    NSDate *dueDate = [[anItem day] addTimeInterval:         B
        [anItem secsUntilDue]];
    NSBeep();
    NSRunAlertPanel(@"To Do", @"%@ on %@", nil, nil, nil,
        [anItem itemName], [dueDate
        descriptionWithCalendarFormat:@"%b %d, %Y at %I:%M %p"
        timeZone:[NSTimeZone defaultTimeZone] locale:nil]);
    [anItem setSecsUntilNotif:0];
    [inspController resetNotifSwitch];
}
```

When a ToDoItem's timer goes off, it invokes the **itemTimerFired:** method (remember, you designated this method when you scheduled the timer).

A This method communicates with ToDoInspector in a more direct manner than notification. It gets the ToDoInspector object through this chain of association: the delegate of the application object is ToDoController, which holds the **id** of the inspector panel as an instance variable, and the delegate of the inspector panel is ToDoInspector.

B Composes the notification time (as an NSDate), beeps, and displays an attention panel specifying the name of a ToDoItem and the time it is due. It then sets the ToDoItem's **secsUntilNotif** instance variable to zero, and sends **resetNotifSwitch** to ToDoInspector to have it reset the "When to Notify" switches to "Do not Notify."

Before You Go On

Exercise: You haven't written ToDoInspector's **resetNotifSwitch** method yet, so do it now as an exercise. It should select the "Do not Notify" switch after turning off all switches in the matrix, and then force a redisplay of the switch matrix.

Next you must send **setTimerForItem:** at the right place and time, which is ToDoInspector, when the user alters a “When to Notify” value.

1 **Send the message that sets the timer at the right times.**

Open **ToDoInspector.m**.

In **switchChecked:**, insert the **setTimerForItem:** message at right *after* the switch statement evaluating which “When to Notify” switch was checked.

In **controlTextDidEndEditing:**, insert the same message at the end of the block related to the **inspNotifOtherHours** variable.

```
[[[NSApp mainWindow] delegate] setTimerForItem:currentItem];
```

Instead of archiving an item’s NSTimer, To Do re-creates and resets it when the application is launched.

1 **When the application is launched, reset item timers.**

Add the code shown at right to ToDoDoc’s **initWithFile:** method.

```
    if ([self activeDays]) {
         dayenum = [[self activeDays] keyEnumerator];
         while (itemDate = [dayenum nextObject]) {
             NSEnumerator *itemenum;
             ToDoItem *anItem=nil;
             NSArray *itemArray = [[self activeDays]
                objectForKey:itemDate];
             itemenum = [itemArray objectEnumerator];
             while ((anItem = [itemenum nextObject]) &&
                    [anItem isKindOfClass:[ToDoItem class]] &&
                    [anItem secsUntilNotif]) {
                 [self setTimerForItem:anItem];
             }
         }
     }
```

This block of code traverses the **activeDays** dictionary, evaluating each ToDoItem within the dictionary. If the ToDoItem has a positive **secsUntilNotif** value, it invokes **setTimerForItem:** to have a timer set for it.

Tick Tock Brrrring: Run Loops and Timers

A run loop—an instance of NSRunLoop—manages and processes sources of input. These sources include mouse and keyboard events from the window system, file descriptor, inter-thread connections (NSConnection), and timers (NSTimer).

Applications typically won't need to either create or explicitly manage NSRunLoop objects. When a thread is created, an NSRunLoop object is automatically created for it. The NSApplication object creates a default thread and therefore creates a default run loop.

NSTimer creates timer objects. A timer object waits until a certain time interval has elapsed and then fires, sending a specified message to a specified object. For example, you could create an NSTimer that periodically sends messages to an object, asking it to respond if an attribute changes.

NSTimer objects work in conjunction with NSRunLoop objects. NSRunLoops control loops that wait for input, and they use NSTimers to help determine the maximum amount of time they should wait. When the NSTimer's time limit has elapsed, the NSRunLoop fires the NSTimer (causing its message to be sent), then checks for new input.

Build, Run, and Extend the Application

Although you probably have been building the ToDo project frequently now, as it's been taking shape, build it one more time and check out what you've created. Go through the following sequence and observe To Do's behavior.

1. When you choose New from the Document menu, the application creates a new To Do document and selects the current day.

2. Enter a few items. Click a new day on the calendar and enter a few more items. Click the previous day and notice how the items you entered reappear.

3. Choose Inspector from the main menu. When the inspector appears, click an item and notice how the name and date of the item appears in the top part of the inspector. Enter due times for a couple items, and some associated notes. Note how the times, as you enter them, appear in the Status/Due column of the To Do document. Click among a few items again and note how the Notifications and Notes displays change.

4. Click a Status/Due button; the image toggles among the three states. Then, with an item that has a due time, select a notification time that has already passed. The application immediately displays an attention panel with a notification message. When you dismiss this panel, To Do sets the notification option to "Do not notify."

5. Click the document window and respond to the attention panel by clicking Save. In the Save panel, give the document a location and name. When the window has closed, chose Open from the Document menu and open the same document. Observe how the items you entered are redisplayed.

Optional Exercises

You should be able now to supplement the To Do application with other features and behaviors. Try some of the following suggestions.

Make Your Own Info Panel

Make your own Info panel. Define a method that responds to a click on the Info panel button by loading a nib file containing the panel. The owner of the panel can be the application controller. You can customize this panel however you wish. For instance, put the application icon in a toggled button (the main image) and make the alternate image a photo (yourself, your significant other, your dog). When users click the button, the image changes between the two.

Implement Application Preferences

Make a Preferences panel for the application, with a new controller object (or the application controller) as the owner of the nib file containing the panel. Follow what you've done for ToDoInspector, especially if the panel has multiple displays. Some ideas for Preferences: how long to keep expired ToDoItems before logging and purging them (see below); the default document to open upon launch; the default rescheduling interval (see below). Store and retrieve specified preferences as user defaults; for more information, see the NSUserDefaults specification.

Implement Rescheduling

ToDo's Inspector panel has a Rescheduling display that does almost nothing now. Implement the capability for rescheduling items by the period specified.

Implement Logging and Purging

After certain period (set via Preferences), append expired ToDoItems (as formatted text) to a log, and expunge the ToDoItems from the application.

A

What You'll Learn

Characteristics of an object-oriented program

What an object is

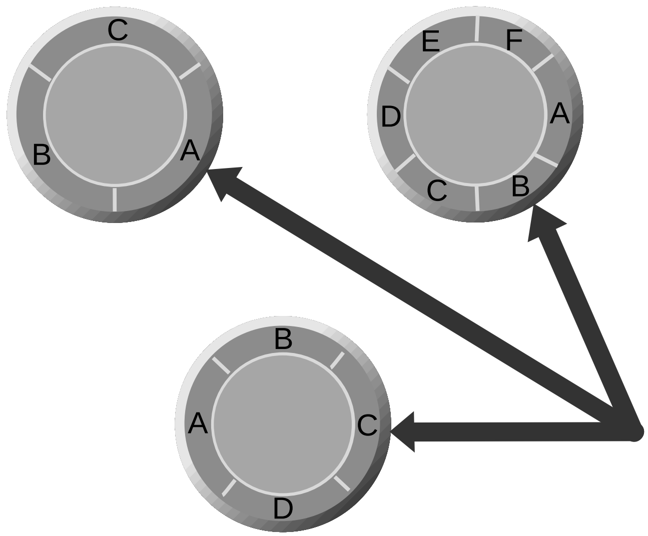
Encapsulation

Messages

What a class is

Inheritance

Categories and protocols

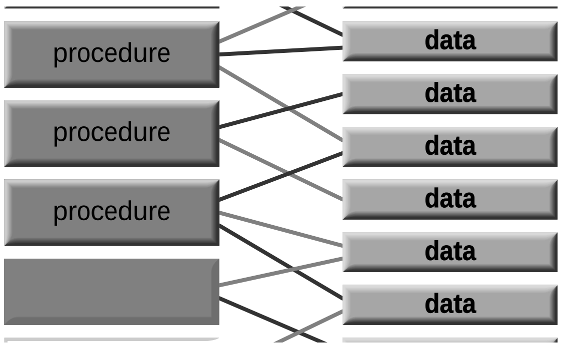


Appendix A
Object-Oriented Programming

You can't get far in Rhapsody or Yellow Box for Windows development without a grasp of the basic concepts of object-oriented programming. For those new to this approach to programming, it might seem strange at first glance, but a common reaction after learning a bit more is "Yes, of course." This appendix presents an overview of object-oriented programming from the particular perspective of Objective-C.

"Object-oriented programming" has become one of the premier buzzwords in the computer industry. To understand why, it's important to cut through the hype and focus on the problem that motivated the object-oriented approach.

In classic *procedural programming* (used with COBOL, Fortran, C, and other languages), programs are made of two fundamental components: *data* and *code*. The data represents what the user needs to manipulate, while the code does the manipulation. To improve project management and maintenance, procedural programs compartmentalize code into *procedures*. However, much of the data is global, and each procedure may manipulate any part of that global data directly.



With the procedural approach, the network of interaction between procedures and data becomes increasingly complex as an application grows. Inevitably, the interrelationships become a hard-to-maintain tangle—spaghetti code. A simple change in a data structure can affect many procedures, many lines of code—a nightmare for those who must maintain and enhance applications. Procedural programming also leads to nasty, hard-to-find bugs in which one function inadvertently changes data that another function relies on.

Objects change all that.

Objects

An object is a self-contained programmatic unit that combines data and the procedures that operate on that data. In the Objective-C language, an object's data comprises its *instance variables*, and its procedures, the functions that affect or make use of the data, are known as *methods*.

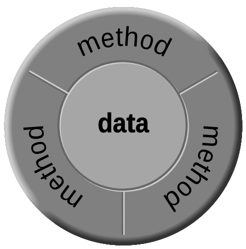
Like objects in the physical world, objects in a program have identifying characteristics and behavior. Often programmatic objects are modeled on real objects. For example, an object such as a button has an analog in the buttons on control devices, such as stereo equipment and telephones. A button object includes the data and code to generate an appearance on the screen that simulates a "real" button and to respond in a familiar way to user actions.

A button object highlights its on-screen representation when the user clicks it.

Encapsulation

Just as procedures compartmentalize code, objects compartmentalize both code *and* data. This results in *data encapsulation*, effectively surrounding data with the procedures for manipulating that data.

Typically, an object is regarded as a "black box," meaning that a program never directly accesses an object's variables. Indeed, a program shouldn't even need to know what variables an object has in order to perform its functions. Instead, the program accesses the object only through its methods. In a sense, the methods surround the data, not only shielding an object's instance variables but mediating access to them:



Objects are the basic building blocks of Objective-C applications. By representing a responsibility in the problem domain, each object encapsulates a particular area of functionality that the program needs. The object's methods provide the interface to this functionality. For example, an object representing a database record both stores data and provides well-defined ways to access that data.

Using this *modularity*, object-oriented programs can be divided into distinct objects for specific data and specific tasks. Programming teams can easily parcel out areas of responsibility among individual members, agreeing on interfaces to the distinct objects while implementing data structures and code in the most efficient way for their specific area of functionality.

Messages

To invoke one of the object's methods you send it a *message*. A message requests an object to perform some functionality or to return a value. In Objective-C, a message expression is enclosed in square brackets, like this:

$$\underbrace{\texttt{celsius}}_{\textit{returned value}} \texttt{ = [}\underbrace{\texttt{converter}}_{\textit{receiver}}\ \underbrace{\texttt{convertTemp:}}_{\textit{method name}}\underbrace{\texttt{fahrenheit]}}_{\textit{argument}}$$

In this example **converter** is the *receiver*, the object that receives the message. Everything to the right of this term is the message itself; it consists of a method name and any arguments the method requires. The message received by **converter** tells it to convert a temperature from Fahrenheit to Celsius and return that value.

In Objective-C, every message argument is identified with a label. Arguments follow colon-terminated *keywords*, which are considered part of the method name. One argument per keyword is allowed. If a method has more than one argument—as NSString's **rangeOfString:options:** method does, for example—the name is broken apart to accept the arguments:

```
range = [string rangeOfString:@"Rhapsody" options:NSLiteralSearch];
```

Often, but not always, messages return values to the sender of the message. Returned values must be received in a variable of an appropriate type. In the above example, the variable **range** must be of type NSRange. Messages that return values can be *nested*, especially if those returned values are objects. By enclosing one message expression within another, you can use a returned value as an argument or as a receiver without having to declare a variable for it.

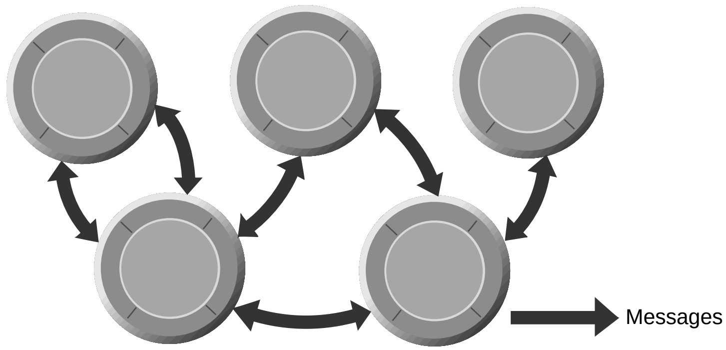
```
newString = [stringOne stringByAppendingString:
    [substringFromRange:[stringTwo rangeOfString:
    @"Rhapsody" at:NSAnchoredSearch]]];
```

The above message nests two other messages, each of which returns a value used as an argument. The inmost message expression is resolved first, then the next nested message expression is resolved, then the third message is sent and a value is returned to **newString**.

An Object-Oriented Program

Object-oriented programming is more than just another way of organizing data and functions. It permits application programmers to conceive and construct solutions to complex programs using a model that resembles—much more so than traditional programs—the way we organize the world around us. The object-oriented model for program structure simplifies problem resolution by clarifying roles and relationships.

You can think of an object-oriented program as a network of objects with well-defined behavior and characteristics, objects that interact through messages.



Different objects in the network play different roles. Some correspond to graphical elements in the user interface. The elements that you can drag from an Interface Builder palette are all objects. In an application, each window is represented by a separate object, as is each button, menu item, or display of text.

Applications also assign to objects functionality that isn't directly apparent in the interface, giving each object a different area of responsibility. Some of these objects might perform very specific computational tasks while others might manage the display and transfer of data, mediating the interaction between user-interface objects and computational objects.

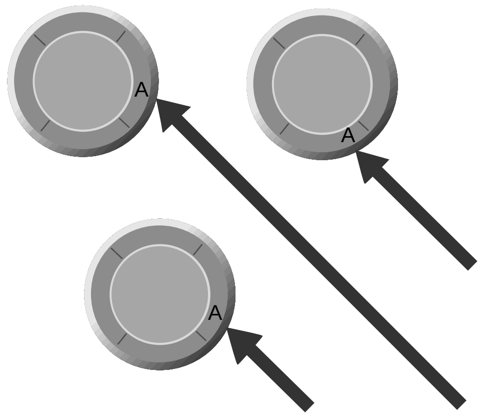
Once you've defined your objects, creating a program is largely a matter of "hooking up" these objects: creating the connections that objects will use to communicate with each other.

Polymorphism and Dynamic Binding

Although the purpose of a message is to invoke a method, a message isn't the same as a function call. An object "knows about" only those methods that were defined for it or that it inherits. It can't confuse its methods with another object's methods, even if the methods are identically named.

Each object is a self-contained unit, with its own name space (an name space being an area of the program where it is uniquely recognized by name). Just as local variables within a C function are isolated from other parts of a program, so too are the variables and methods of an object. Thus if two different kinds of objects have the same names for their methods, both objects could receive the same message, but each would respond to it differently. The ability of one message to cause different behavior in different receivers is referred to as *polymorphism*.

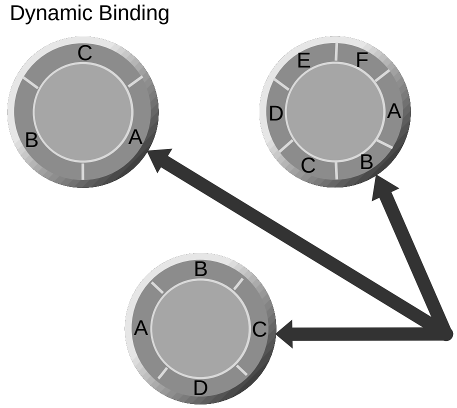
Polymorphism



The advantage polymorphism brings to application developers is significant. It helps improve program flexibility while maintaining code simplicity. You can write code that might have an effect on a variety of objects without having to know at the time you write the code what objects they might be. For example, most user-interface objects respond to the message **display**; you can send **display** to any of these objects in your interface and it will draw itself, in its own way.

Dynamic binding is perhaps even more useful than polymorphism. It means both the object receiving a message and the message that an object receives can be set within your program as it runs. This is particularly important in a graphical, user-driven environment, where one user command—say, Copy or Paste—may apply to any number of user-interface objects.

The example of **display** highlights the role of inheritance in polymorphism: a subclass often implements an identically named method (that is, *overrides* the method) of its superclass to achieve more specialized behavior. See the following section, "Classes," for details.



In dynamic binding, a run-time process finds the method implementation appropriate for the receiver of the message; it then invokes this implementation and passes it the receiver's data structure. This mechanism makes it easier to structure programs that respond to selections and actions chosen by users at run time. For example, either or both parts of a message expression—the receiver and the method name—can be variables whose values are determined by user actions. A simple message expression can deliver a Cut, Copy, or Paste menu command to whatever object controls the current selection.

Dynamic binding even enables applications to deal with new kinds of objects, ones that were not envisioned when the application itself was built. For example, it lets Interface Builder send messages to objects such as EOModeler when it is loaded into the application by means of custom palettes.

Polymorphism and dynamic binding depend on two other features: *dynamic typing* and *introspection*. The Objective-C language allows you to identify objects *generically* with the data type of **id**. This type defines a pointer to an object and its data structure (that is, instance variables) which, by inheritance from the root class NSObject, include a pointer to the object's class. What this means is that you don't have to type objects strictly by class in your code: the class for the object can be determined at run time through introspection.

Introspection means that an object, even one typed as **id**, can reveal its class and divulge other characteristics at run time. Several introspection methods allow you to ascertain the inheritance relationships of an object, the methods it responds to, and the protocols that it conforms to.

Classes

Some of the objects networked together in an applications are of different kinds, and some might be of the same kind. Objects of the same kind belong to the same *class*. A class is a programmatic entity that creates *instances* of itself—objects. A class defines the structure and interface of its instances and specifies their behavior.

When you want a new kind of object, you define a new class. You can think of a class definition as a type definition for a kind of object. It specifies the data structure that all objects belonging to the class will have and the methods they will use to respond to messages. Any number of objects can be created from a single class definition. In this sense, a class is like a factory for a particular kind of object.

In terms of lines of code, an object-oriented program consists mainly of class definitions. The objects the program will use to do its work are created at run time from class definitions (or, if pre-built with Interface Builder, are loaded at run time from the files where they are stored).

A class is more than just an object "factory," however. It can be assigned methods and receive messages just as an object can. As such it acts as a *class object*.

Object Creation

One of the primary functions of a class is to create new objects of the type the class defines. For example, the NSButton class creates new NSButton objects and the NSArray class creates new NSArrays. Objects are created at run time in a two-step process that first allocates memory for the instance variables of the new object and then initializes those variables. The following code creates a new Country object:

```
id newCountry = [[Country alloc] init];
```

The receiver for the **alloc** message is a class (the Country class from the Travel Advisor tutorial). The **alloc** method dynamically allocates memory for a new instance of the receiving class and returns the new object. The receiver for the **init** message is the new Country object that was dynamically allocated by **alloc**. Once allocated and initialized, this new record is assigned to the variable **newCountry**.

Note: You can create objects in your code with the **alloc** and **init** methods described here. But when you define a class in Interface Builder, that class definition is stored in a nib file. When an application loads that nib file, Interface Builder causes an instance of that class to be created.

After being allocated and initialized, a new object is a fully functional member of its class with its own set of variables. The **newCountry** object has all the behavior of any Country object, so it can receive messages, store values in its instance variables, and do all the other things a Country object does. If you need other Country objects, you create them in the same way from the same class definition.

Objects can be typed as **id**, as in the above example, or can be more restrictively typed, based on their class. Here, **newCountry** is typed as a Country object:

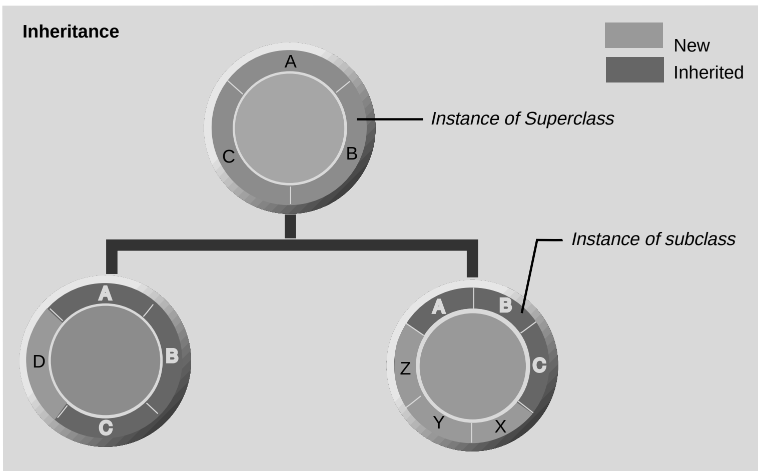
```
Country *newCountry = [[Country alloc] init];
```

The more restrictive typing by class enables the compiler to perform type-checking in assignment statements.

Inheritance

Inheritance is one of the most powerful aspects of object-oriented programming. Just as people inherit traits from their forbearers, instances of a class inherit attributes and behavior from that class's "ancestors." An object's total complement of instance variables and methods derives not only from the class that creates it, but from all the classes that class inherits from.

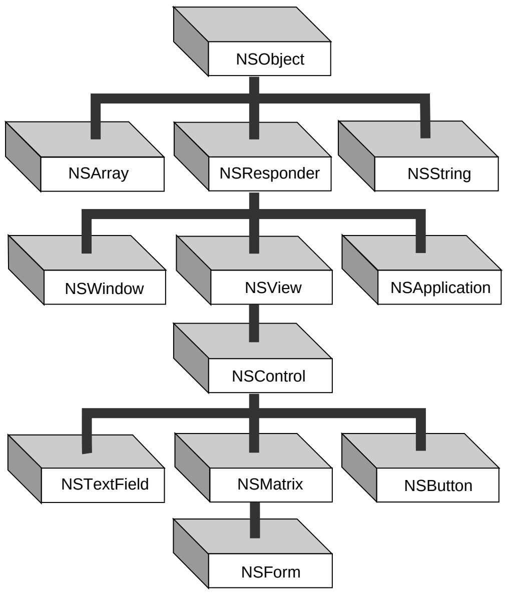
Because of inheritance, an Objective-C class definition doesn't have to specify every method and variable. If there's a class that does almost everything you want, but you need some additional features, you can define a new class that inherits from the existing class. The new class is called a *subclass* of the original class; the class it inherits from its *superclass*.



Creating a new class is often a matter of specialization. Since the new class inherits all its superclass's behavior, you don't need to re-implement the things that work as you want them to. The subclass merely extends the inherited behavior by adding new methods and any variables needed to support the additional methods. All the methods and variables defined for—or inherited by—the superclass are inherited by the subclass. A subclass can also alter superclass behavior by *overriding* an inherited method, re-implementing the method to achieve a behavior different from the superclass's implementation. (The technique for doing this is discussed later.)

The Class Hierarchy and the Root Class

A class can have any number of subclasses, but only one superclass. This means that classes are arranged in a branching hierarchy, with one class at the top—the *root class*—that has no superclass:



Part of the Yellow Box class hierarchy.

NSObject is the root class of this hierarchy, as it is of most Objective-C class hierarchies. From NSObject, other classes inherit the basic functionality that makes messaging work, enables objects to work together, and otherwise invests objects with the ability to behave as objects. Among other things, the root class creates a framework for the creation, initialization, deallocation, introspection, and storage of objects.

Note: Other root classes are possible. In fact, Distributed Objects makes use of another root class, NSProxy.

As noted earlier, you often create a subclass of another class because that superclass provides most, but not all, the behavior you require. But a subclass can have its own unique purpose that does not build on the role of an existing class. To define a new class that doesn't need to inherit any special behavior other than the default behavior of objects, you make it a subclass of the NSObject class. Subclasses of NSObject, because of their general-purpose nature as objects, are very common in Rhapsody applications. They often perform computational or application-specific functions.

Advantages of Inheritance

Inheritance makes it easy to bundle functionality common to a group of classes into a single class definition. For example, every object that draws on the screen—whether it draws an image of a button, a slider, a text display, or a graph of points—must keep track of which window it draws in and where in the window it draws. It must also know when it's appropriate to draw and when to respond to a user action. The code that handles all these details is part of a single class definition (the NSView class in the Application Kit). The specific work of drawing a button, a slider, or a text display can then be entrusted to a subclass.

This bundling of functionality both simplifies the organization of the code that needs to be written for an application and makes it easier to define objects that do complicated things. Each subclass need only implement the things it does differently from its superclass; there's no need to reimplement anything that's already been done.

What's more, hierarchical design assures more robust code. By building on a widely used, well-tested class such as NSView, a subclass inherits a proven foundation of functionality. Because the new code for a subclass is limited to implementing unique behavior, it's easier to test and debug that code.

Any class can be the superclass for a new subclass. Thus inheritance makes every class easily extensible—those provided by the Yellow Box frameworks, those you create, and those offered by third-party vendors.

Defining a Class

You define classes in two parts: One part declares the instance variables and the interface, principally the methods that can be invoked by messages sent to objects belonging to the class, and the other part actually implements those methods. The interface is public. The implementation is private, and can change without affecting the interface or the way the class is used.

The basic procedure for defining a class (using Interface Builder) is covered in the Currency Converter tutorial. However, here is a supplemental list of conventions and other points to remember when you define a class:

- The public interface for a class is usually declared in a header file (with an **.h** extension), the name of which is the name of the class. This header file can be imported into any program that makes use of the class.
- The code implementing a class is usually in a file taking the name of the class and having an extension of **.m**. This code must be present—in the form of a framework, dynamic shared library, static library, or the implementation file itself—when the project containing the class is compiled.
- Method declarations and implementations must begin with a minus (**–**) sign or a plus (+) sign. The dash indicates that these methods are used by instances of the class; a **+** sign precedes methods that the class object itself uses.
- Method definitions are much like function definitions. Note that methods not only respond to messages, they often initiate messages of their own—just as one function might call another.
- In a method implementation you can refer directly to an object's instance variables, as long as that object belongs to the class the method is defined in. There's no extra syntax for accessing variables or passing the object's data structure. The language keeps all that hidden.
- A method can also refer to the receiving object as **self**. This variable makes it possible for an object, in its method definitions, to send messages to itself.

Overriding a Method

A subclass can not only add new methods to the ones it inherits, it can also replace an inherited method with a new implementation. No special syntax is required; all you do is reimplement the method.

Overriding methods doesn't alter the set of messages that an object can receive; it alters the method implementations that will be used to respond to those messages. As mentioned earlier, this ability of each class to implement its own version of a method is referred to as polymorphism.

It's also possible to extend an inherited method, rather than replace it outright. To do this you override the method but invoke the superclass's same method in the new implementation. This invocation occurs with a message to **super**, which is a special receiver in the Objective-C language. The term **super** indicates that an inherited method should be performed, rather than one defined in the current class.

The Yellow Box Frameworks

When you write an object-oriented program, you rarely do it from scratch. There are almost always class definitions available that you can use. All you need are the class interface files, a library or framework with compiled versions of the class implementations, and some documentation. The task is to fit your pieces with the pieces that are already provided. As you'll realize after awhile, much of the task of writing object-oriented programs is simply implementing methods that respond to system-generated messages.

Categories and Protocols

In addition to subclassing, you can expand an object and make it fit with other classes using two Objective-C mechanisms: categories and protocols.

Categories provide a way to extend classes defined by other implementors—for example, you can add methods to the classes defined in the Yellow Box frameworks. The added methods are inherited by subclasses and are indistinguishable at run time from the original methods of the class. Categories can also be used as a way to distribute the implementation of a class into groups of related methods and to simplify the management of large classes where more than one developer is responsible for components of the code.

Protocols provide a way to declare groups of methods independent of a specific class—methods which any class, and perhaps many classes, might implement. Protocols declare interfaces to objects, leaving the programmer free to choose the implementation most appropriate for a specific class. Protocols free method declarations from dependency on the class hierarchy, so they can be used in ways that subclasses and categories cannot. They allow objects of any class to communicate with each other for a specific purpose.

The Rhapsody APIs provide a number of protocols. For example, the spell-checking protocols and the object-dragging protocols enable other developers to seamlessly integrate their spell-checking and object-dragging implementations into an existing system.

Programming Tools Resources Appendix B

B

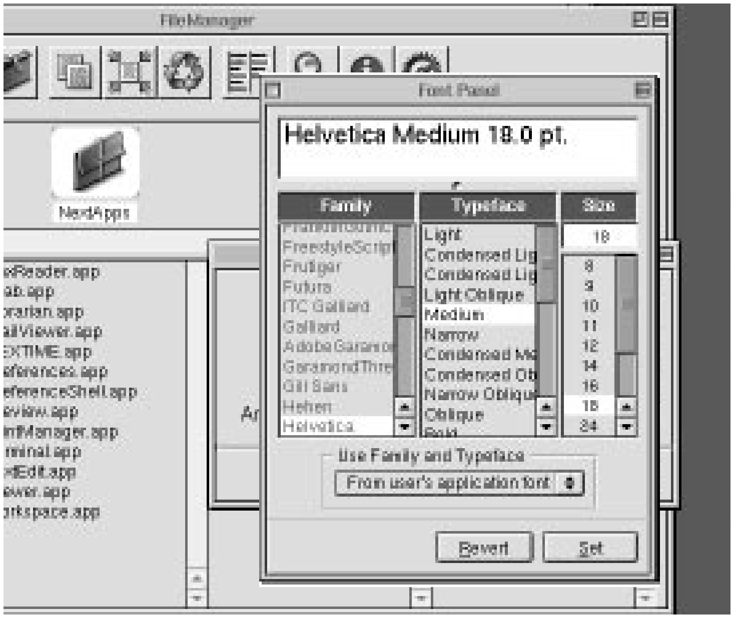
What You'll Learn

Secondary development applications

Other OpenStep frameworks

Useful command-line tools

Other programming resources



Appendix B
Programming Tools and Resources

There is more to the Apple development environment than just Project Builder and Interface Builder. This appendix describes some of the other applications, frameworks, command-line utilities, and other resources that are available to programmers.

Other Development Applications

The Apple development environment includes applications other than Project Builder and Interface Builder. Except where noted, these applications are installed in **/System/Developer/Apps**.

Name	Description
FileMerge	Visually compares the contents of two files or two directories. You can use FileMerge, for example, to determine the differences between versions of the same source code file or between two project directories. You can also use it to merge changes.
MallocDebug	Measures the dynamic-memory usage of applications, finds memory leaks, analyzes all allocated memory in an application, and measures the memory allocated since a given time.
Yap	A utility for editing and previewing PostScript code.
Sampler	Analyzes performance problems with your application by sampling the call stack of your program over a period. (In /System/Developer/Demos)

Other Installed Frameworks

A framework contains a dynamic shared library, related header files, and resources (including nib files, images, sounds, documentation, and localized strings) used by the library. All frameworks are installed in **/System/Library/Frameworks**. The Apple development environment provides these other frameworks besides the Application Kit, Display PostScript, and the Foundation frameworks:

Name	Description
System	Operating-system and low-level Objective-C run time APIs
SoundKit	Sound recording, playback, and editing capabilities.
InterfaceBuilder	Creation of custom static (compiled) palettes for use in Interface Builder
ProjectBuilder	Creation of custom project types, source-code management (SCM) adaptors, and other Project Builder extension bundles.

Useful Command-Line Tools

Apple has created or modified several tools for compilation, debugging, performance analysis, and so on. The following table lists some of the more useful of these tools. You can get further information using the man pages system.

Name	Description	Location
cc	Compiles C, Objective-C, C++, and Objective-C++ source code files.	/bin
gdb	Source-level symbolic debugger for C, extended by Apple to support Objective-C, C++, Mach, Windows NT, and (by late 1996) Windows 95.	/bin
gnumake	Utility for making programming projects.	/bin
as	Assembler; translates assembly code into object code.	/bin
defaults	Reads, writes, searches, and deletes user defaults. The defaults system records user preferences that persist when the application isn't running. When users specify defaults in an application's Preferences panel, NSUserDefaults methods are used to write the defaults.	/usr/bin
pswrap	Creates C functions that "wrap" PostScript code and send it to the Window Server for interpretation.	/usr/bin
nibTool	Reads the contents of an Interface Builder nib file. Prints classes, the hierarchy, objects, connections, and localizable strings.	/usr/bin
libtool	Creates static or dynamic libraries from specified object bin files for one or multiple architectures.	
otool	Displays specified parts of object files or libraries.	/bin
nm	Displays the symbol table, in whole or in part, of the specified object file or files.	/bin
AnalyzeAllocation	Analyzes program memory allocation.	/usr/bin
fixPrecomps	Creates or refreshes a precompiled header file for each of the major frameworks.	/usr/bin
strip	Removes or modifies the symbol table attached to assembled and linked output.	/bin
lipo	Creates, lists, and manipulates multi-architecture object files	/bin

Other Programming Resources

You can find programming resources—such as fonts, sounds, and palettes—in various subdirectories of **/System/Library.**

Name	Comments
SystemResources	Character-set information and location of headers for automatic precompilation (fixPrecomps)
Colors	Bundles containing the default set of color binaries for the Colors panel
Fonts	Default set of system fonts, including AFM, bitmap, and outline versions
PS2Resources	PostScript files containing calibrated color space and color rendering, printing halftones, and gray-shading patterns
Rulebooks	Glyph generators for various string encodings
Sounds	Default sound files (".snd") such as Cricket, Ping, and Rooster

Normally these resources are accessed via Application Kit APIs. Be careful about having dependencies on these resources in your code since the location and format of these resources might change in future releases.

A

acceptsFirstResponder 181
accessing
- data 84
- information 56–??

accessor method 49, 85, 86, 87, 192
- implementing 122
- retaining object 176

action 38, 41, 80, 99, 130, 182
- connecting 46
- defining 40, 43
- implementing 137
- setting programmatically 133

adding
- action 43
- application icon 75
- menu item 72
- outlet 43
- submenu 72

addObject: 159
addTimeInterval: 161
alignment 33
- of text 30

Alignment command 33
alloc 112, 210
AnalyzeAllocation command 221
animation 37
Apple events 9
application 157
- attributes 75, 122
- behavior 36
- creating 22
- design 187
- icon 75
- launch behavior 150
- multi-document 115, 138, 139
- NSApplication 168
- possibilities 37
- resources 126, 139
- standard features 36
- start-up routine 122

application controller 116, 126, 138, 139, 199
application controller, *See also* controller object
Application Kit 8, 62, 149, 169, 181, 188
application wrapper 53, 106, 122, 126
applicationShouldTerminate: 106, 150, 157
architecture 53
archiveRootObject:toFile: 106, 147
archiving 80, 86, 104, 106, 124, 147
argument 206
array 158
arrayWithObject: 145
ASCII 83
assembler 221
assigning the class 171
attention panel 197
attribute,setting 172
Attributes display 30
autorelease
- mechanism 88
- pool 88, 89

autorelease 87, 88, 89, 90, 112
auxiliary nib file 126, 139
awakeFromNib 52, 133, 164

B

background color 30
base coordinate system 27, 148
Blue Box 9
- communication with Yellow Box 9

bounds 188
box object 32, 67, 172
browser 69
Build panel 54
building a project 53, 54
- and errors 55

bundle 106, 126
- accessing resources 191
- loadable 126
- main 126

button 31, 95

and images 71, 130
custom 190
state 190
switch 65, 179
types 65

C

C language 53
C++ 53
calendar format 133
category 131, 216
cc 53, 221
cell 41, 107
enabling and disabling 135
highlighing 135
installing 195
prototype 132
setting state 193
setting title 135
cellAtIndex: 92, 94
cellAtRow:column: 179
cellWithTag: 137
class 39
adding to project 191
and object creation 210
assigning in Interface Builder 171
creating 212, 213
definition 128, 210, 213
principal 122
relation to object 39
reusing 77
specifying 38, 76
testing membership in 123, 160
class hierarchy 212
class method 49
class object 210
Close command 146
closing a document 146
coding 104
collection classes 86
color 222
Color panel 30
column identifier 68
columns, of objects 33
compare: 97
compiler 53
compositeToPoint:operation: 189
compositing images 181, 188, 189
connecting objects 45
direction of 41
Connections display 34, 45
containsObject: 177
content area 152, 153
content view 27, 152, 153, 177
box 172, 173
replacing 177
contents of nib file 25
contentView 153, 187
context-sensitive Help 56
control object 38, 41, 107
control:isValidObject: 187
controller object 20, 63
application 116
document 116
controlTextDidChange: 149
controlTextDidEndEditing: 159–161, 184, 185
coordinate system 27, 148, 153, 188
flipping 188
copy 87, 89, 112
copying objects 29, 123
and reference count 89
copyWithZone: 123
core operating system 8
core program framework 152–153, 187
coverage 189
creating
class 38
custom view object 181
document 143
form 66
object 210
panel 172

currentEvent 153
custom palette 209
custom view 117
  and Interface Builder 129
customizing menus 72
CustomView object 181

D

data
  at port 168
  mediating 94
  serializing 104
  storage 84
  synchronizing displays of 185, 186
data encapsulation 29, 205
data source 63, 68, 97
DataViews palette 68, 73
date 132
date and time 133, 137
  creating object 134
  formatting 134
date formatter 174
dateWithYear:month:day:hour:minute:second:timeZone: 137
day of the week 134
dealloc 89, 90, 108, 124
deallocation 84, 85, 88–89, 90, 108, 112, 124, 212
debugger 112
debugging 124
declaration 49, 142
  method 50, 120
deep copy 123
defaults 221
delegate 41, 80, 100, 106, 136, 139, 152, 153, 162, 169
  method implemented by 68, 116, 149, 183
delegate 153
delegation 63, 100, 157
delimiter checking 93
description 124
descriptionWithCalendarFormat:timeZone:locale: 162
design,hierarchical 213
determining class membership 209
dictionary 63, 106
display 177, 181, 188
Display PostScript 27, 122, 189
  framework 8
displays, synchronizing 186
document 115, 139
  and nib file 139
  closing 146
  creating 143
  default 150
  icon 118
  initial values 186
  management 37
  marking as edited 147, 149
  opening 145
  saving 146, 156, 157
  setting type of 118
document controller 116, 126, 138, 139, 140
document controller, *See also* controller object
Document menu 138
  Interface Builder 116
document type 118
documentation 56
  reference 56
drag-and-drop 28, 37
drawing 37, 152, 153, 181, 188–189
  functions 189
drawRect: 181, 188
duplicating object 29
dynamic binding 49, 97, 208–209
dynamic loading 126
dynamic typing 40, 49, 209

E

editable text 30
Emacs key bindings 93
enableDoubleReleaseCheck: 112
encapsulation 29, 205
encodeObject: 91
encodeValueOfObjCType:at: 91

encodeWithCoder: 86, 91, 104, 124
Enterprise Objects Framework 68
entity object 89
enum constant 92
EOModeler 209
event 27, 168–169, 198
  and custom NSView 181
  dispatching 152, 169
  handling 152, 181
  keyboard 169
  message 152
event cycle 168–169
event queue 169
extensibility 213
extension, file 118

F

fat files 53
field,formatting 187
file
  extension 118
  management 37
  opening 145
  saving 146–147
  type 145
file descriptor 168, 198
file formats 8
file package 53
File's Owner 79, 126
FileMerge application 220
finding information 56
first responder 152, 169
firstResponder 153
focusing 188
font 222
  setting 137
Font panel 31
Font submenu 36
form 66
formatter 37, 107
  applying in Interface Builder 73
  date 174
  number 73
  setting 187
Foundation framework 8, 84
frame 188
framework 53, 126, 220
  Foundation 84
frameworks
  core 8
function 125

G

gdb 112, 124, 221
generating
  instances 78
  source-code files 48, 82
generating code files 119
gnumake 221
Graphics APIs 8
grid, aligning on 33
grouping methods 216
grouping objects 67

H

header file 48, 50, 214
Help 37, 56, 96, 138
hierarchical data 69
hierarchy
  of classes 212

I

icon
  application 75
  document 118
icon mode (Interface Builder) 78
id 38, 209
identifier, column 68
image
  adding to button 130
  adding to interface 71
  compositing 189

image view 71
implementation file 48, 51, 214
importing header files 52
incremental search 96
indentation 93
Info panel 138, 199
informal protocol 97, 131, 136
inheritance 39, 211, 213
    advantage of 213
init 90, 108, 124, 210
initialization 85, 90, 108, 124, 132, 212
    default 90
initializing text 28
initWithCoder: 86, 91, 104, 124
initWithFrame: 132, 181
input source 198
inspector panel 26, 172
    creating 172
    display of 173, 174
    managing 172
instance 39, 210
    generating 44
instance method 49
instance variable 100, 119, 205, 211, 214
    declaring 83, 119, 213
    inheriting 211
    scope 49
    setting 89
Instantiate command 44, 78
interface 24
    creating with Interface Builder 24–35
    testing 35, 36, 75
Interface Builder 24, 107, 210
    and custom NSView 181
    inspector 26
    palettes 28
interface file 214
    Objective-C 50
internationalization 83
introspection 209, 212
invalidate 122, 196
isDocumentEdited: 156, 160
isEqual: 123
isEqualToString: 160
isKindOfClass: 123, 160

J

Java
    packages 9
    virtual machine 9

K

key 63, 158
key alternative 101
key window 27, 169
keyboard event 168, 169
keyWindow 153
keyword 49, 90, 206

L

label 30
library 53
line, creating in Interface Builder 32
link editor 53
linking 53
lipo command 221
loadNibNamed: 144
localization, and nib files 71
localTimeZone 132

M

Mac OS compatibility 9
Mach 8
Macintosh
    look and feel 9
main bundle 126
main menu 138
main nib file 24, 122
main window 27, 146, 169
main() 23, 122
mainMenu 153

mainWindow 146, 153
Makefile 53
makeFirstResponder: 156, 162, 169
makeKeyAndOrderFront: 52
making a connection 46
MallocDebug application 220
matrix 132
  and tabbing 141
menu 27, 72
  customizing 138
  Document 116
Menus palette 72, 116
message 49, 52, 206, 214
  action 41
  nesting 49, 52, 206
method 29, 49, 119, 205
  accessor 49, 85, 86
  class 49
  declaring 50, 85, 120, 213
  delegation 80, 116, 136, 149, 159, 162, 183
  difference from function 208
  extending 214
  inheriting 211
  instance 49
  invoking the superclass 171
  overriding 170, 192, 212, 214
  private 193
  syntax 214
model object 20, 62, 158
Model-View-Controller paradigm 20, 62, 127
modifier key 168
modularity 29, 206
mouse click, simulating 146
mouse event 168
mouseDown: 170, 181
multi-document application 115, 138
  design 139
mutableCopy 112

N

name completion 93
nesting messages 52
New command 139, 143
NeXT
  publications 56
next responder 152, 169
nextKeyView 35, 36, 74
NEXTSTEP 8
nib file 23, 25, 199, 210
  and localization 71
  auxiliary 79, 126, 139
  creating 172
  document 126
  loading 122, 144
  main 24, 122
  sound and images 71
nibTool 221
nil 49, 90, 159
nm command 221
notification 63, 100, 185, 186, 196
  adding an observer 108
  advantages 187
  identifying 170
  posting 171
notification center 100
notification queue 100
NSActionCell 107
NSApp 80, 122, 146, 153, 169
NSApplication 27, 122, 152, 153, 169
NSApplicationMain() 122
NSArchiver 104, 106
NSArray 84, 86, 97, 123
NSBox 172, 187
NSBrowser 153
NSBundle 106, 108, 126, 191
NSButtonCell 132, 161, 190, 191
NSCalendarDate 132
NSCell 107
NSCoder 104
NSCoding protocol 104, 119
NSCompositingOperation 189
NSConnection 198

NSControl 107, 127, 132, 169
NSCopying protocol 119, 123
NSCountedSet 86
NSData 84, 189
NSDate 107, 132, 191, 197
NSDateFormatter 107, 187
NSDictionary 63, 84, 86, 106, 108
 inserting objects 163
NSEvent 168
NSEventType 168
NSFormatter 107
NSHomeDirectory() 147
NSImage 189, 191
NSImageRep 189
NSImageView 71
NSMatrix 127, 132
NSMutableArray 63, 117
NSMutableDictionary 63
NSMutableString 83
NSNotification 100, 109, 185
NSNotificationCenter 100, 109, 185
NSNumber 84, 107
NSObject 38, 84, 89, 131, 212
NSOpenPanel 145
NSProcessInfo 84
NSResponder 127, 152, 168, 169
NSRunAlertPanel() 105, 156
NSRunLoop 122, 198
NSSavePanel 147
NSSet 86
NSString 83, 84, 97
NSTableColumn 68
NSTableDataSource 97
NSTableView 68, 97
NSText 95
NSTextFieldCell 94
NSThread 84
NSTimeInterval
NSTimer 122, 196, 198
NSTimeZone 132
NSUnarchiver 104
NSUserDefaults 150
NSValue 84
NSView 127, 132, 148, 152, 153, 169, 188, 213
 custom 181
 focusing 188
NSWindow 27, 146, 152, 153, 168, 169
number formatter 73
numberofRowsInTableView: 98

O

object 39
 aligning 33
 allocation 84
 analog to 205
 and dynamic binding 209
 and name space 208
 archiving 86, 147, 163
 array 117
 attribute 172
 box 32, 67, 172, 177
 button 31
 class membership 209
 communication 197
 comparison 123
 connecting 40, 45, 46
 controller 20, 63
 copying 29, 123
 creation 210
 deallocation 84, 85, 90, 112, 124
 definition 205
 dictionary 63
 disposal 88, 89
 duplicating 29
 dynamic typing 38
 entity 89
 form 66
 formatter 187
 initialization 85, 90, 108, 124
 initializing text 28
 inspector 172
 interface 29, 206

- introspection 209
- matrix 117, 132, 141
- model 20, 62
- ownership policy 84, 88, 89
- placing 28
- pop-up list 179
- putting in NSDictionary 163
- relation to class 39
- resizing 28
- retaining 88, 187
- retention 89
- scroll view 68
- sizing 28
- text 67, 184
- text field 183, 184
- unarchiving 144
- value 89
- view 117, 127

objectAtIndex: 145, 160
objectForKey:
103
objectForKey: 63, 164
Objective-C 53, 205, 206, 209, 211, 214, 216
- header file 50
- summary 49

object-oriented program 207
- design 20

object-oriented programming 204
objects
- connecting 41, 78
- grouping 67, 173, 174
- making same size 29
- sharing 176

observer 100
oh command 112, 221
Open command 139, 145
Open panel 116, 139, 145
opening a document 145
openPanel 145
OPENSTEP
- application 36, 37

OpenStep 8
- development applications 220

OpenStep frameworks 8
operating system
- core 8

origin point 148
outlet 35, 38, 76, 79, 130
- connecting 45
- defining 40, 42, 43

outline mode 78
overriding a method 170, 190, 192, 212, 214
- invoking superclass 171

ownership, of objects 88

P

palette 28, 126, 222
panel 27
- creating 172
- inspector 173, 174
- off-screen 173, 174

pasteboard 37
pathForImageResource: 191
performClose: 146
periodic event 168
persistence 91
placing objects 28
plug-and-play 37
po command 124
polymorphism 49, 97, 208, 214
pop-up list 27
posting, a notification 100
PostScript 168, 188
- and drawing 189

Preferences panel 138, 200
principal class 122
print: 74
printing 37, 74
procedural programming 204
procedures 204
program development
- command-line tools 221

resources 222
program,object-oriented 207
project 22
adding class to 119
building 54
directory 22
project browser 23
Project Builder 126
checking delimiters 93
indentation 93
launching 22
Project Find panel 56
protocol 216
adopting 86, 119
informal 131
pswrap 181, 189, 221
putCell:atRow:column: 195

R

radio mode 132
receiver 52, 206
reference documentation 56
release 89, 90, 112
removeFromSuperView 187
replaceObjectAtIndex:withObject: 160
representedFilename 145
resizing
view object 28
window 26
resource
for programming 222
resources, application 191
responder chain 152, 169
retain
and content view 177
retain 87, 89, 112, 176
retaining object
implications of 89
reusing object 77
Rhapsody 8
and Power PC 9
architecture 9
supported platforms 9
user experience 9
root class 212
root object 84, 104, 106
rows, of objects 33
runModalForDirectory:file:types: 145

S

Same Size command 29
Sampler application 220
Save command 139, 146
Save panel 116, 139, 147
savePanel 147
saving a document 146, 156
screen, coordinate system 148
scroll view 68
scrolling 188
selectable text 30
selectedCell 137
selectedRow 101
selectText: 52, 94
self 49, 214
sender 52
services 37
setAutosizesCells: 133
setContentView: 177
setDataSource: 97
setDocumentEdited: 147, 149
setFormatter: 107, 187
setFrameOrigin: 188
setFrameSize: 188
setImage: 161
setObject:forKey: 163
setState: 65, 95
setString: 95
setStringValue: 94
setTarget: 169
setting the font 31, 137
setTitle: 144, 161

setTitleWithRepresentedFilename: 144
shallow copy 123
sharedApplication 122
sizing objects 29
sortedArrayUsingSelector: 97
sound 222
Sound Kit 220
source-code files
  generating 48, 82, 119
specialization 213
standard window 27
startTrackingAt:inView: 194
state 65
static typing 49
stopTracking:at:inView:mouseIsUp: 194
string object
  and character strings 83
  empty 90
stringValue 95
strip command 221
subclass 39, 211, 212
  creating 38, 170, 190, 213
  custom view 181
Subclass command 76
subview 67, 153, 177
subview 177
subviews 153
suitcase 23
super 49, 90, 132, 214
superclass 38, 39, 76, 127, 170, 211, 212
  accessor method 192
superview 67, 153, 169
superview 153
switch 65, 95, 179

T

tabbing, between fields 34
table view 63, 68, 97
  configuring 69
tableView:objectValueForTableColumn:row: 98
tableView:setObjectValue:forTableColumn:row: 98
TabulationViews palette 68
tag 131, 132, 135, 177, 179
tag 137
target 40, 46
  setting programmatically 133
target/action paradigm 41, 169, 183
techncial documentation 56
testing an interface 35, 75
text
  aligning 30
  color 30
  editable 30
  selectable 30
text field 183, 184
  attributes 30
  font of 31
  formatting 187
  placing and resizing 28
  tabbing between 34
  validation 187
text object 67, 184
Text submenu 36
textDidChange: 109
textDidEndEditing: 184
thread 198
tile 101
time zone 133
time, *See* date and time
timer 122, 168, 196, 198
  firing 197
  scheduling 196
Title object 30
tools, command-line 221
tracking-rectangle event 168
transparency 189
typing, static and dynamic 211

U

unarchiveObjectWithFile: 108, 144
unarchiving 86, 124, 144

Unicode 83
user defaults 150
userInfo dictionary 171

V

validation, of fields 187
value object 63, 89
view hierarchy 27, 152, 153
view object 27, 127
  custom 117, 181
  printing 74
  removing from superview 187
Views palette 28, 181
virtual machine (Java) 9
volume formats 8

W

window 27, 168
  attributes 26
  behavior 36
  closing 154
  depth 189
  events 169
  key 27
  locating 144
  main 27
  making first responder 156
  positioning 148
  resizing 26
  setting title 144
  status 146
window 144
Window Server 27, 168
windows 153
Windows palette 173, 174
windowShouldClose: 154

Y

Yap application 220
Yellow Box 8, 9