VarietiesofButtons;¬Varieties of Buttons

If in Interface Builder you select the ªEnglish widely spokenº 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 buttonsÐsuch as switches and radio buttonsÐ 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.

_IB_ButtonAttributes.eps ¬

938726_TableRule.eps ¬

MoreAboutForms;¬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 many corporate client-server applications do.

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

n addition to the obvious controls in the Forms inspector, there's the ªCell tags = positionsº 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 idenitfy 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.

55588_TableRule.eps ¬

MoreAboutTableViews;¬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 Object Framework (EOF) 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).

_IB_TVonPalette.eps ¬

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 NSTableDataSources 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.

814601_TableRule.eps ¬

TheCollectionClasses;¬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.

CollectionClasses.eps ¬

- 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 ªredº 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 (see ``Abstract Classes and Class Clusters, below'').

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.

509242_TableRule.eps ¬

CheckingConnectionsinOutlineMode;¬Checking 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.

_IB_OutlineView.eps ¬

618096_TableRule.eps ¬

File'sOwner;¬File's Owner

Every nib file has one owner, represented by the File Owner's 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 can specify a file's owner in Interface Builder or programmatically, with NSBundle's **loadNibNamed:owner:**. The File's Owner icon for the main nib file 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 fileÐ *auxillary* nib filesÐcontain objects and resources that an application may load only when it needs them (for example, an Info panel). You must specify the owner of auxillary 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.

558885_TableRule.eps ¬

JustAddaSmock:CompiledandDynamicPalettes;¬Just Add a Smock: Compiled and Dynamic Palettes

A palette is a display on the 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 ªstatic palettesº). 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 codeÐthey'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 inspectors and editors, which dynamic-palette objects cannot have.

You usually create a static palette as a way to distribute your objectsÐand the logic informing these objects' behaviorÐto potential users. Many developers of commercial OpenStep objects make use of static palettes as a distribution media. 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.

See *OpenStep Development: Tools and Techniques* for the procedure required to develop static palettes.

Using Dynamic Palettes

Dynamic palettes are a big 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 rightarrow.tiff ¬Palettes 214094_rightarrow.tiff ¬New to create a blank palette.

- Select objects singly or in groups on the interface or in the nib file window (both icon and outline modes)
- Alternate-drag these objects and drop them on the blank palette.

_IB_DynamicPalette.eps ¬

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 ªstandardº text field of a certain length, font, and background color stored on a dynamic palete.
- 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.

698812_TableRule.eps ¬

NSString:AStringforAllCountries;¬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 ***.

ªBut why?º you might be saying. ªWhy not stick with the tried and true?º By representing strings as objects, you confer on them all the advantages that belong to objects, such as persistency and distributability. 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, Japanese, 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 kit classes now use string objects almost exclusively. A number of NSString methods enable you to convert string objects to **char *** strings and back again.

29995_TableRule.eps ¬

TheFoundationFramework:Capabilities,Concepts,andParadigms;¬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. 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'' ;TravelAdvisorConcepts.rtfd;ObjectOwnership,Retention,andDisposal;¬ for more information.

Data Storage and Access

The Foundation Framework provides object-oriented storage for

- Srrays 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,'' above ;TravelAdvisorConcepts.rtfd;TheCollectionClasses;¬).

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 provide methods for searching, combining, and comparing strings. NSCharacterSet represents various groupings of characters which are used by NSString. You use an NSScanner object to scan numbers and words from an NSString object. For more information, see ``NSString: A String for All Countries'' on page 21.

You use NSBundle objects to load code and localized resources dynamically. The NSUserDefaults class enables you to store and access default values based on locale.

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 and NSProcessInfo let you create multi-threaded applications and query 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.

87445_TableRule.eps ¬

ObjectOwnership,Retention,andDisposal;¬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 ªconsumer' objects in an

application, which object is responsible for disposing of it? And when? If the object is not deallocated, 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 whenÐand only whenÐthey 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 **retain** increments an object's reference count. The flip side of this rule is: If you don't own an object, you need not worry about releasing it.

OK, 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 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, and 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 has an *autorelease pool*. 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. When your code finishes executing and control returns to the application object (typically at the end of the event 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.

How Autorelease Pools Work: An Example

A. **myObj** creates an object (**anObj**):

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

Autorelease1.eps ¬

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

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

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

C. **yourObj** retains the object:

```
[anObj retain];
```

`Autorelease2.eps ¬`

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 decrements the reference counts of all objects it references.

   Autorelease3.eps ¬

E. yourObj sends autorelease to the object. At the end of the event cycle, the reference count is set to zero and the object is deallocated.

   Autorelease4.eps ¬

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

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 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** to a 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 received from it and retained is usually invalid. If you had retained an instance variable of the owning object, and that instance variable is reassigned, your reference would also become invalid.

Autorelease_sideEffect.eps ¬
*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.*

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 ªsnapshotº of the object; use **retain** when you intend to share it. 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**Ðnot **retain**Ðto 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.

919289_TableRule.eps ¬

TurboCodingWithProjectBuilder;¬Turbo Coding With Project Builder

When you write code with Project Builder you have a set of ªworkbenchº 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.

Brace and Bracket Checking

Double-click a brace (left or right, it doesn't matter) to locate the matching brace; the code in-

between the braces is highlighted. In an identical fashion, double-click a square bracket in a message expression to locate the matching bracket.

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 (or Tab, if that key is bound in Preferences). 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 issue 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). You can also perform an incremental search by pressing Control-s; this command displays a small search panel and takes you to the next occurrence of whatever you type.

656724_TableRule.eps ¬

FindingInformationWithinYourProject;¬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.

_IB_FindingInformationA.eps ¬

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.

- **Man pages**: Choose Edit 240442_rightarrow.tiff ¬Find 676333_rightarrow.tiff ¬Man Page to bring up the ªShow man pageº panel. Enter the name of a tool in the panel to get the man page on that tool.

- **Librarian via Services**: Select a symbol or any word (for example, ªservicesº) in Project Builder, then choose Services 761957_rightarrow.tiff ¬Librarian 853130_rightarrow.tiff ¬Search to have Digital Librarian find related documentation.

- **Help**: Project Builder and Interface Builder also feature context-sensitive help and task-

related help. See ªWhere to Go For Helpº for details ;../2_CurrencyConverter/CurrencyConverterConcepts.rtfd;linkMarkername WheretoGoForHelp;¬.

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

467022_TableRule.eps ¬

GettinginontheAction:DelegationandNotification;¬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 offer two mechanisms through which your objects can participate 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 a 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.

Delegation.eps ¬

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.

Notification.eps ¬

Here's the way the notification process works:

- Objects that are interested in an event that happens elsewhere in the application Ð say the addition of a record to a database Ð register themselves with a *notification center* (an instance of NSNotificationCenter) as observers. 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.

491602_TableRule.eps ¬

AbstractClassesandClassClusters;¬Abstract Classes and Class Clusters

Many of the classes in the Foundation Framework fall into functional constellations of public and private classes called *class clusters*. Class clusters simplify the programming interface and

permit more efficient storage of data.

An abstract class (such as NSArray) defines the public interface for objects vended from class clusters. Abstract classes declare methods common to private, concrete subclasses, but do not declare any instance variables to hold dataÐthat's done by the private classes. When you send an object-creation message to an abstract class, it instantiates and returns an instance of the appropriate private subclass. What's appropriate depends on the creation method, which indicates the type of storage required. The class membership of the returned object is hidden, but its interface, as declared by the abstract superclass, is public.

Many OpenStep class clusters have two or more abstract classes. Usually one class provides the interface for obtaining immutable objects (for example, NSArray) and another class, which inherits from the mutable class, vends mutable versions of the same type of object (NSMutableArray).

870391_TableRule.eps ¬

BehindªClickHereº:Controls,Cells,andFormatters;¬Behind ªClick Hereº: 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 objectsÐcontrols inherit from NSControl while cells inherit from NSCellÐthey 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," above. 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 ªembeddedº 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.

_ControlCellExample.eps ¬

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 ªweightº of cells and controls, provides a great boost to application performance.

ControlCellMechanics.eps ¬

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 buttonÐfrom Interface Builder or programmatically createdÐis 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.

FormatterExample.eps ¬

The Foundation framework provides the NSDateFormatter and NSNumberFormatter classes. You can make a custom subclass of NSFormatter to derive your own formatters.

9829_TableRule.eps ¬

FlatteningtheObjectNetwork:CodingandArchiving;¬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 its 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.

415287_TableRule.eps ¬

UsingtheGraphicalDebugger;¬ Using the Graphical Debugger

Project Builder's graphical debugger provides an easy-to-use, intuitive user interface to **gdb**, the GNU debugger.

_PB_Debugging.eps ¬

_PB_DebugOptions.eps ¬

980121_TableRule.eps ¬

TipsforEliminatingDocumentationBugs;¬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. Send **enableFreedObjectCheck:** to NSAutoreleasePool with an argument of YES.
2. Set a breakpoint on _**NSAutoreleaseFreedObject**.
3. Run the program under the debugger.
4. When the program hits the breakpoint, do a backtrace and check the stack to find the method releasing the object.

Other tools help you track down problems related to release and autorelease:

- The **oh** command records allocation and deallocation events related to a specific process. It produces a report showing the stack frame for an object each time the object is allocated, copied, retained or released.
- The **AnalyzeAllocation** tool compiles statistics on memory allocation during the time a program executes.

See the man pages on these tools for more information.

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 request an object stored in a collection object, it's returned to you autoreleased.
- 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.