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

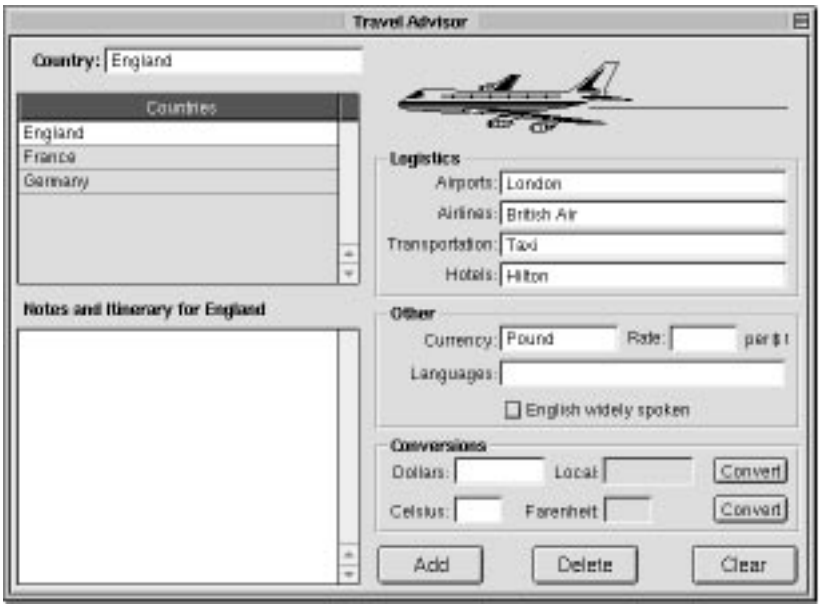
Delegation and notification

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 interesting, 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

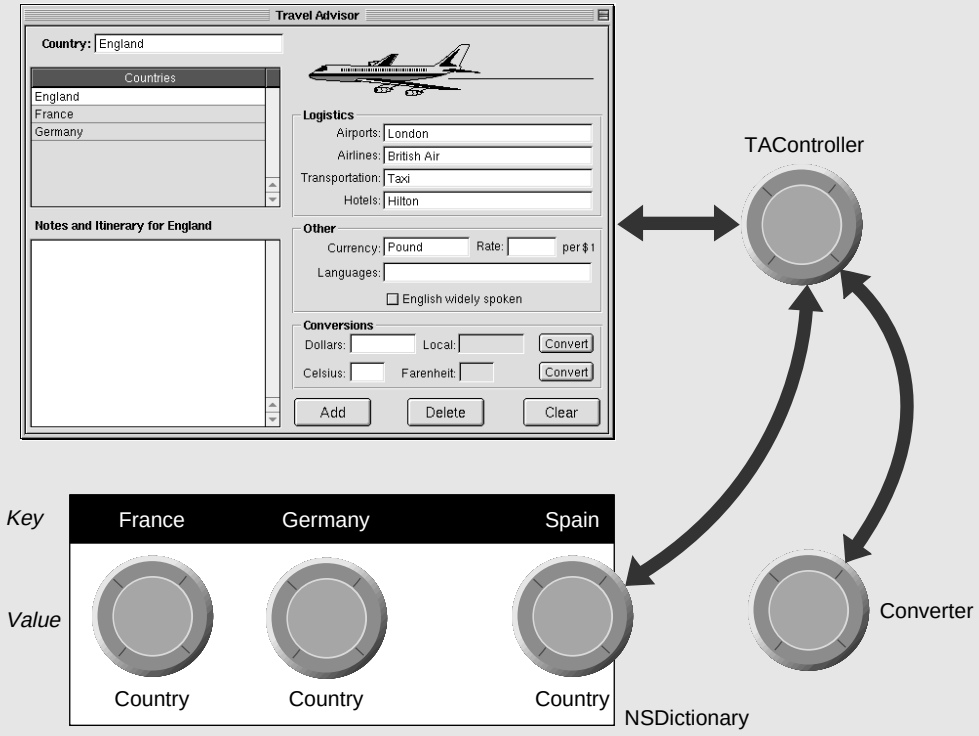
Travel Advisor is much like Currency Converter in its basic design. Like Currency Converter, it's based on the Model-View-Controller paradigm. A controller object (TAController) manages a user interface comprised of Application Kit objects. Also as before, the controller sends a message to the Converter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

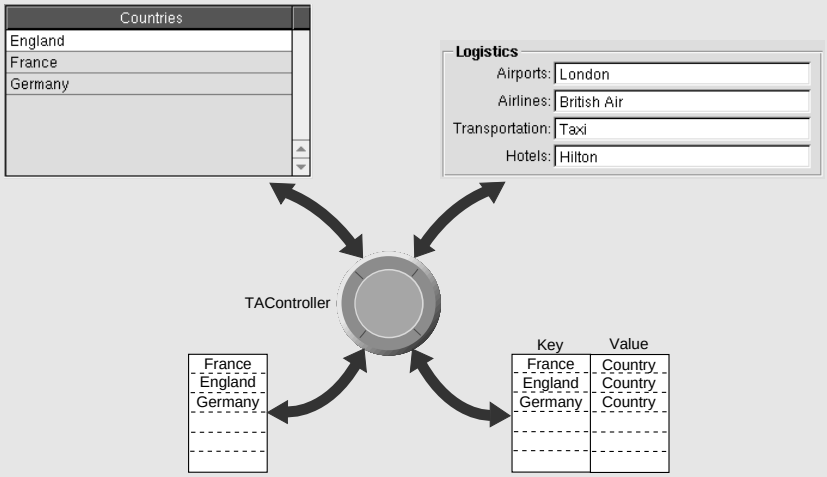
Delegation and Notification. An essential aspect of design not evident from the diagram are the roles *delegation* and *notification* play. The TAController object is the delegate of the application object and thereby receives messages that enable it to manage the application, which includes tracking 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.

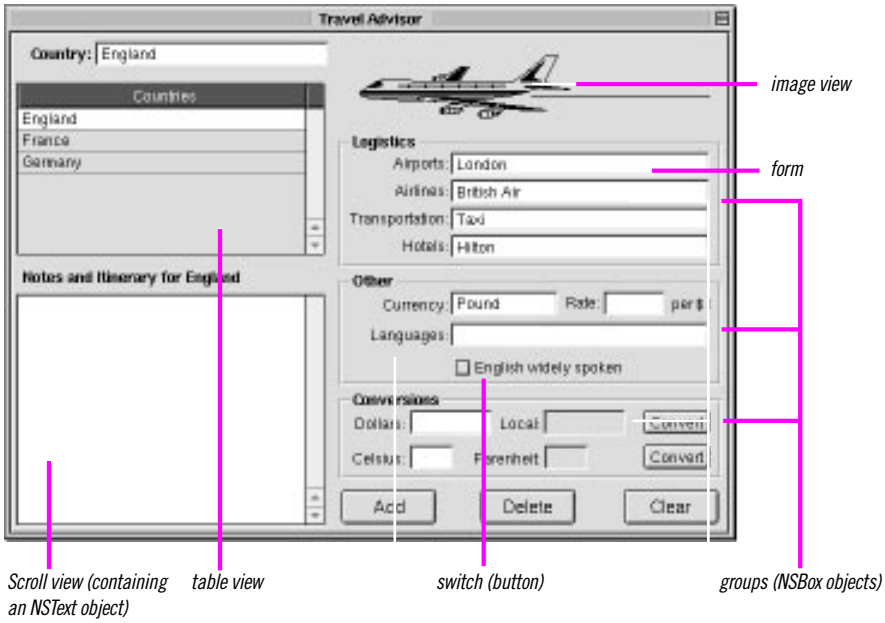


Creating the Travel Advisor Interface

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.



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 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:

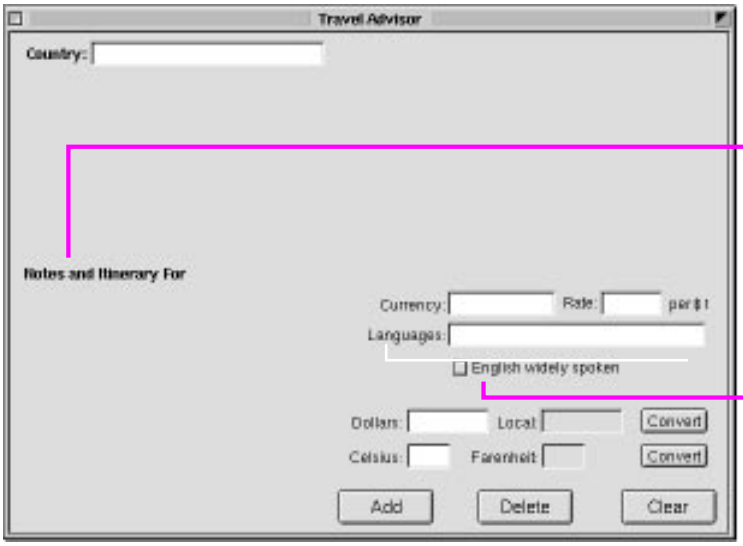
Resize the window, using the example at right as a guide.

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

Turn off the resize bar.

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

Position, resize, and initialize the objects as shown.



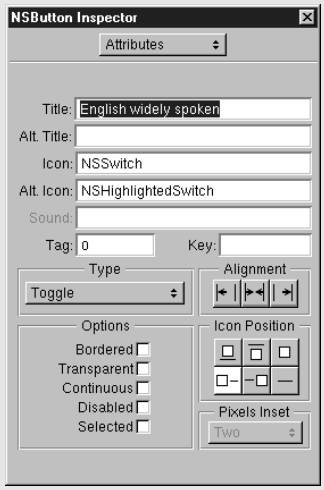
You might think the "English widely spoken" object is a new kind of object. It's actually a button, a special style of button called a switch.

Set up the switch.

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.

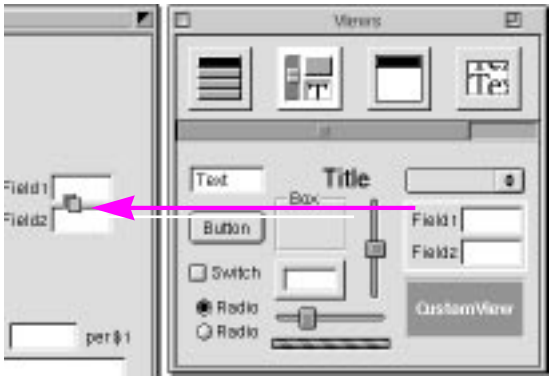


New Objects: Forms, Groups, and Scroll Views

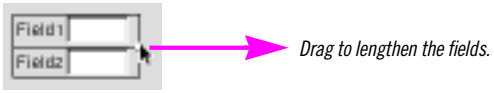
Construct the "Logistics" section of the interface using a form object.

1 Place a form on the interface and prepare it.

Drag the form object from the Views palette.

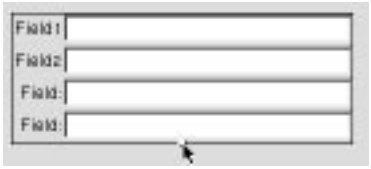


Increase the size of the form's fields by dragging the middle resize handle sideways.



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

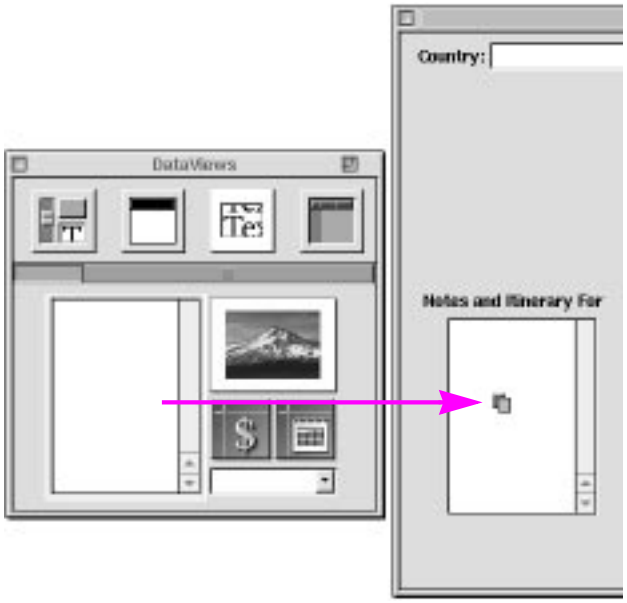
In addition to the obvious controls in the Forms inspector, there's the "Cell tags = positions" attribute. Switching this on assigns tags to each NSFormCell that correspond to the cells' indices. (A tag is a number assigned to an object that is used to identify and access that object. You'll use tags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

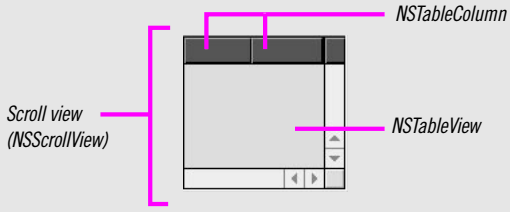


You don't need to change any of the default attributes of the scroll view (but you might want to look at the attributes that you can set, if you're curious).

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.

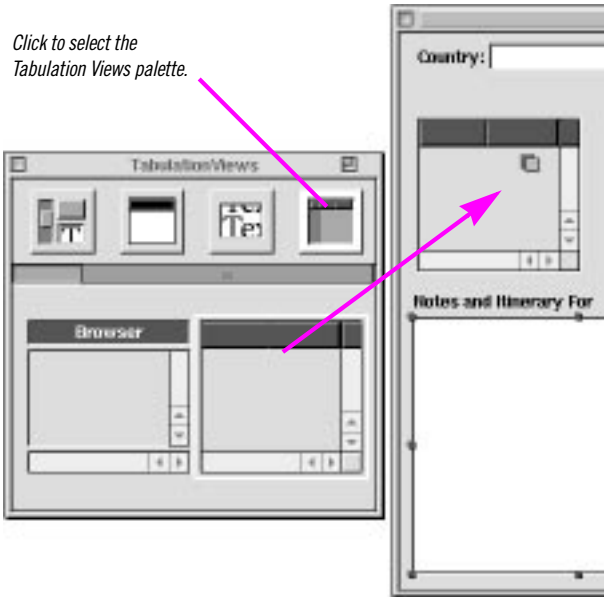
More New Objects: Table Views, Image Views, and Menus

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

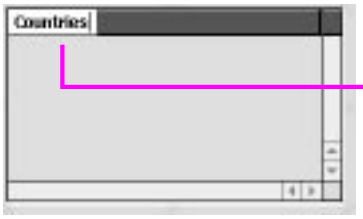
1 Place and configure the table view.

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.

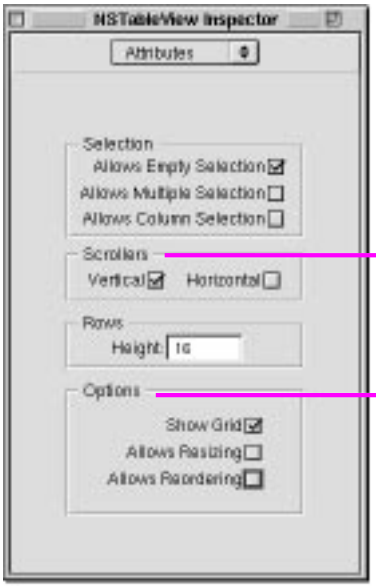
You can also delete the unneeded column by selecting it and pressing the Delete key.

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.

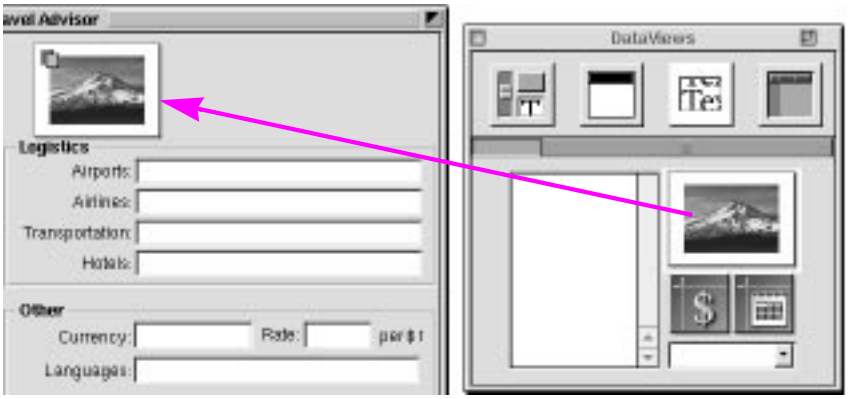
Select the DataViews palette (see example).

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.

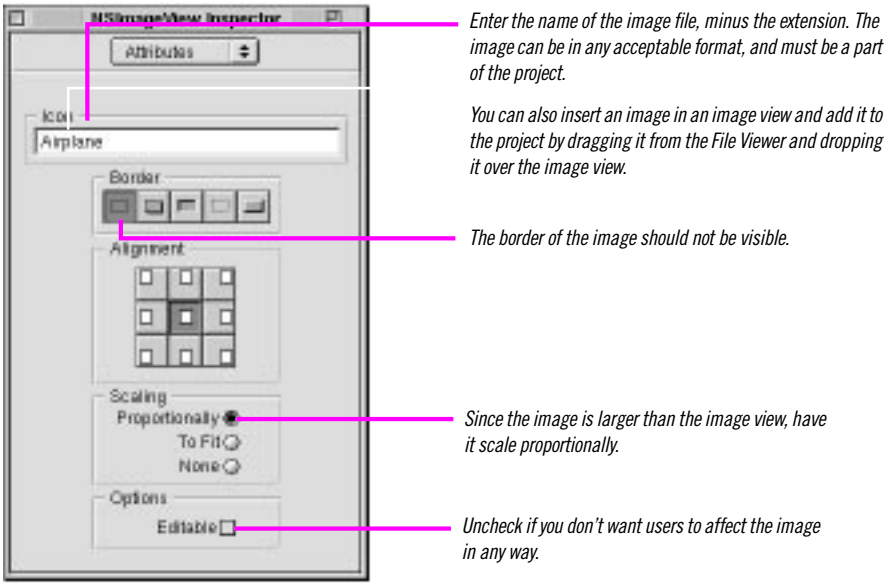
When you drop an image over a button or image view, Interface Builder adds it to the both the nib file and the project (upon your approval). You can add the image only to the nib file by dropping the image over the nib file window. Resources in a nib file are accessible only when the nib file has been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



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.

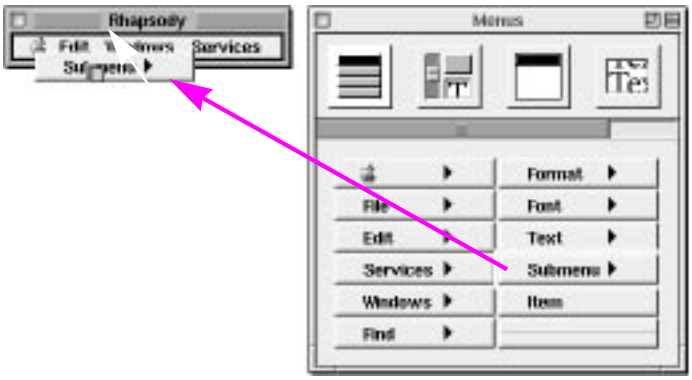
Change the command names to those shown at right.

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.

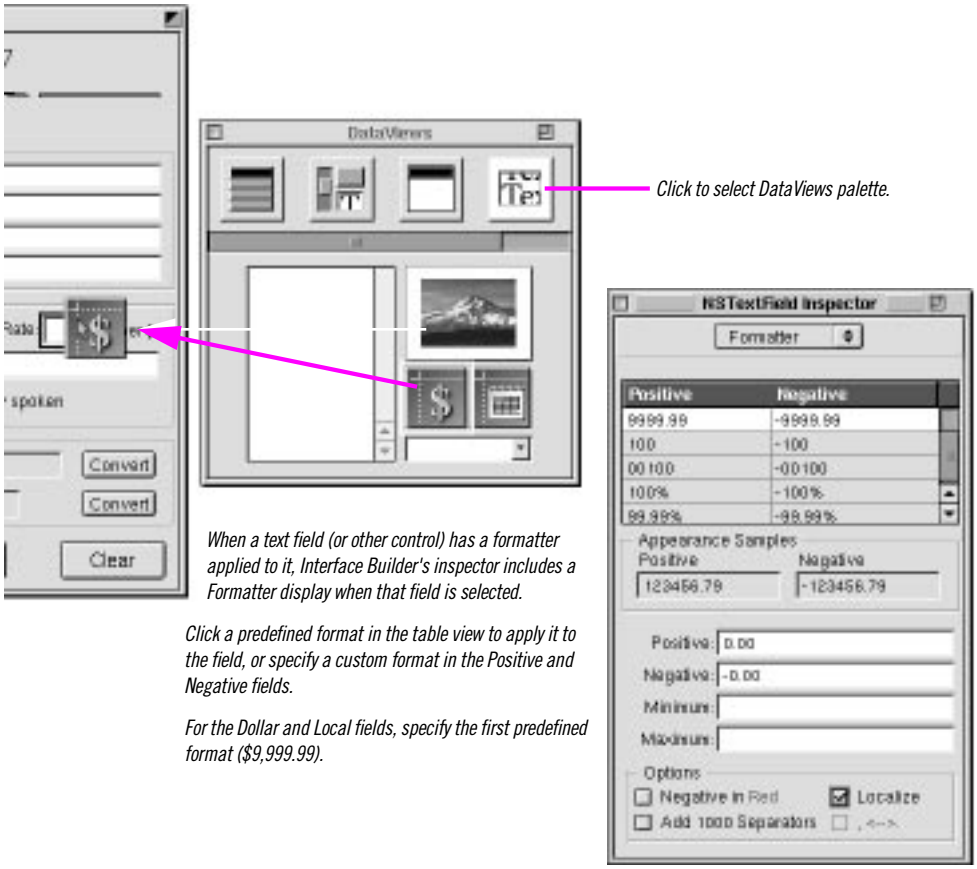
Select the DataViews palette in the Palette window.

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.



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.)

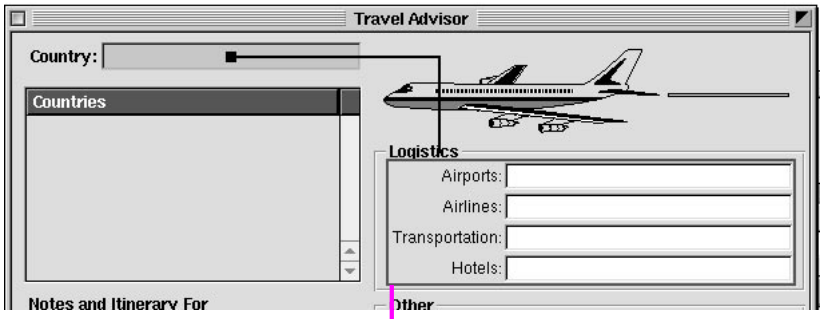
1 Connect Application Kit outlets for inter-field tabbing and printing.

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.



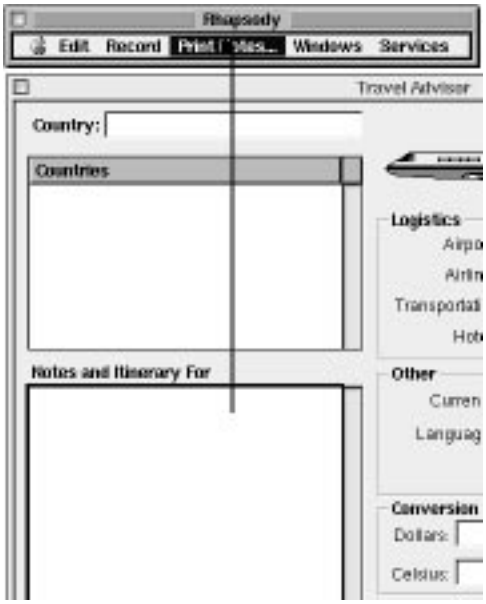
When a line borders the form inside the box, the form is selected. Release the mouse button and set the **nextKeyView** *outlet connection in the Connections inspector.*

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.

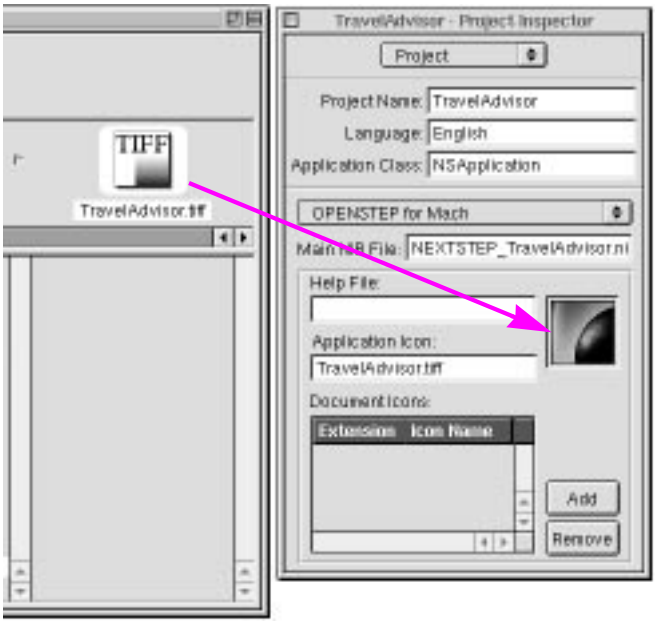
Go to the Project Attributes display of the inspector.

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.

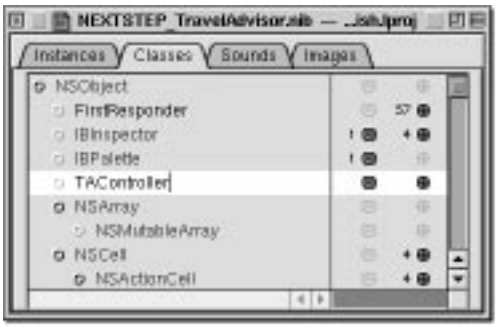
In Interface Builder, bring up the Classes display of the nib file window.

Select NSObject as the superclass.

Choose Subclass from the Classes menu.

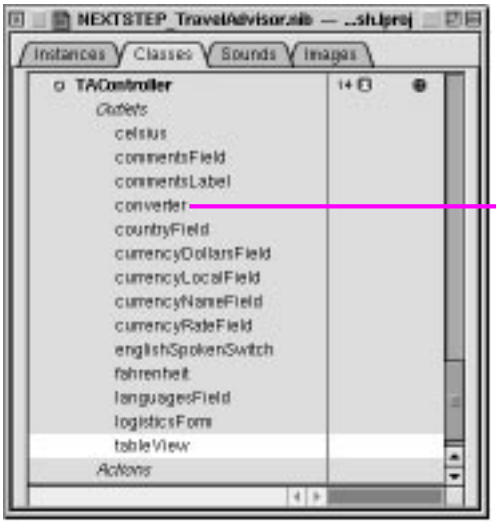
Type “Country” in place of “MyNSObject.”

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.

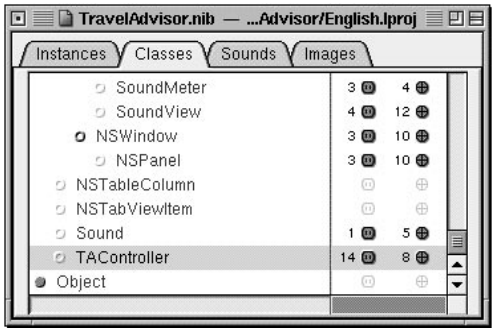
Choose Classes ► Read File.

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

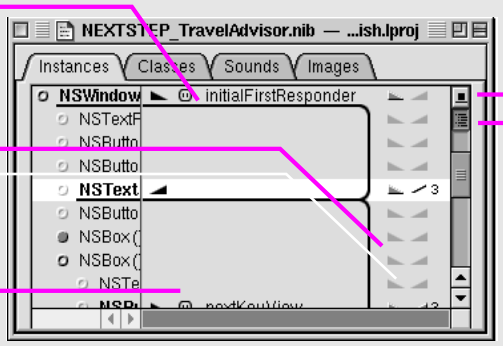
The nib file window of Interface Builder gives you two modes in which to view the objects in a nib file and to make connections between those objects. So far you've been working in the *icon mode* of the Instances display, which pictorially represents objects such as windows and custom objects.

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.

A connection is identified by name and icon for type (electrical outlet for outlet, cross-hairs for action).

To see connections from the object, click a right-pointing triangle; click a left-pointing triangle for connections to the object.

Move the vertical line left or right to see details (this is a vertical split view).



Click here for icon mode.

Click here for outline mode.

Connect objects in outline mode just as you do in icon mode: Control-drag a connection line between objects.

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.

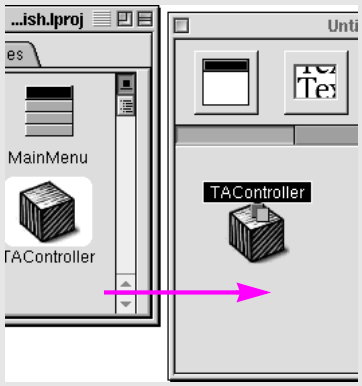
Every application has a global NSApplication object (called NSApp) that coordinates events specific to the application. Among many other messages, NSApp sends a message to its delegate notifying it that the application is about to terminate. Later, you will implement TAController so that, when it receives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



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

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.

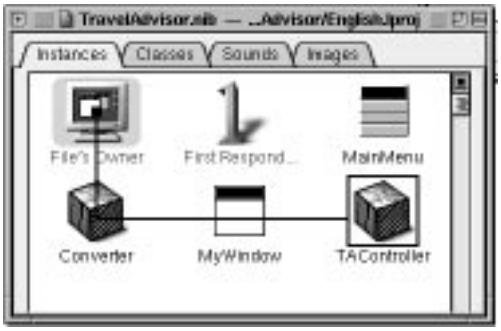
You can use dynamic palettes to:

- 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.

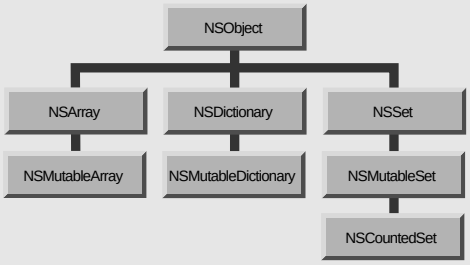
The Collection Classes

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:

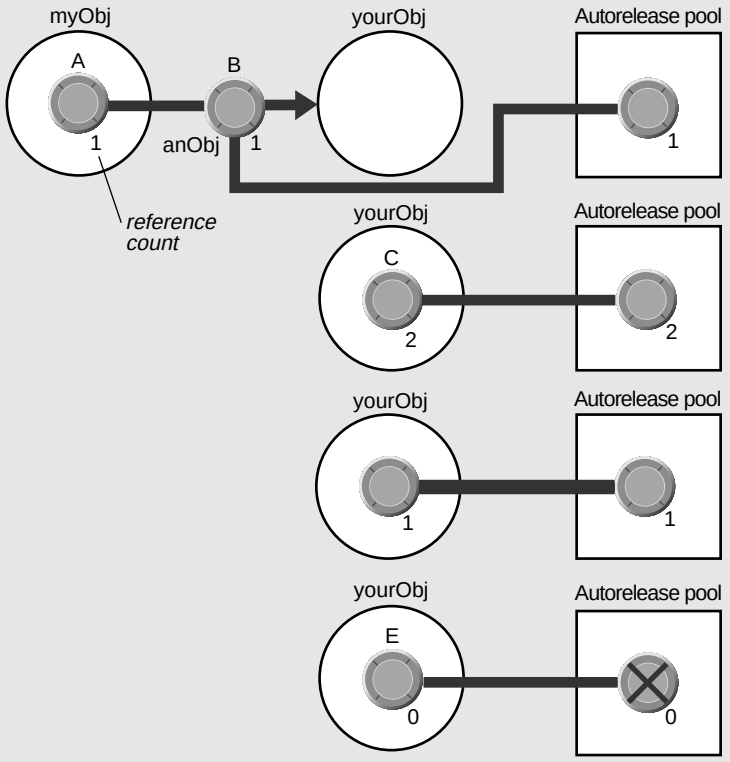
```
[anObj retain];
```

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

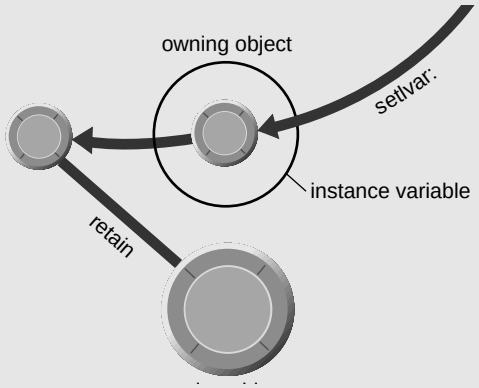
When an object is used solely within the scope of the method that creates it, you can deallocate it immediately by sending it **release**. Otherwise, send **autorelease** to all created objects that you no longer need but will return or pass to other objects.

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 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.



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 "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]];                          A

    [[logisticsForm cellAtIndex:LGairports] setStringValue:
        [aRec airports]];                                               B
    [[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];                                     C
}
```

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

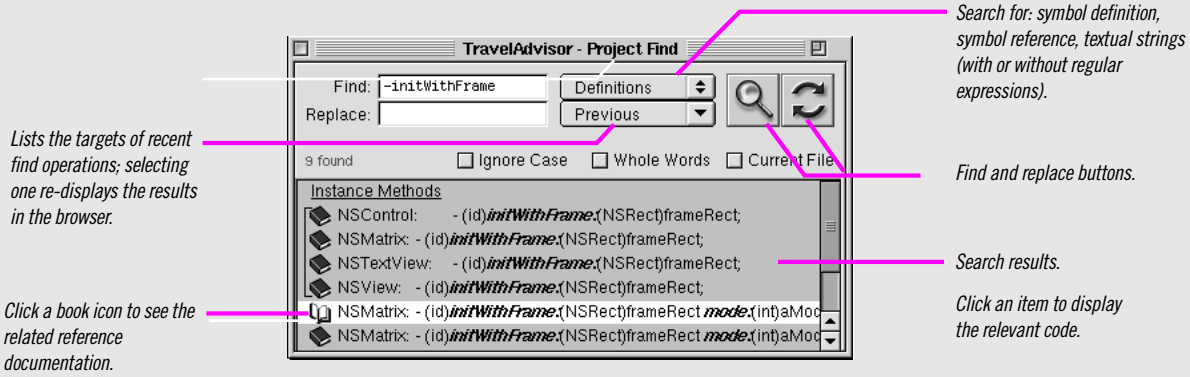
Exercise: Implement the **extractFields:** method. In this method set the values of the passed-in Country record's instance variables with the contents of the associated fields.

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.



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]                    Ⓐ
        sortedArrayUsingSelector:@selector(compare:)];
    countryKeys = [[NSMutableArray alloc] initWithArray:tmpArray];

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

Ⓐ 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.

Ⓑ 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.

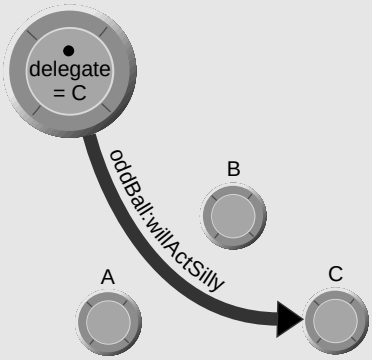
Getting in on the Action: Delegation and Notification

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:

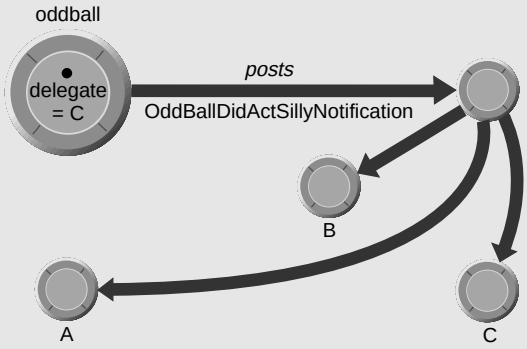
- Some messages are purely informational, occurring after an event has happened. They allow a delegate to coordinate its actions with the other object.
- Some messages are sent before an action will occur, allowing the delegate to veto or permit the action.
- Other delegation messages assign a specific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

Ⓐ 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.

Ⓑ 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.

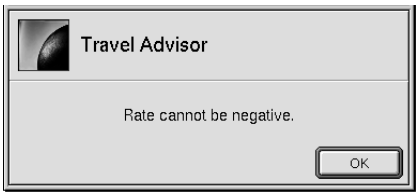
Implement the **control:isValidObject:** method to validate the value of the field.

```
[currencyRateField setDelegate:self];
```

```
- (BOOL)control:(NSControl *)control isValidObject:(id)obj
{
    if (control == currencyRateField) {                    A
        if ([obj floatValue] < 0.0) {
            NSRunAlertPanel(@"Travel Advisor",             B
                @"Rate cannot be negative.", nil, nil, nil);
            return NO;
        }
    }
    return YES;
}
```

A 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.

B 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.

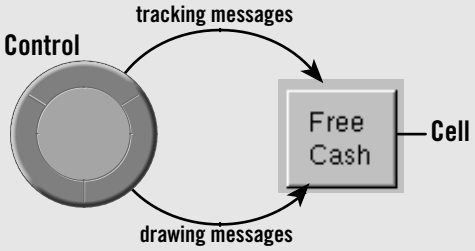
B If the **countryDict** dictionary holds Country objects, TAController archives it with the NSArchiver class method **archiveRootObject:toFile:**. Since the dictionary is designated as the root object for archiving, all objects that the dictionary references (that is, the Country objects it contains) will be archived too.

Behind 'Click Here': Controls, Cells, and Formatters

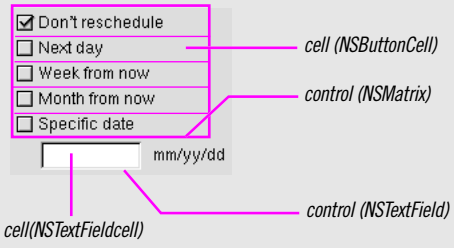
Controls and cells lie behind the appearance and behavior of most user-interface objects in OpenStep, including buttons, text fields, sliders, and browsers. Although they are quite different types of objects—controls inherit from NSControl while cells inherit from NSCell—they interact closely.

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.

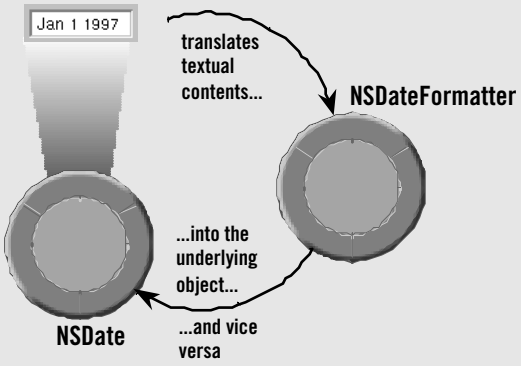


A control does not have to have a cell associated with it, but most user-interface objects available on Interface Builder's standard palettes are cell-control combinations. Even a simple button—from Interface Builder or programmatically created—is a control (an NSButton instance) associated with an NSButtonCell. 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:

$F° = 9/5C° + 32$

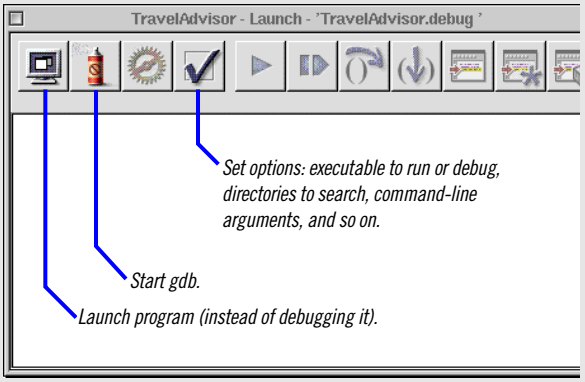
Using the Graphical Debugger

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**.

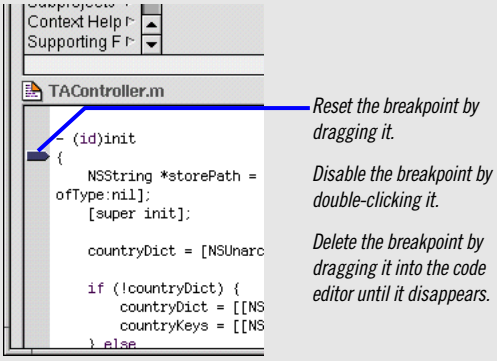


2. Set a breakpoint.

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.

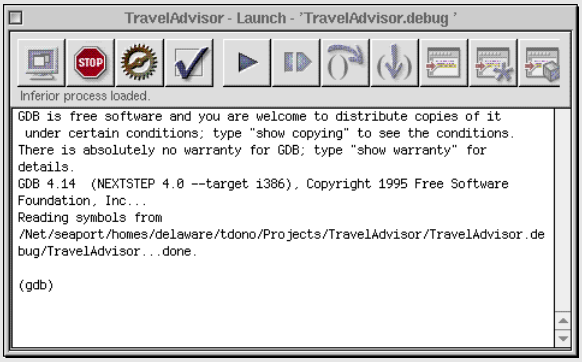


3. Start debugging the application.

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.

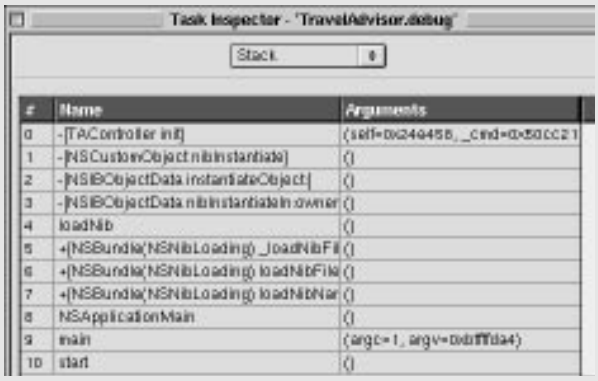
You can type **gdb** commands at this prompt. There are many **gdb** commands not represented in the user interface. For on-line information on these commands, enter "help" at the prompt. You can also find more about commands in the on-line **gdb** reference.



4. Inspect the stack trace.

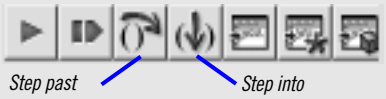
When a program running under the debugger hits a breakpoint, the graphical debugger displays a trace of the call stack. You can see the sequence of calls leading up to the breakpoint as well as the values of arguments of methods or functions implemented by your project.

The Stack display is part of the Task Inspector, which you open by clicking the following button on the Launch panel:

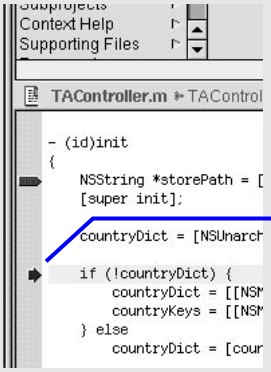


5. Step through code.

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.



The arrow shows the program counter as you step through code.

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.