

6

Subclassing

A roadmap to making or adding custom classes

Naming a new class

Specifying outlets and actions

Creating an instance of your class

Connecting your class's outlets

Connecting your class's actions

Generating source code files

Implementing a subclass of NSObject

Making your class a delegate

Implementing a subclass of NSView

Adding existing classes to your nib file

Updating a class definition

I inherited it brick, and left it marble.

Emperor Augustus

They rightly do inherit heaven's graces,
And husband nature's riches from expense.

Shakespeare, *Sonnets*

Observe how system into system runs,
What other planets circle other suns.

Alexander Pope, *An Essay on Man*

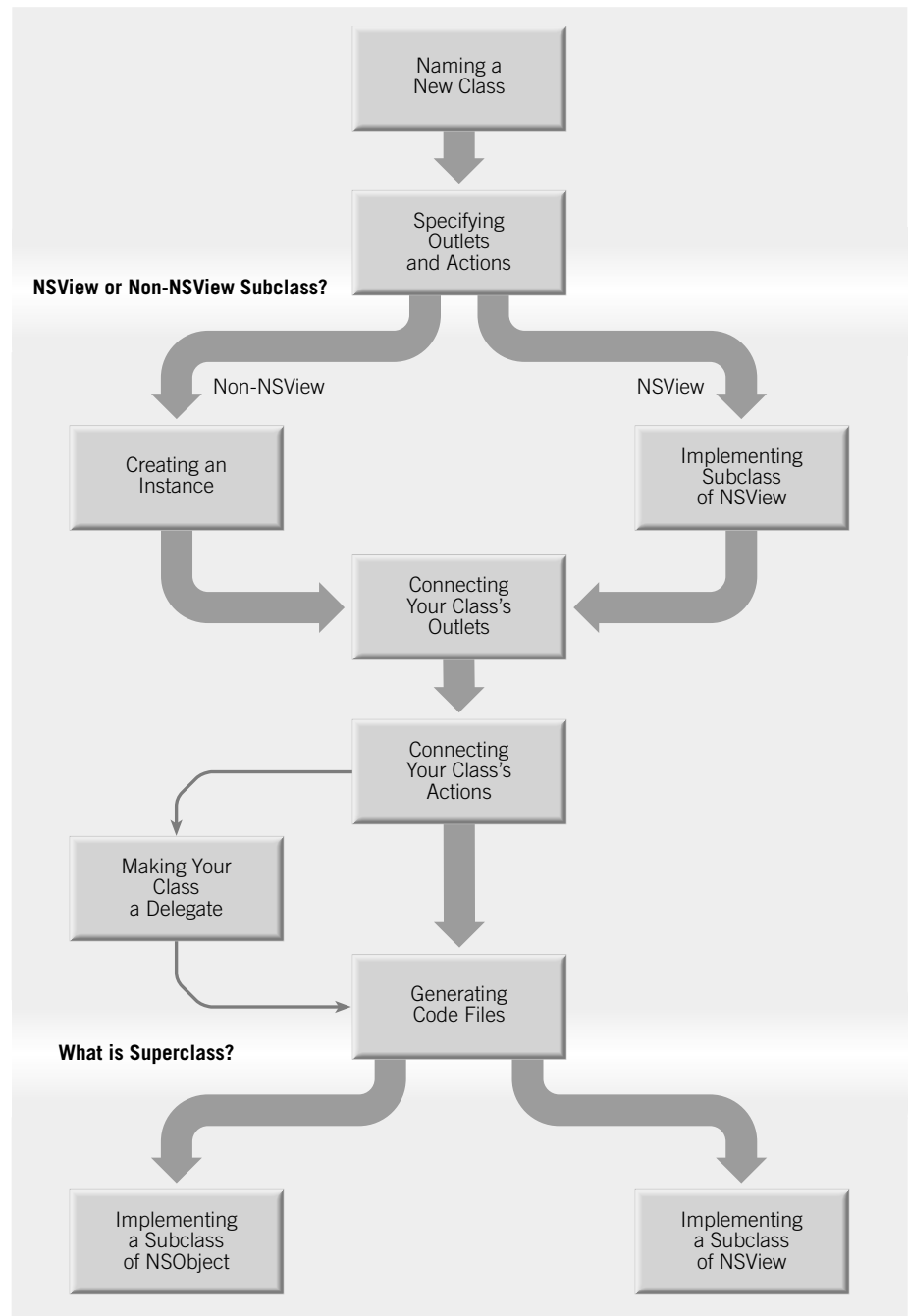
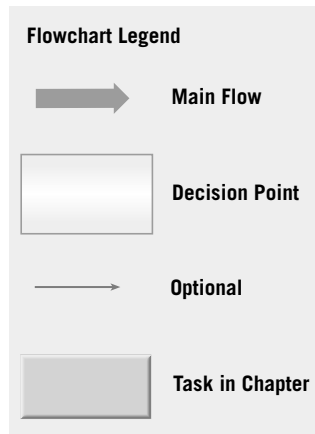
A roadmap to making or adding custom classes

- ▶ **Determine which flowchart applies to your situation.**
- ▶ **Follow the tasks in this chapter in the order specified by that flowchart.**

This chapter differs from the other chapters in this book because its subject is different. Creating a class (or adding an existing class) is not a set of discrete, modular tasks, but a process consisting of many interdependent tasks. The order of tasks in this chapter is therefore significant; with some exceptions, you need only follow the tasks sequentially, from first task to last task, and you'll end up with a useful class.

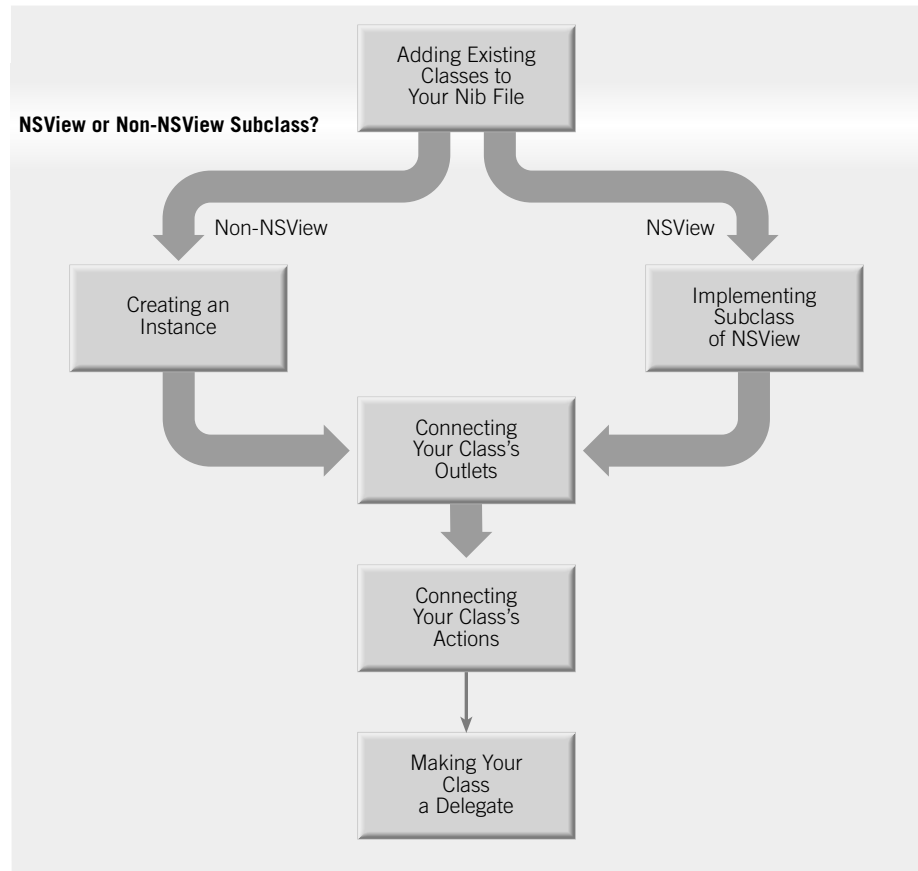
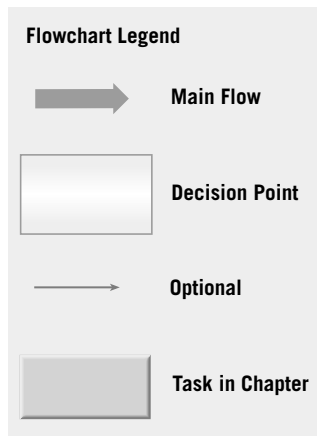
But those exceptions are significant, and so flowcharts are provided to point the way. The flowchart on the facing page guides you through the tasks required to define and implement a subclass of the NSObject class or of the NSView class. An additional flowchart identifies the tasks you must complete to integrate an existing class into an application.

This chapter also differs from other chapters in this book because it covers a topic that involves both Interface Builder and Project Builder. To start creating a class, you use Interface Builder. It helps you locate the class in the hierarchy, name it, connect an instance of it with other objects in an application, and generate template source files. When Interface Builder's role is done, you switch to Project Builder and provide the most important contribution, the source code that gives your class its distinctive behavior. (As an alternative, you can start creating a class in Project Builder then add it to Interface Builder and make the connections to other objects later.)



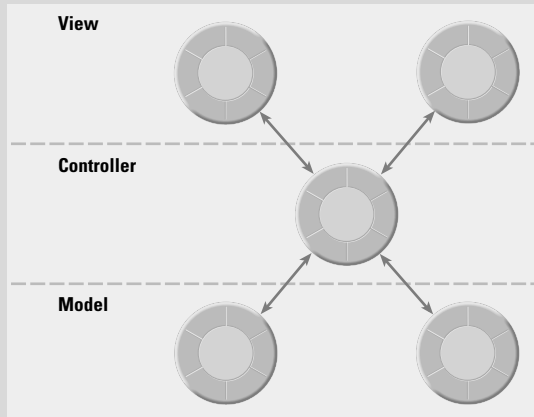
If you branch to “Implementing a subclass of NSView” after specifying outlets and actions, complete only the step “Making an Instance of an NSView Subclass” in that task for now, and go on to the next task. Do the rest of “Implementing a subclass of NSView” after you’ve generated code files.

After generating code files, you must switch over to Project Builder and open the header and implementation files.



The Model-View-Controller Paradigm

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



Model Objects

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

View Objects

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

Controller Object

Acting as a mediator between Model objects and View objects in an application is a Controller object. There is usually one per application or window. A Controller object communicates data back and forth between the Model objects and the View objects. It also performs all the application-specific chores, such as loading nib files and acting as window and application delegate. Since what a Controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application's code.

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

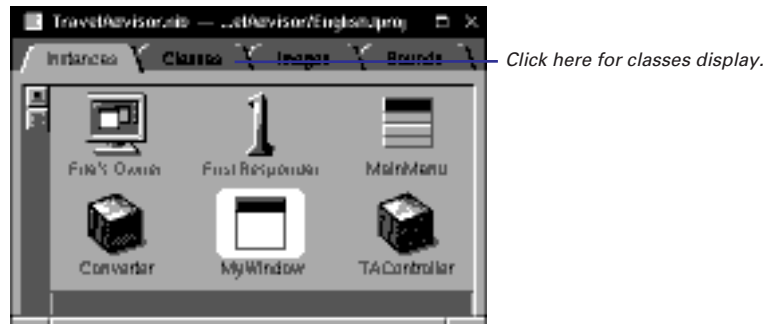
Hybrid Models

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

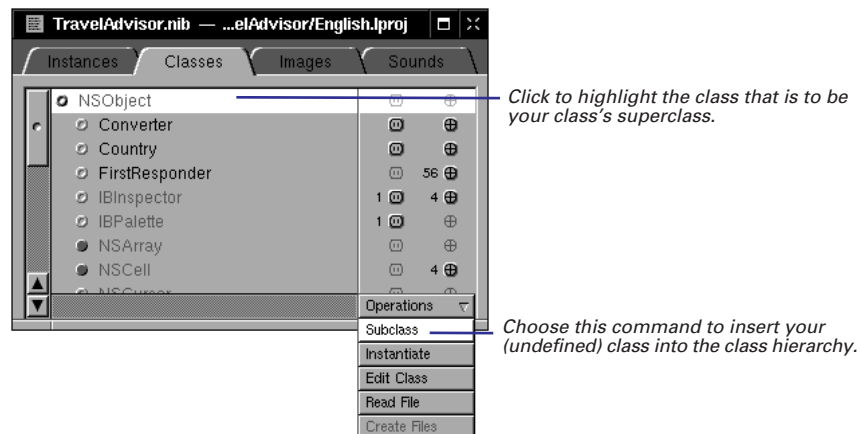
Naming a new class

- 1 In Interface Builder, display the **Classes** display of the nib file window.
- 2 Select the class you want your class to inherit from.
- 3 Choose **Subclass** from the **Operations** menu.
- 4 Type the name of your class over the highlighted “default” name.

When you create an application, you must create at least one subclass to do anything meaningful. The OpenStep frameworks do a lot of the work for you, but you must always supply, in one or more subclasses, the distinctive logical and computational flow of your application.



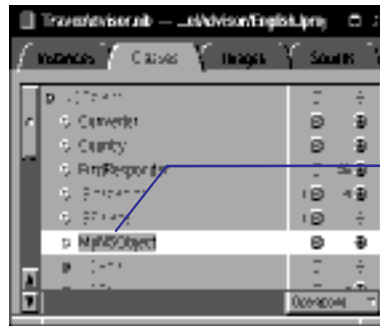
When you subclass, the first thing you must do is select your class's superclass. Ideally, the superclass of your class should behave much the way you want your class to behave. Your class merely adds the behavior you want to what the superclass offers, or modifies the superclass's behavior in some way. Often the behavior you want is so bound to resolving a particular problem that the proper choice of superclass is NSObject because it provides the most generic behavior.



See “A Short Practical Guide to Subclassing” in this chapter for more on the relation between superclasses and subclasses.

Tip: Pressing Return when a class is selected is equivalent to choosing the Subclass command.

The new class is listed under its superclass with a default name: the superclass name prefixed with “My” (such as “MyNSObject”). Replace this default name with the new name.



Type the name of your Class over the default name. Press Return.

Later, if you want to rename the class, first re-select the class name by double-clicking it. Then type the new name, replacing the selected text.

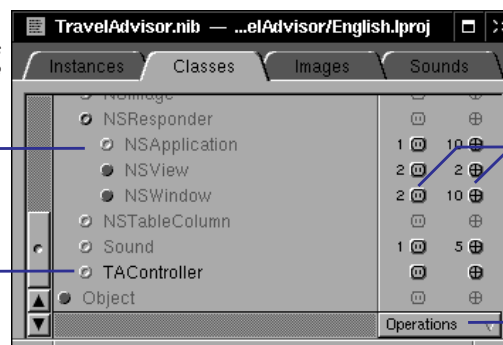
A Perspective on the Class Hierarchy

The Classes display of the nib file window shows the classes that the current nib file is aware of. The display lets you browse through both OpenStep classes and custom classes. The Classes display also depicts (by indentation) class-inheritance relationships and reveals the names of each class’s outlets and actions.

Keyboard Navigation Move up and down in the list of classes pressing the up arrow and the down arrow. When a class is highlighted, show its subclasses by pressing the right arrow; collapse an indented list by selecting the superclass and pressing the left arrow. If the nib file window is active, incremental search is active: just type the first few letters of a class until its name is highlighted.

The Classes display shows hierarchy by indentation (for example, `NSApplication` inherits from `NSResponder`). If the circle button is filled, the class has subclasses that are not shown. Click the button to display the subclasses.

If the class name is black, it is a custom class. If the class name is gray, the class is a NeXT-provided class.



Outlet (electrical-outlet icon) and target/action (cross-hairs icon) buttons. Click to display class outlets and actions.

A pull-down list of operations related to creating a class.

Specifying outlets and actions

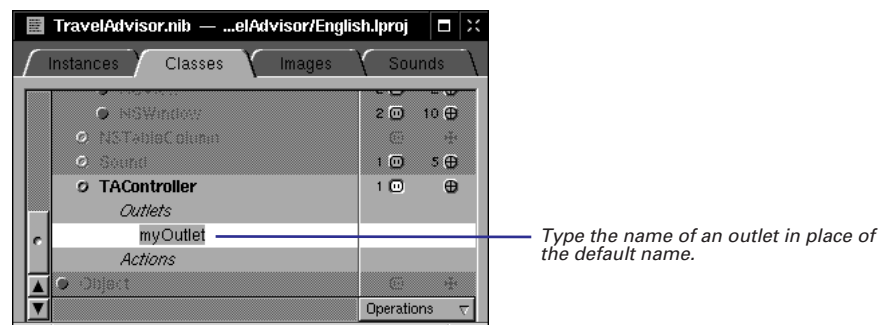
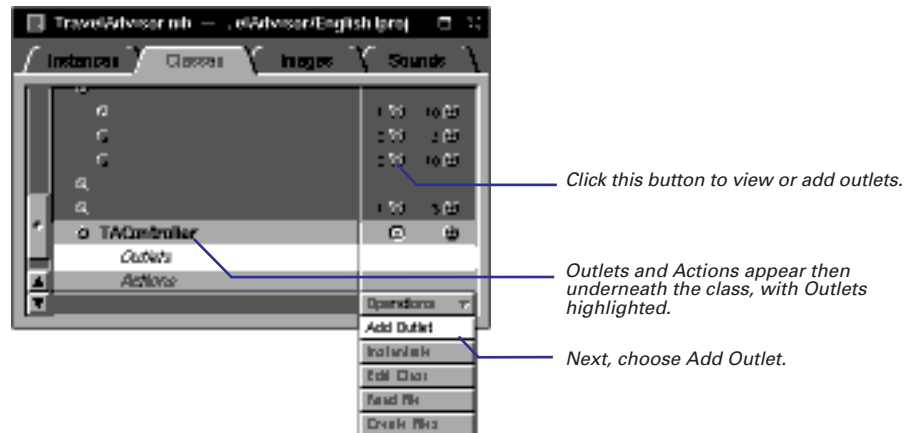
- 1 Click the button for an outlet or an action.
- 2 Select *Outlets* or *Actions*.
- 3 Choose the appropriate command from the Operations menu.
- 4 Enter the name of the outlet or action in place of the default name.

An object isolated from other objects is of little use. Interface Builder provides two ways for you to specify how objects of your class communicate with other objects: outlets and actions.

Before you begin this task, take a moment to consider what other objects you want instances of your class to send messages to, and the requests that instances of your class are apt to receive from other objects. The procedure itself is simple, and almost identical for outlets and actions.

Adding Outlets

Outlets are instance variables that identify other objects. In the Classes display, you access the outlets of a class by clicking the electrical-outlet button.

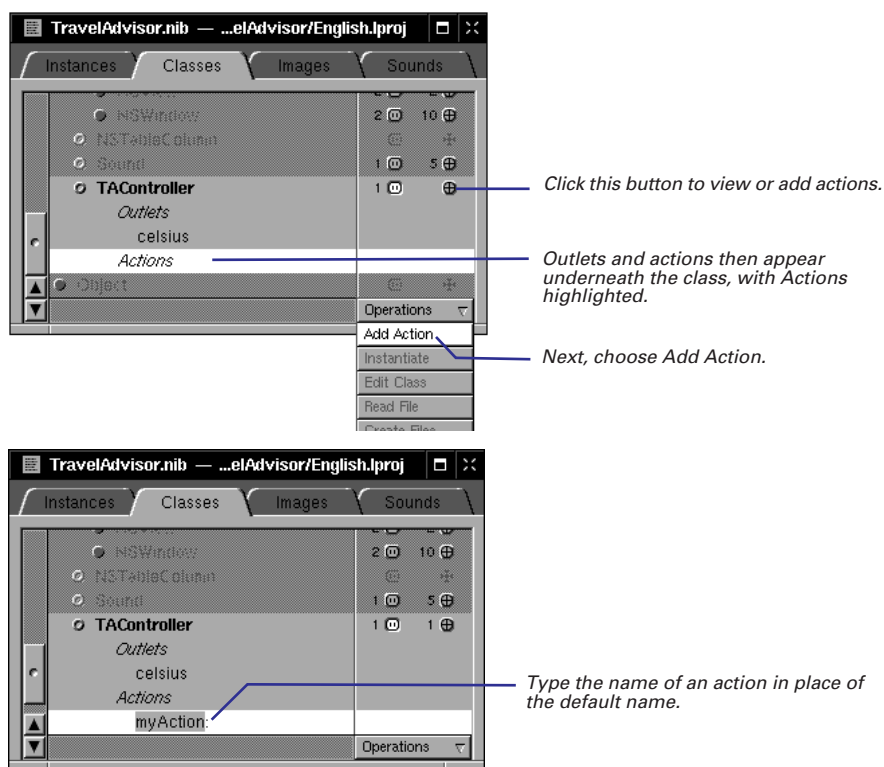


For background information on outlets and actions, see “Communicating With Other Objects: Outlets and Actions” in Chapter 4, “Making and Managing Connections.”

When you press Return, the outlet is renamed and Interface Builder highlights the new outlet. If you have another outlet to specify, choose Add Outlet again from the Operations menu and type the outlet’s name over the default name.

Adding Actions

Actions are methods invoked as a direct consequence of the manipulation of NSControl objects in the interface, such as when users click a button. In the Classes display, you access a class's actions by clicking the cross-hairs button.



When you press Return, the action is renamed and Interface Builder highlights the new action. If you have another action to specify, choose Add action from the Operations menu, and type the new action's name over the default action name ("MyAction").

When you are finished specifying outlets and actions, click the class name to collapse the list of outlets and actions.

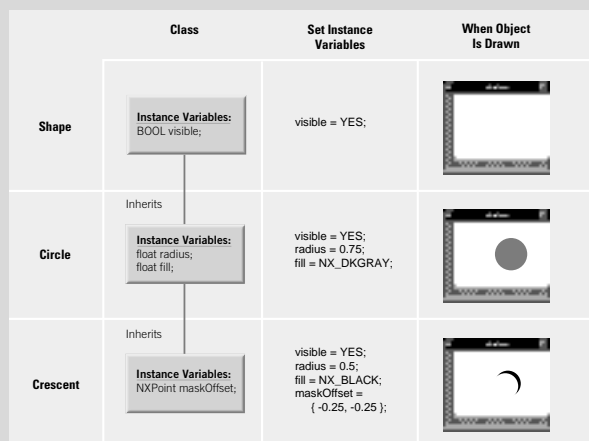
Tip: When an outlet or action (but not its text) is highlighted, you can add a new outlet or action by pressing the Return key instead of using the menu.

A Short Practical Guide to Subclassing

Subclassing is not an esoteric art but one of the most common and essential tasks in object-oriented programming. But it doesn't need to be a difficult chore, especially if you take the time to learn what's in the class hierarchy.

What is Subclassing?

The principal notion behind subclassing is inheritance. Classes stand in relation to other classes as child to parent or parent to child. A class might have many child classes (or subclasses), but always has only one parent class (superclass). At the head of this class hierarchy is the root class.



The attributes (instance variables) and behavior (methods) defined by a class are shared by all descendents of that class. To put it another way, each new class is the accumulation of all class definitions in its inheritance chain.

For example, the `NSView` class defines two instance variables for location and size (**frame** for theSuperview orientation, and **bounds** for within the view) from which all instances of its numerous subclasses derive their own basic position and dimensions. The `NSView` class also defines several methods for setting and getting these instance variables; again, all subclasses of `NSView` inherit the behavior defined by these methods. You can send the same messages to any instance of an `NSView` subclass to have it resize itself.

So subclassing is usually the extension and specialization of the inheritance chain. When you define a class that inherits from another class, you are specifying how it differs from that superclass.

But there are reasons for creating a subclass—or a “branch” of subclasses—other than getting different behavior. You may want to define a class that dispenses generic functionality to its subclasses, such as an `Output` class that performs tasks common to both a `Printer` class and a `Fax` class. You might want a class to declare methods (perhaps unimplemented) that set up a protocol that future subclasses can implement. Code reusability is an additional motive: the behavioral elements shared among classes can go into a single superclass for those classes.

Analyzing the Inheritance Chain

As the first step in subclassing you should analyze the inheritance chain. This point may seem obvious, but it is important enough to emphasize. You should do more than just identify the most suitable superclass; you want to understand exactly what it does and how it interacts with other classes.

Carefully read the specifications. Note which methods are available. Determine what the methods do and how they are related to each other; identify the accessor methods, those that get and set the instance variables; identify the interfaces to instances of other classes (such as outlets).

If the behavior you want for your class is targeted at a special problem, even if that problem is managing an application or window, it might make the most sense to subclass `NSObject`. These kind of subclasses, often called controller or model classes, are common in OpenStep applications. See “Implementing a subclass of `NSObject`” for details on creating typical controller classes. Also, see “The Model-View-Controller Paradigm” in this chapter for a description of the distinguishing characteristics of controller and model types of classes.

Instance Variables: To Add or Not to Add


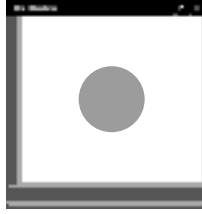
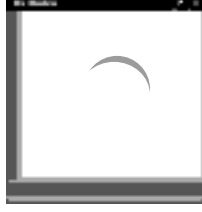
Instance variables represent an object's attributes and hold pointers to other objects (outlets). If instances of your class require special attributes or outlets, add them.

But, as a general rule, avoid adding instance variables unless they are absolutely necessary. Instance variables add weight to objects. You sometimes generate certain objects (for example, cells in a file-system browser) in large numbers. The more data these objects carry, the more memory gets consumed.

Often you can compute values from other values. Sometimes you can get pointers to other objects without having to specify outlets. Or you can represent attributes in lightweight fashion, especially if they are Boolean in nature, by encoding them as bits in an integer.

If you do not want to give subclasses of your class access to its instance variables, put the **@private** directive before the declarations of the instance variables you want to conceal. (Many instance variables are private in OpenStep classes.)

This example illustrates the effects of polymorphism and inheritance in a hypothetical class hierarchy. The Shape class provides basic functionality and a single instance variable. The Circle class, a subclass of Shape, adds more instance data and actually implements drawing. The Crescent class supplements its superclass (Circle) with more specialized behavior and data.

	Class	Set Instance Variables	When Object Is Drawn
Shape	<p>Instance Variables: BOOL visible;</p>	visible = YES;	
Circle	<p>Inherits</p> <p>Instance Variables: float radius; float fill;</p>	visible = YES; radius = 0.75; fill = NSDarkGray	
Crescent	<p>Inherits</p> <p>Instance Variables: NSPoint maskOffset;</p>	visible = YES; radius = 0.5; fill = NSBlack; maskOffset = {-0.25, -0.25};	

A Short Practical Guide to Subclassing (continued)

Determining Your Class's Methods

Look at your class from the perspective of potential clients. What will they want it to do? What information will they expect back? The answers to these and similar questions will lead to the set of methods for your class. Based on relation to superclass, methods generally come in three types:

- **Added methods** These new methods extend the class definition. They include accessor methods for new instance variables.
- **Replacement methods** These types of methods completely override the superclass method of the same name. They can also, by being a “null” implementation, block the invocation of the superclass method.
- **Extended methods** These methods also override a superclass method, but then in the implementation invoke the superclass method by calling **super**. This is a common technique for adding behavior or getting cumulative behavior (such as archiving) across the inheritance chain in response to a single message (such as **encodeWithCoder:**).

What is Public, What is Private?

When designing your subclass, also identify the code that is part of the interface and code that is private to the class.

- **Public methods** These implement your class's interface. External objects invoke these methods by sending messages to instances of your class. Among these types of methods are accessor methods, which mediate client access to instance variables. You declare public methods in the header file for your class.
- **Private methods** These methods can be invoked by objects within a project but are invisible to external objects. You usually declare them in a private header file and prefix the method name with an underscore character.
- **Functions** Non-library static C functions are also private to your class. They are marginally faster than methods because they don't involve the overhead of the run-time object system.

Use a method if you're accessing instance variables, and use a public method if that method is part of your public interface.

Alternatives to Subclassing

Sometimes you can get particular behavior without additional subclassing. OpenStep and the Objective-C language give you

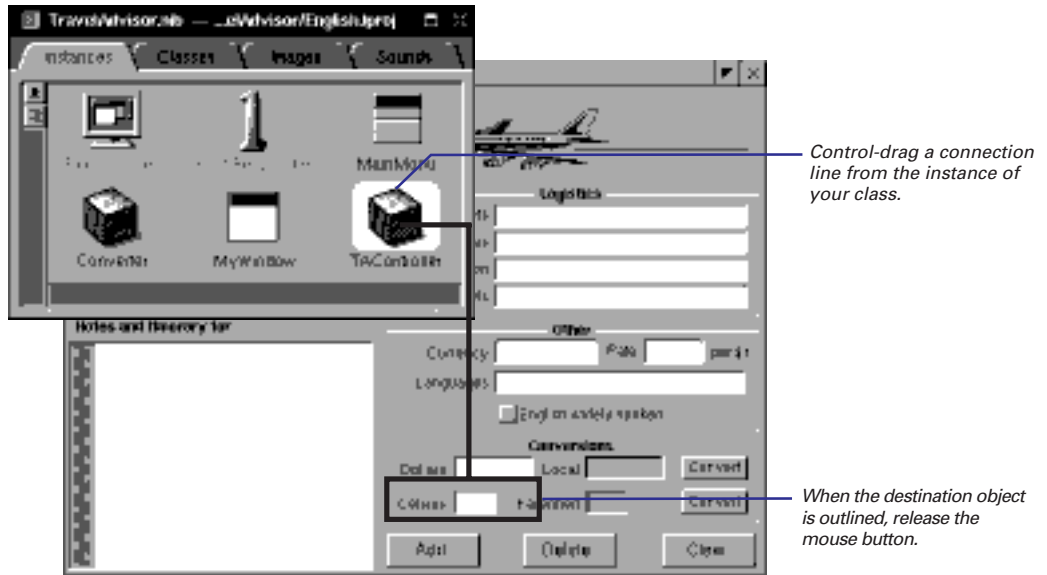
many ways to merge and synchronize your class's behavior with the behavior of OpenStep classes and even other custom classes.

- **Delegation** An object can send, on specific occasions, messages to another object registered as its delegate. If the delegate implements the methods so invoked, it can participate in the work of the object. For example, an `NSBrowser` object sends messages to its delegate requesting cells to insert into a column. Other major Application Kit classes with delegation protocols are `NSApplication`, `NSWindow`, and `NSText`.
- **Notifications** Many objects post notifications to all interested observers when a particular event takes place or is about to take place. Notifications allow observing objects to coordinate related activities and sometimes give them a chance to veto the event. This can be better than delegation because an object can have many observers but only one delegate. See the specification for `NSNotificationCenter` (a Foundation Framework class) for details on adding an observer object and on responding to notifications.
- **Protocols** A protocol is a list of method declarations associated with a particular purpose but unattached to a class definition. By adopting the protocol and implementing the methods, your class can interact with OpenStep classes and accomplish that purpose. OpenStep publishes many protocols, including those for copying objects and encoding objects for archiving.
- **Categories** These are Objective-C constructs that enable you to add methods to a class without having to subclass it. The methods become part of the class, inherited by all of its subclasses. The only major drawback is that you cannot declare new instance variables (however, you can access all existing instance variables). Besides extending a class definition, you use categories to group, manage, and configure methods in large classes.

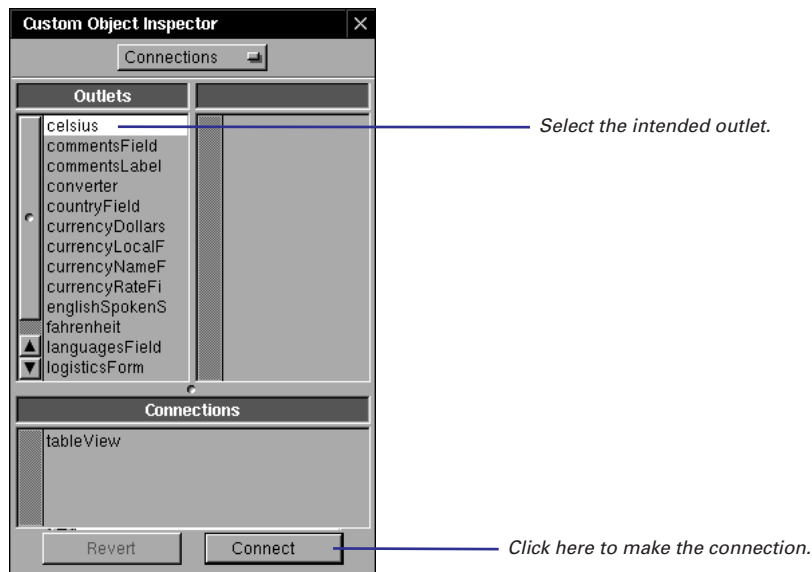
Connecting your class's outlets

- 1 **Control-drag a connection line from the instance to another object.**
- 2 **In the Inspector's Connections display, select the outlet that identifies the destination object.**
- 3 **Click the Connect button.**

You initialize an outlet in Interface Builder by making a connection from your instance to another object.



When you establish the line connection, the Inspector panel for the destination object becomes the key window. Specify the outlet identifier for this object.

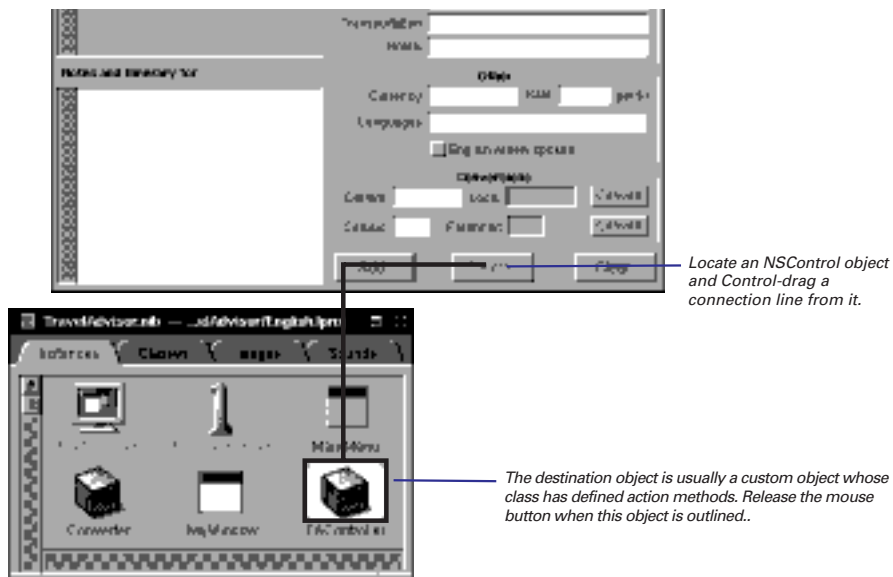


This task and the next one “Connecting your class’s actions,” summarize information more fully presented in Chapter 4, “Making and Managing Connections.”

Connecting your class's actions

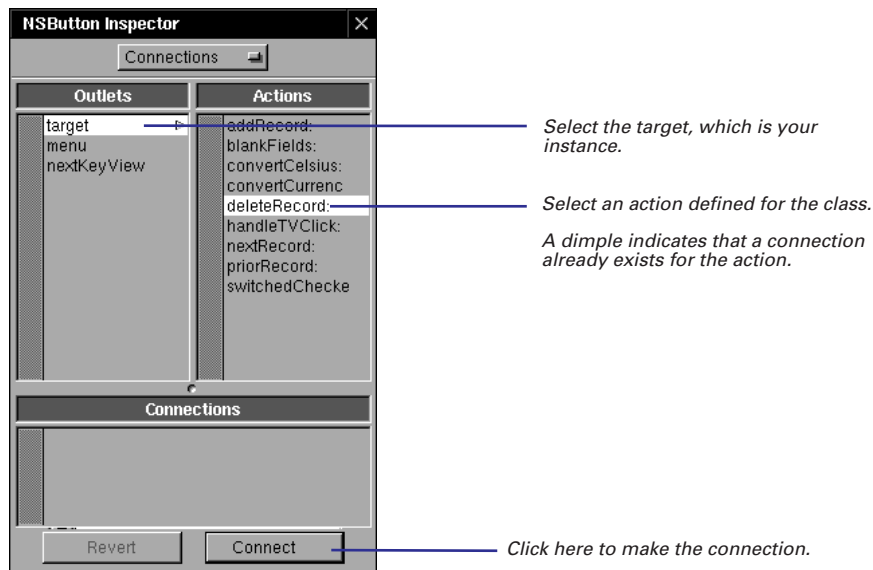
- 1 Control-drag a connection line from a Control object to your class's instance.
- 2 In the Inspector's Connections display, select the appropriate action.
- 3 Click the Connect button.

Action connections go from an NSControl object to your class's instance.



When the line is set between the objects, the second column of the Connections display shows the action methods that the target object (your instance) has declared. Select the action for this NSControl object.

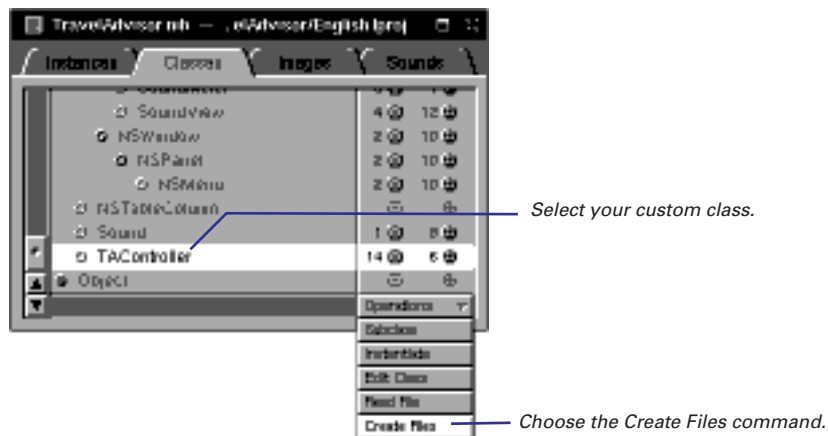
You can make connections between objects entirely within the outline mode of the Instances display. For more information on the outline mode, see Chapter 4, “Making and Managing Connections.”



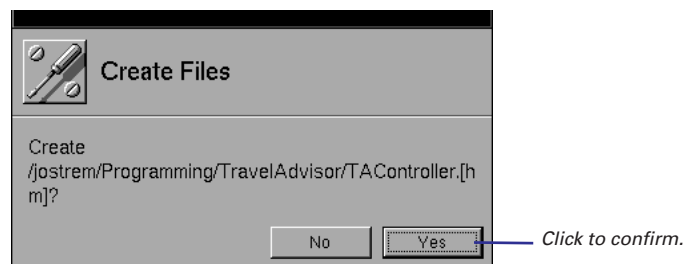
Generating source code files

- 1 **Select your class in the Classes display.**
- 2 **Choose Create Files from the Operations pull-down list.**
- 3 **Click Yes in the subsequent attention panels/message boxes.**

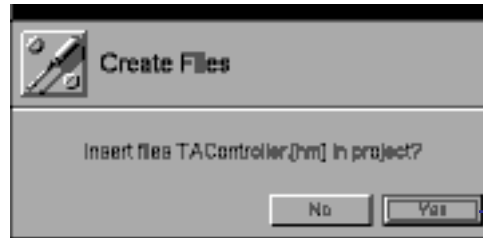
Before you begin specifying the behavior of your class in code, you typically generate template source code files for your class from the information contained in the nib file. The header file (*MyClass.h*) created by Interface Builder declares the outlets you specified as instance variables (of type *id*) and declares the actions as instance methods of the form *methodName:sender*. The implementation file (*MyClass.m*) contains empty function blocks for each of these methods.



When you generate source code files, Interface Builder displays an attention panel/message box to confirm creation of the files.



If you confirm creation and the nib file is associated with a project, another attention panel/message box subsequently asks if you wish to add the template code files to the project. Click Yes to add the files to the project.



And then they appear in Project Builder.



Implementing a subclass of NSObject

- ▶ Import header files.
- ▶ Declare new instance variables.
- ▶ Implement accessor methods.
- ▶ Define target/action behavior.
- ▶ Define initialization and deallocation behavior.
- ▶ Define how objects are copied.
- ▶ Define how objects are compared.
- ▶ Implement archiving and unarchiving.
- ▶ Define special behavior for your class.

This task summarizes the steps that you must complete—and can optionally complete—to implement a subclass of NSObject. With this kind of subclass, the subtleties arising from inherited behavior are simplified. Still, the interaction of your class with the root class is very important and applies to all subclasses.

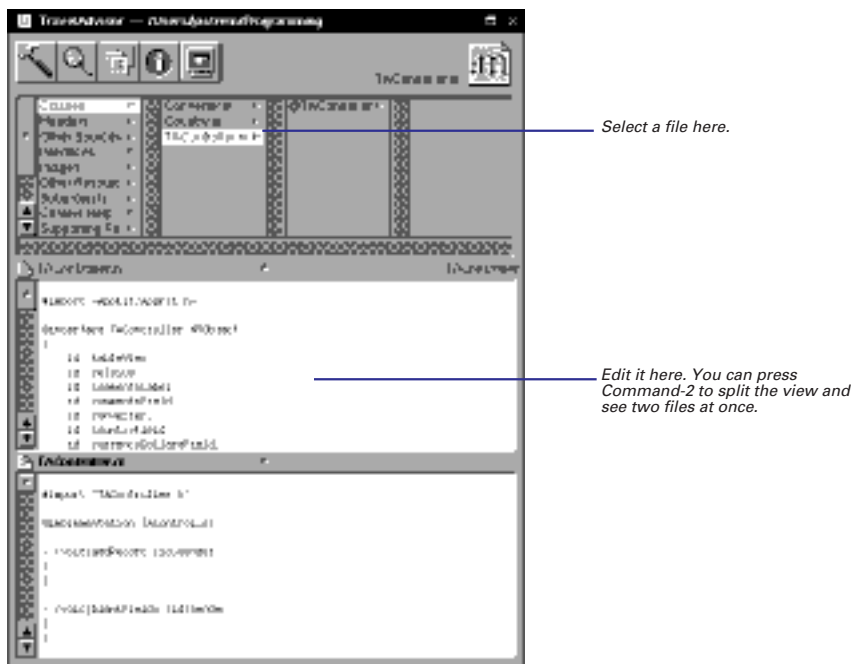
The task assumes that you have completed the following prerequisites in Interface Builder, presented earlier in this chapter:

- Naming a class, positioning it in the class hierarchy
- Specifying outlets and actions for the class
- Creating an instance of the class
- Connecting the instance to other objects through the outlets and actions
- Generating code files from the nib file

When you have generated code files in Interface Builder, switch over to the Project Builder application and open your project. Open your class's header file (*ClassName.h*) and implementation file (*ClassName.m*).

For more on the NSObject class, see its description in the *Foundation Framework Reference*.

The book *Object-Oriented Programming and the Objective-C Language* describes in detail many topics related to the NSObject class and class creation.



Importing Header Files

This step is little different from what you must do in regular C programming: At the beginning of your implementation file include the header files declaring all types and functions that your code is using, as well as the header files for all referenced classes, protocols, and methods. Instead of `#include`, however, use the `#import` directive; `#import` ensures that the file is included only once.

Remember to import your class's header file. By doing so you include the interface files for all inherited classes. To include the Application Kit classes, all you need to do is `#import <AppKit/AppKit.h>`. (Interface Builder imports both `AppKit.h` and your class header files for you automatically).

```
/* TAController.h */
#import <AppKit/AppKit.h>
#import "Country.h"
```

```
/* TAController.m (implementation file) */
#import "TAController.h"
#import "Converter.h" /* Needed in implementation, not interface */
```

Declaring New Instance Variables

The header file that Interface Builder generates declares outlets as instance variables. You might want to add new instance variables for your class to this list. All instance variables should be data that is essential to an instance of your class. They can be strings, integers, floating-point values, and other objects.

```
@interface TAController:NSObject
{
    id tableView;
    ...
    NSMutableDictionary *countryDict;
}
```

Notes on the code: In this example, the instance variable `tableView` derives from an outlet specified in Interface Builder. It is written to the header file when template files are generated. The instance variable `countryDict` has been added to identify an instance of the Foundation class `NSMutableDictionary`. Explicit typing is recommended.

Implementing Accessor Methods

Accessor methods retrieve and set the values of instance variables. They provide the encapsulation of an object's data, which only the object itself (and usually instances of subclasses) can directly access. Accessor methods mediate access to instance variables, allowing client objects to get and set values through an object's interface—that is, by sending messages.

Accessor methods that *retrieve* the value of an instance variable by convention take the same name as the instance variable. They usually have a single statement that returns the value of the instance variable. Methods that *set* the value of an instance variable by convention take the name of the instance variable (first letter capitalized) prefixed with “set.” Set methods often test passed-in values for validity before assigning them.

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

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

Notes on the code: The **name** method retrieves the value of the instance variable **name**; it simply returns the value. The **setName:** method sets the value of the instance variable **name**. Because **name** is an object, it releases the instance variable before assigning it the new value. Again because **name** is an object, the new value is copied to make sure that it remains valid.

Your class might not need to implement accessor methods if it has no need for client objects to set or retrieve the values of its objects' instance variables.

Defining Target/Action Behavior

When you defined your class in Interface Builder, you specified certain methods (*actions*) that NSControl objects in the interface invoke in your object (the *target*) when an certain user event occurs. In implementing your class, you must specify the behavior of these methods. The sole argument of action methods is **sender**, the object sending the message.

```
- (void)handleTVClick:(id)sender
{
    Country *newerRec;
    int index = [sender selectedRow];

    if (index >= 0 && index < [countryKeys count]) {
        newerRec = [countryDict objectForKey:[countryKeys
            objectAtIndex:index]];
        [self populateFields:newerRec];
        [commentsLabel setStringValue:[NSString stringWithFormat:
            @"Notes and Itinerary for %@",
            [countryField stringValue]]];
        recordNeedsSaving=NO;
        [tableView tile];
    }
    return;
}
```

Notes on the code: This method updates other fields in a window with information from an NSDictionary when the user selects a row in a table view. It uses **sender**, which identifies the NSControl object sending the message, to find out which key to use when retrieving the information from the NSDictionary.

handleTVClick: is an abbreviated version of a method you implemented if you worked through the TravelAdvisor tutorial in *Discovering OPENSTEP Programming*.

Defining Initialization and Deallocation Behavior

The NSObject class defines methods that subclasses must override to initialize their instances and to deallocate them. These methods are invoked at the start and end of an object's life. Initialization sets the initial values of instance variables and dynamically allocates and initializes variables. Deallocation frees the memory allocated to these variables.

Subclasses of NSObject almost always need to override **init** and **dealloc**. (An exception is a subclass that has no instance variables; in this case, it can rely on NSObject's implementation of **init**, which simply returns **self**.) You can define other initialization methods for your class that take arguments and perform more specialized initializations. However, a subclass of NSObject must always implement **init**, even if **init** only invokes one of these specialized initializers, passing in a default value.

Designated Initializer One of a subclass's initialization methods must be the *designated initializer*. The designated initializer invokes its superclass's designated initializer (in NSObject's case, **init**), performs most of the work, and returns **self**. The other initialization methods in a class eventually end up invoking the designated initializer.

Invoking super's Initializer Since an object's full complement of attributes includes those instance variables declared and initialized by superclasses, initialization should cascade down the inheritance chain, starting with the NSObject class. This means that initialization should almost always *begin* with the invocation of the superclass's designated initializer. For the same reason, deallocation should almost always *end* by invoking the superclass's **dealloc** method, after deallocating its own dynamically allocated instance variables. If your **dealloc** method invokes **super's dealloc** first, the object will be deallocated before it has had a chance to free its own allocated storage.

For more on designated initializers, see the description of the **init** method in the NSObject class specification in the *Foundation Framework Reference* or see *Object-Oriented Programming and the Objective-C Language*.

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

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

    return self;
}

- (void)dealloc
{
    [name release];
    [airports release];
    [airlines release];
    [transportation release];
    [hotels release];
    [languages release];
    [currencyName release];
    [comments release];

    [super dealloc];
}
```

Remember, if you create an object (such as an instance of `NSString`) in your initialization code or elsewhere, you are responsible for its deallocation (with **autorelease** or **release**). If you create an object in an initialization method, the proper place for releasing it is in **dealloc**.

See “Creating and Deallocating Different Types of Objects” in this chapter for some background. For complete details, read the introduction to the *Foundation Framework Reference*.

Notes on the code: This example shows the **init** method (which is also the designated initializer in this case) starting off by sending **init** to **super** to have its superclass (`NSObject`) complete its initializations first. It then sets the object’s instance variables to initial values (empty strings here) and returns **self**. Until it returns **self**, the object is in an unusable state. The **dealloc** method mirrors the **init** method. It releases all dynamically allocated instance variables. The **release** method decrements an object’s reference count and, if the count afterwards is zero, **dealloc** is invoked and the object is deallocated. It then invokes **super’s dealloc** method to have the superclass deallocate its own instance variables.

Defining How Objects Are Copied

If you expect that objects of your class will be copied, adopt the `NSCopying` protocol; if your class can create mutable versions of an object, also adopt the `NSMutableCopying` protocol.

```
@interface MyClass : NSObject <NSCopying, NSMutableCopying>
```

Next implement the protocol methods, `copyWithZone:` and `mutableCopyWithZone:`. These are simple implementations of these methods:

```
- (id)copyWithZone:(NSZone *)zone {
    return [[MyClass allocWithZone:zone] init];
}

- (id)mutableCopyWithZone:(NSZone *)zone {
    return [[MyMutableClass allocWithZone:zone] init];
}
```

Defining How Objects are Compared

A problem similar to copying objects is comparing objects. `NSObject`'s default behavior, in the `isEqual:` method, is to compare the identifiers of objects (their `ids`). If the `ids` of the receiving and argument objects are equal, the objects are considered equal. You might find this behavior acceptable for instances of your class, but if you don't, override `isEqual:`.

Suppose you have a class named `Color`, and this class has one instance variable, an integer which holds an industry-accepted identifier of a color. What is important in demonstrating equality of objects in this case is not the equality of `ids`, but of the values of their color instance variables.

Implementing Archiving and Unarchiving

When an object of your class has been around for awhile, responding to events and to messages from other objects, its state—the values of its instance variables—is likely to change. “Off” might change to “on,” true to false, red to green. When the user quits the application owning your object, you want to save the important parts of that object’s state and then restore them the next time the application runs. This is called archiving.

The mechanism for archiving and unarchiving objects is implemented using the classes NSCoder, NSArchiver, and NSUnarchiver and the protocol NSCodering. It encodes an application’s object in a way that enhances their persistency and distributability. The repository of this encoded object information can be a file or an NSData object. You should archive any instance variables or other data critical to an object’s state.

When a class adopts the NSCodering protocol, it receives a message requesting that it encode itself and a message asking that it decode and initialize itself. You implement two NSCodering methods to intercept these messages:

encodeWithCoder: and **initWithCoder:**.

Both **encodeWithCoder:** and **initWithCoder:** should begin by invoking the corresponding superclass method so that the superclass archives or unarchives its instance variables first. (If the class inherits directly from NSObject or any other class that does not adopt NSCodering, however, these methods should not invoke the superclass method.) The invocation of **super’s initWithCoder:** returns the partially initialized object (**self**). End **initWithCoder:** by returning **self**.

NSArchiver and NSUnarchiver provide methods that write data to and read data from the archive. Among these are **encodeObject:**, **encodeValuesOfObjCTypes:**, **decodeObject:**, and **decodeValuesOfObjCTypes:**. You send the message **encodeRootObject:** or **archiveRootObjectToFile:** to the NSArchiver class to invoke an **encodeWithCoder:** method. To invoke an **initWithCoder:** method, you send the message **unarchiveObjectWithFile:** or **decodeObject** to the NSUnarchiver class. You never invoke **encodeWithCoder:** or **initWithCoder:** directly.

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

- (id)initWithCoder:(NSCoder *)coder
{
    name = [[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];
    return self;
}
```

Notes on the code: NSCoder defines matching sets of methods for encoding and decoding objects of different types. In this example, several objects are encoded using the **encodeObject:** method and decoded using the **decodeObject:** method. One Boolean and one float variable are encoded and decoded using **encodeValueOfObjCType:** and **decodeValueOfObjCType:**, respectively. Note that the data, by type, must be decoded in the same sequence as it was encoded. The superclass method is not invoked because the class inherits directly from NSObject, which does not conform to NSCoder.

You don't need to archive every instance variable of your class. Some of these values you can re-create from scratch and others are transitory and hence unimportant (such as a seconds variable used for timing the period since a certain event). Application Kit objects configured in Interface Builder are automatically unarchived from their nib file, but only as you originally initialized them. If you want to retain some changed attribute of these objects, you should archive the attribute and then initialize the object with the unarchived attribute in the **awakeFromNib** method. (An **awakeFromNib** message is sent to each of the objects unarchived from a nib file after all of the objects in the nib file have been unarchived and all of the outlets are set.)

```

- (void)awakeFromNib
{
    [countryField selectText:self];
    ...
    [commentsField setDelegate:self];
    ...
    [currencyRateField setDelegate:self];
}

```

Notes on the code: In this implementation of `awakeFromNib`, the object must communicate with fields on its interface through the outlets `countryField`, `commentsField`, and `currencyField`. It places the cursor inside `countryField` and makes itself the delegate of the fields `commentsField` and `currencyRateField`. These initializations are done here and not in `init` because the connection between the objects must be unarchived from the nib file first.

Defining Special Behavior

The final step in implementing a subclass of NSObject is writing the methods that are special to your class, those methods that give it its distinctive behavior. This step is all up to you. If you want examples that you can use as models, look in `($NEXTROOT)/NextDeveloper/Examples`.

Other NSObject Methods You Could Override

There are several other NSObject methods that you might want to implement:

description Implement this method to return a descriptive debugging message as an NSString object. When you're debugging, `gdb` displays your message when you use the `po` command.

awakeAfterUsingCoder: Implement this method to re-initialize the object, providing it one last chance to propose another object in its place.

replacementObjectForCoder: Implement this method to substitute another object for your object during encoding.

initialize Implement this class method if you want to initialize your class before it receives its first message. This is a good place to set the version of your class (`setVersion:`).

forwardInvocation: Implement this method if you want to forward messages with unrecognized selectors to another object that can handle the message.

The Structure of Header Files and Implementation Files

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

@interface TAController:NSObject
{
    id tableView;
    ...
    BOOL recordNeedsSaving;
}

/* target/action */
- (void)addRecord:(id)sender;
- (void)deleteRecord:(id)sender;
/* housekeeping methods */
- (id)init;
- (void)awakeFromNib;
- (void)dealloc;
...
@end
```

```
#import "TAController.h"

@implementation TAController

- (void)addRecord:(id)sender
{
    /* some code here */
    return;
}

- (id)init
{
    /* some code here */
    return self;
}

/* ... */
@end
```

Header File

- Begin by importing header files for declaration types (**#import**).
- **@interface** begins class interface declaration. Class name precedes superclass, separated by a colon.
- Put the declarations of instance variables within curly braces.
- After right curly brace declare your methods.
- Action methods take the argument **sender**.
- End class interface declaration with **@end**.

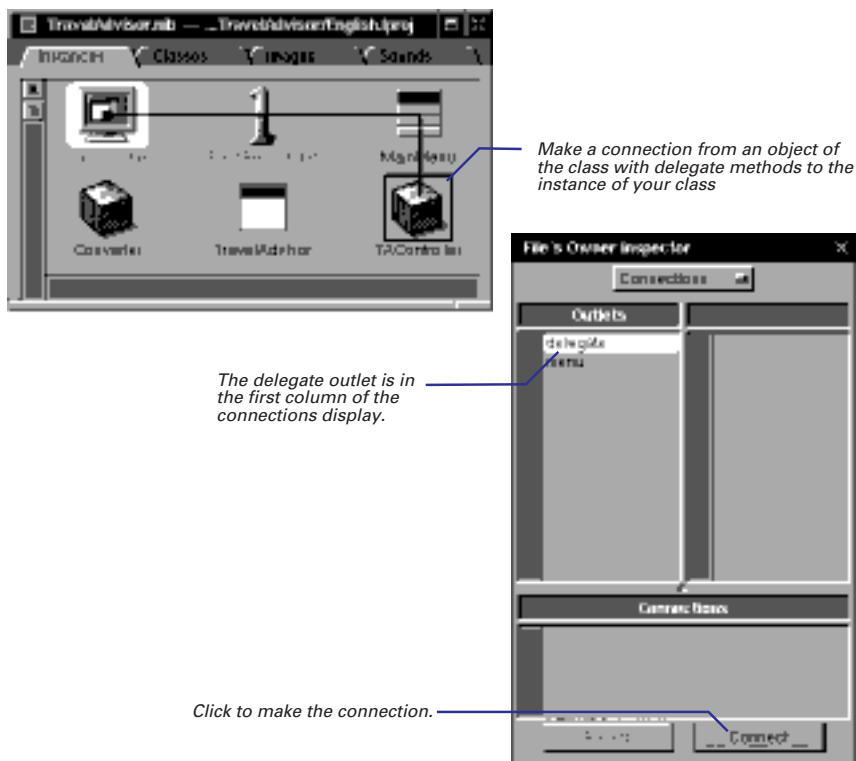
Implementation File

- Begin by importing relevant header files, especially the class header file.
- **@implementation** followed by class name begins implementation section.
- Implement all methods.
- End implementation section with **@end**.

Making your class a delegate

- 1 **Connect your instance to an object that has delegates.**
- 2 **Select the delegate outlet in the Connections inspector.**
- 3 **Click Connect.**
- 4 **Implement the delegate methods.**

Several OpenStep classes allow you to register an object as their delegate. As certain events occur, the objects send messages to their delegates, giving them the opportunity to participate in processing. In Interface Builder, you can easily designate your class's instance as a delegate.



Messages to delegates sometimes notify them of impending or just-transpired events, and sometimes request them to complete some work. Major classes with delegate methods are `NSApplication`, `NSWindow`, `NSText`, and `NSBrowser`. See the *Application Kit Reference* for details on delegate methods.

Next, implement the delegate methods you want your class to respond to. In this example, the object acting as delegate archives itself before the application terminates.

```
- (void)applicationWillTerminate:(NSNotification *)note
{
    [NSArchiver archiveRootObject:self toFile:@"TravelData"];
}
```

Tip: You can programmatically set an object's delegate by sending it the `setDelegate:` method.

Implementing a subclass of NSView

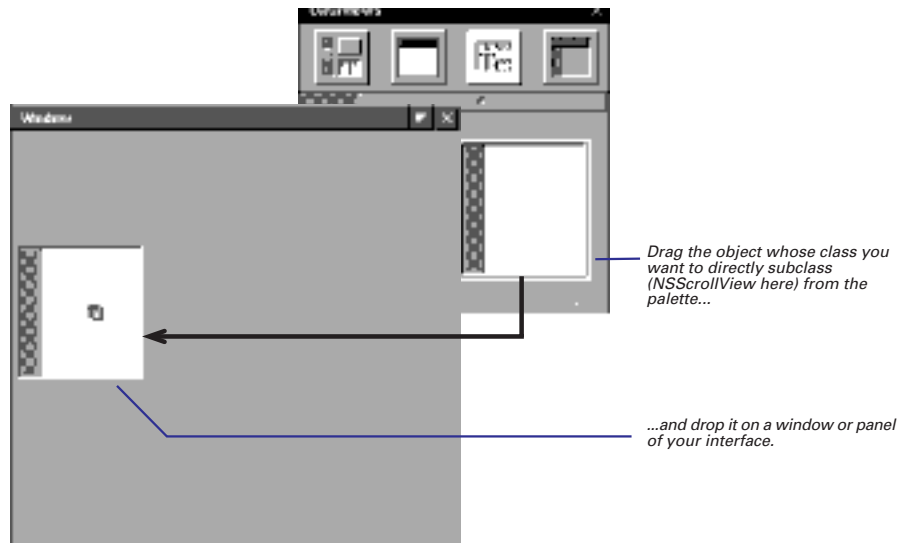
- 1 Identify the class and its outlets and actions.
- 2 Place and resize an object from the Views palette on a window or panel.
- 3 Assign your class as the class of the object.
- 4 Connect the instance to other objects in the interface.
- 5 Generate code files.
- 6 Complete programming tasks necessary for any object.
- 7 Complete programming tasks specific to NSView objects:
 - ▶ Initialize an NSView object.
 - ▶ Draw an NSView object.
 - ▶ If necessary, handle events.

Making a subclass of the NSView class is a procedure that differs from making a subclass of the NSObject class. But it starts out the same. In the Classes display of Interface Builder, choose Subclass from the Operations menu while NSView or one of its subclasses is highlighted in the browser. Then name your class and add outlets and actions.

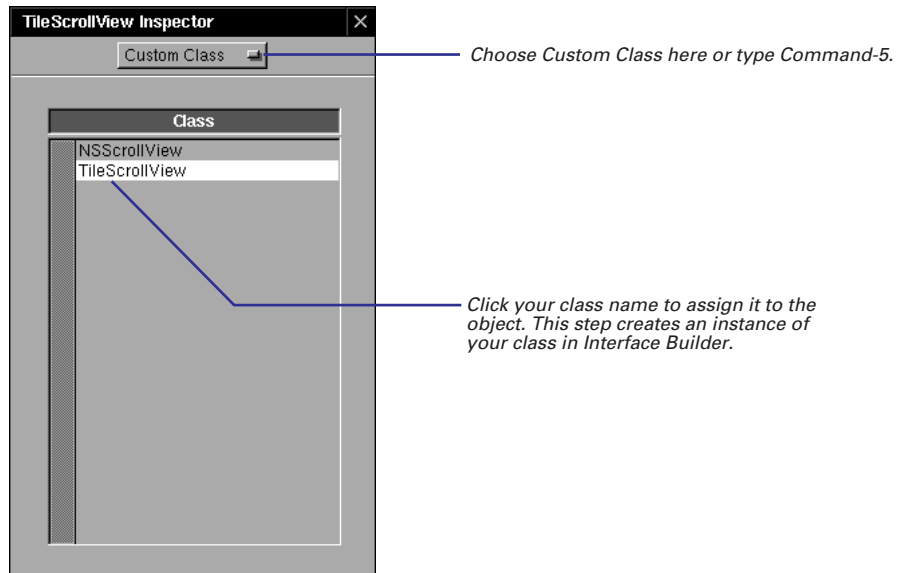


Making an Instance of an NSView Subclass

Place an instance of your class on your interface. If you're subclassing an NSView subclass (such as NSButton or NSTextField), drag the object that represents that class (that is, the button or the text field) from the palette window into your interface's window. If you're subclassing NSView directly, use the CustomView object on the Views palette.



Position and resize the object, and while it's still selected, bring up the Custom Class display of the Inspector panel by typing **Command/Control-5**. Assign a class name to the object; this creates an instance of your NSView subclass.



Tip: Make sure you choose the appropriate superclass. If you subclass an NSView subclass, rather than subclassing NSView directly, you can still set that class's attributes for your object using the Inspector panel's Attributes display. If you subclass NSView directly, you lose the ability to set attributes using the Inspector panel.

The next three steps that you must complete are the same tasks that follow the instantiation of NSObject subclasses:

- Connect the instance to other objects in the interface (“Connecting your class’s outlets” and “Connecting your class’s actions”). But now the instance appears as part of the interface, and not as an icon in the Instances display of the nib file window.
- Generate code files and have them inserted in your project (“Generating source code files”).
- Switch over to the project in Project Builder that contains the nib file, and open your class’s code files.

Since NSView inherits from NSObject, next complete some of the same programming tasks recommended for subclasses of NSObject:

- Declaring new instance variables
- Implementing accessor methods
- Implementing target/action methods
- Archiving and unarchiving

To create a functional subclass of NSView, you must complete two additional steps (and might want to complete another), which are described on the following pages.

Initializing NSView Objects

Subclasses of NSView override **initWithFrame:** instead of **init**. In **initWithFrame:** (NSView's designated initializer) you initialize a just-allocated instance of your class, setting its attributes to an initial state. The method's sole argument is the rectangle in which drawing is to occur (usually the frame of the view).

In this example, **initWithFrame:** initializes instance variables of varying types and performs other housekeeping chores.

```
- (id)initWithFrame:(NSRect)frameRect {
    [super initWithFrame:frameRect];
    glist = [[NSMutableArray allocWithZone:[self zone]]
            init];
    slist = [[NSMutableArray allocWithZone:[self zone]]
            init];
    cacheImage = [self createCacheWithSize:[self bounds].size];
    [self cache:[self bounds] andUpdateLinks:NO];
    gvFlags.grid = 10;
    gvFlags.gridDisabled = 1;
    [self allocateGState];
    gridGray = DEFAULT_GRID_GRAY;
    PSInit();
    currentGraphic = [Rectangle class];      /* default graphic */
    /* trick to allow NSApp to control currentGraphic */
    currentGraphic = [self currentGraphic];
    editView = [self createEditView];
    [[self class] initClassVars];
    [self registerForDragging];
    spellDocTag = 0;
    return self;
}
```

The NSView class offers your subclass a wealth of inherent functionality. It includes methods for managing the view hierarchy, for converting coordinates and modifying the coordinate system, for managing cursors and events, and for focusing, clipping, scrolling, dragging, and printing. See the description of the NSView class in the *Application Kit Reference*.

Notes on the code: The implementation of an **initWithFrame:** method begins by invoking **super's** **initWithFrame:** method, ends by returning **self**, and in between sets the instance variables to initial values. Often the attributes set have a visual aspect, and affect how the view is drawn.

As with NSObject subclasses, you might have to implement the **dealloc** method to deallocate dynamically allocated storage.

Drawing NSView Objects

An NSView object draws itself with the **drawRect:** method. To invoke **drawRect:**, another object must send **display** to the NSView object. The **drawRect:** method is also invoked automatically when windows are resized and exposed, when NSViews are scrolled, and when similar events happen. The NSRect argument passed to **drawRect:** indicates how much of the NSView needs to be drawn.

```

- (void)drawRect:(NSRect)rect
{
    int grid;
    float gray;

    grid = [spacing intValue];
    grid = MAX(grid, 0.0);
    PSsetgray(NSWhite);
    NSRectFill(rect);
    if (grid >= 4) {
        gray = [grayField floatValue];
        gray = MIN(gray, 1.0);
        gray = MAX(gray, 0.0);
        PSsetgray(gray);
        PSsetlinewidth(0.0);
        [self drawGrid:grid];
    }
    PSsetgray(NSBlack);
    NSFrameRect([self bounds]);
}

```

The PostScript functions and operators available for use are described in *DPSClientLibrary Reference*.

pswrap is a program that creates a C function to correspond to a sequence of PostScript code. Note that your custom **pswrap** code (extension **.psw**) must go in Project Builder under Other Sources. **pswrap** is described in detail in Adobe Systems' *pswrap Reference Manual*.

Notes on the code: The example above shows **drawRect:**. This example fills in the view with a white background, draws a grid using a user-selectable gray value, then uses **NSFrameRect()** to draw a black border around the view.

In implementing **drawRect:**, write whatever code helps to draw your NSView. You can call **pswrap**-generated functions to send PostScript code to the Window Server. You can send messages to bitmap objects, requesting them to composite source images stored in off-screen windows. You can change font styles and text colors. If your NSView uses an NSCell to do any of its drawing, you can send **drawWithFrame:inView:** or **drawInteriorWithFrame:inView:** to the NSCell within **drawRect:**.

The **drawRect:** method defines an NSView's static appearance on the screen. Your subclass can also add other methods for dynamic drawing in response to user events. In these methods you might highlight the NSView, drag it from one place to another, or animate it. The Application Kit locks focus automatically when **drawRect:** is invoked. In dynamic-drawing contexts you must lock and unlock focus yourself when drawing.

If you want your view to respond to mouse clicks, key presses, or other user events, you must do at least two things:

- Re-implement NSView's **acceptsFirstResponder** method to return YES.
- Decide which event types you want to respond to and implement the appropriate methods: **mouseUp:**, **mouseDown:**, **keyDown:**, **mouseEntered:**, and so on.

The event methods are defined in the NSResponder class, where the default implementation is to forward the event message to the next responder.

When it invokes an event method, the input system passes in an NSEvent object. This object holds details related to the event: the type of event, the mouse's location (in the window's base coordinates), the window number, a time value associated with the event, flags indicating modifier keys and mouse buttons, and supplementary data.

You can find or derive much of the information required for handling an event in the NSEvent parameter. For instance, you can convert the NSEvent mouse location to your NSView's base coordinate system with **convertPoint:fromView:**. You can check for modifier keys or mouse buttons using the keyboard-state flags masks.

The following example illustrates several of these techniques.

The NSEvent class is described in the *Application Kit Reference*.

```
- (void)mouseDown:(NSEvent *)event
{
    NSPoint p, start;
    int grid, gridCount;

    start = [event locationInWindow];
    start = [self convertPoint:start fromView:nil];
    grid = MAX([spacing intValue], 1.0);
    gridCount = (int)MAX(start.x, start.y) / grid;
    gridCount = MAX(gridCount, 1.0);

    event = [[self window] nextEventMatchingMask:
             NSLeftMouseDownMask|NSLeftMouseUpMask];
    while ([event type] != NSLeftMouseUp) {
        p = [event locationInWindow];
        p = [self convertPoint:p fromView:nil];
        grid = (int)MAX(p.x, p.y) / gridCount;
        grid = MAX(grid, 1.0);
        if (grid != [spacing intValue]) {
            [form abortEditing];
            [spacing setIntValue:grid];
            [self display];
        }
        event = [[self window] nextEventMatchingMask:
                NSLeftMouseDownMask|NSLeftMouseUpMask];
    }
}
```

Tip: If you want your `NSView` to handle target/action messages sent to the First Responder (for example, copy and paste), be sure to override `acceptsFirstResponder` to return YES, and then implement the appropriate methods (`copy:` and `paste:`).

Creating and Deallocating Different Types of Objects

As you create objects, you need to make sure that they are going to be deallocated eventually, and you also need to make this doesn't happen until you don't need the object anymore. You do this by sending messages that increment and decrement the object's reference count, a count of how many objects refer to it. When and how you should do this depends on when and how you create the object.

The Autorelease Pool

OpenStep uses an autorelease pool to automatically deallocate objects. When you send an **autorelease** message to an object, it adds the object to the autorelease pool. At the top of the event loop, the pool sends every object in it the **release** message. **release** decrements the reference count. If the reference count becomes 0, it deallocates the object (by sending **dealloc**).

Application projects automatically have an autorelease pool, just as they automatically have an event loop. If you're working on a non-Application project, you can create an autorelease pool by creating an instance of the Foundation `NSAutoreleasePool` class. (See its specification in the *Foundation Framework Reference*.)

Temporary Objects

If you create an object inside a method and you want that object to go away after the method has finished executing, use a **+classname** method (so called because their names begin with the name of the class minus the NS prefix) to create the object. These methods allocate the object (which increments the reference count), initialize it, and send it an **autorelease** message so that it is deallocated at the top of the event loop. For example, this `NSNumber` object will exist only for one event cycle:

```
NSNumber *intObject = [NSNumber
    numberWithInt:anInt];
```

The methods **alloc**, **copy**, and **mutableCopy** increment an object's reference count, so if you use one of these to create a temporary object, be sure to send that same object an **autorelease** message.

Instance Variables

Objects that are instance variables should be created when an object is initialized and not go away until that object is deallocated. If you use a **+classname** method to create an instance variable, it will be deallocated at the top of the event loop. To prevent this, send **retain** to the object immediately after you create it. **retain** increments the reference count. Another way to make sure that an instance variable is not deallocated is to use the **alloc** method directly (or **copy** or **mutableCopy**) to create it.

No matter which method you use to create the instance variable, send it a **release** message in your object's **dealloc** method to indicate that you're done with it.

Sometimes you have two objects with instance variables that refer to each other. In this case, only one of the objects should retain the other. For example, an `NSView` object has a `Superview` and one or more `Subviews`, each pointing to other `NSView` objects. If an `NSView` object retained both its `Superview` and its `Subviews`, no `NSView` would ever be deallocated. The `Superview` won't release its `Subview` instance variables until it is deallocated, and it can't be deallocated because the `Subviews` don't release the `Superview` until they are deallocated. For this reason, `NSView` objects retain their `Subviews`, but not their `Superviews`.

As a rule of thumb, if your application has a similar object hierarchy, the "parent" object should retain its "children," but the children should not retain their parents.

Custom Objects Created in Interface Builder

If you create a custom object that does not inherit from `NSView` or `NSWindow` in Interface Builder, send it a **release** message in your object's **dealloc** method. Custom objects have a retain count of 1 when they're unarchived from the nib file.

NSView Objects Created in Interface Builder

Views created in Interface Builder are retained and released automatically. `Superviews` retain all `Subviews` as they are added to the hierarchy and release them as they are removed. If you swap views in and out of the hierarchy or move views from one window to another, you should **retain** the views that are not in the hierarchy (and **release** them either after you add them to the hierarchy or in **dealloc**).

NSWindow Objects Created in Interface Builder

Windows created in Interface Builder are not released until the user quits the application. If you want a window to be released when the user closes it, set the "Release when closed" attribute in Interface Builder.

For more on this topic, see the introduction to the *Foundation Framework Reference*.

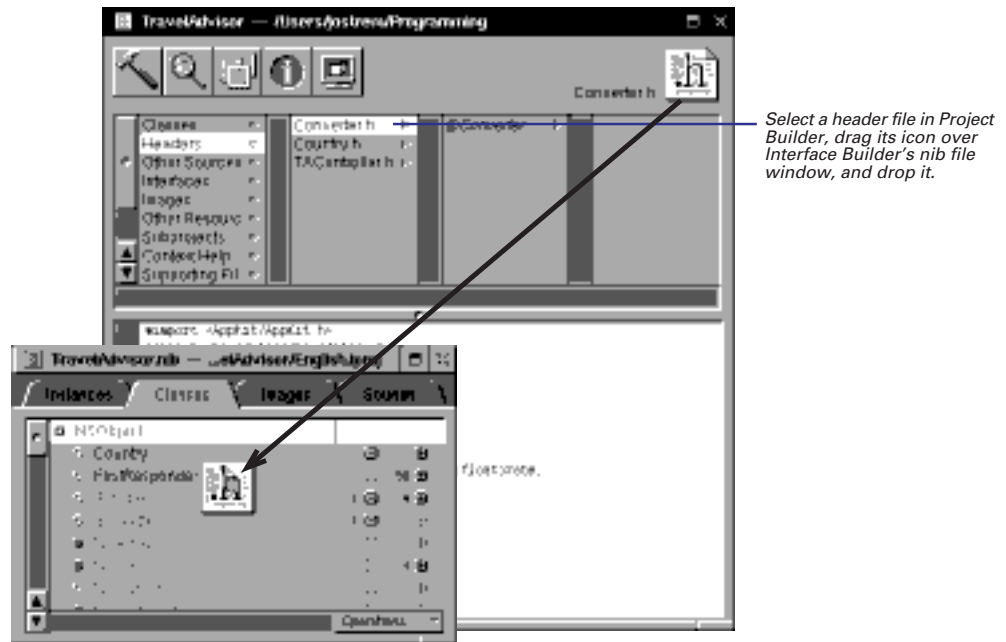
Adding existing classes to your nib file

- ▶ **Drag the header file from the File Viewer/Desktop, File Manager, or Project Builder into the nib file window.**

Or

- ▶ **Copy a class in one nib file and paste it in another.**

The easiest way to add a class to your nib file is to drag the header file for an existing custom class from the Workspace Manager's File Viewer/Desktop or File Manager or from Project Builder's main window into Interface Builder.



The new class appears in the Classes display under its subclass and with its outlets and actions defined. After adding the class, you must still connect it to other objects through its outlets and actions. To do this, complete these steps:

- Make an instance of the class (for `NSView` subclasses, that means assigning your class to a view object).
- Connect the instance's outlets and actions to other objects in the nib file.

If you are going to write a header file and then drag the file into Interface Builder, follow the conventions for header files described in “The Structure of Header Files and Implementation Files,” in this chapter.

Creating Classes in Project Builder

Instead of defining a class in Interface Builder and using Create Files to create the source code, you can create the source code in Project Builder first and add the class to Interface Builder later. To create a class in Project Builder, use the File ► New in Project command to create template source code files, write your code, then drag the header file into the nib file window. When you create a class in this manner, any method you've defined that takes a single argument named **sender** and that returns **id** or **void** is considered an action. Any instance variable that is type **id** or has the **IBOutlet** keyword prefixed to its declaration is an outlet.

```
#import <AppKit/AppKit.h>
#define IBOutlet          /* Needed to avoid compiler errors. */

@interface TAController:NSObject
{
    IBOutlet NSTableView *tableView;          /* an outlet */
    id commentsLabel;          /* another outlet */
    ...
    NSMutableDictionary *countryDict;      /* not an outlet. */
    ...
}

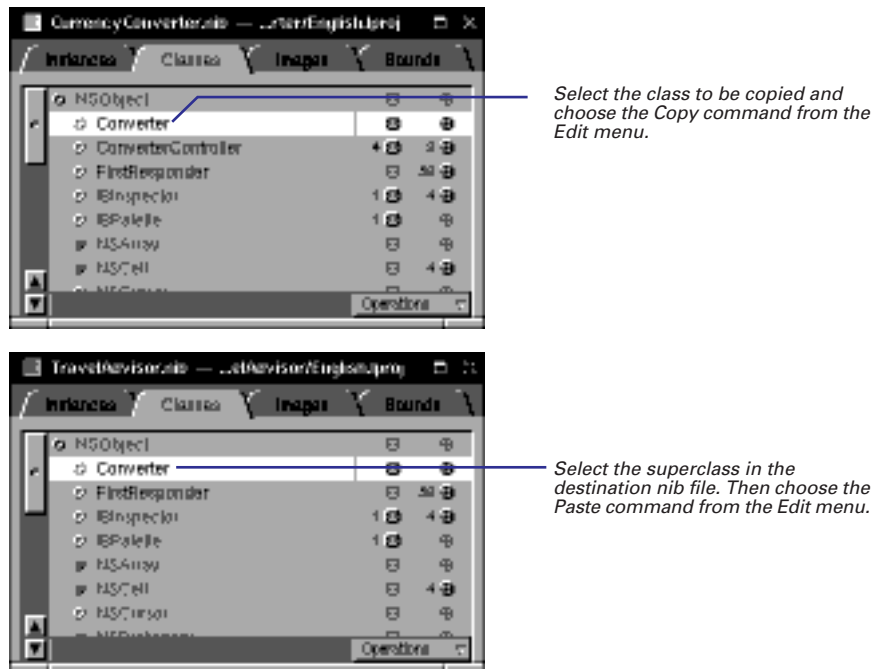
/* target/action */
- (void)addRecord:(NSButton *)sender;          /* an action */
- (void)convertCelsius:(id)sender;          /* another action */

/* Data read and write methods */
- (void)populateFields:(Country *)aRec;      /* not an action */

@end
```

Copying Classes Between Nib Files

You can copy class definitions between nib files, in the same or different projects, by copying a class in one nib file and pasting it into another nib file.

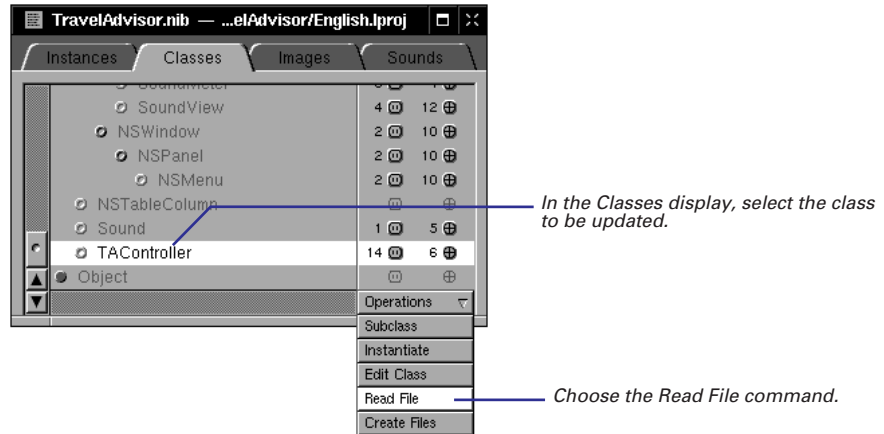


A duplicate of the original class appears in the Classes display of the destination nib file. Generate an instance of the class in the destination nib file and connect it to other objects in the nib file through its outlets and actions.

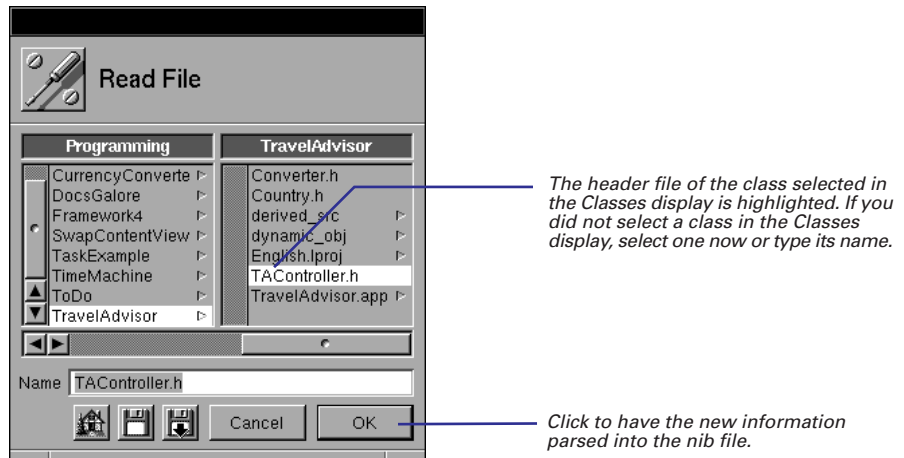
Updating a class definition

- Choose the Read File command and select a header file in the Open panel/dialog.

If you later add outlets and actions to the header file, or delete them from it, Interface Builder allows you to update the nib file with this new information.



Interface Builder brings up an Open panel/dialog for you to confirm (or select) the class definition to update.



If there are any new outlets and actions, remember to connect these outlets and actions to other objects in the nib file.

Tip: You can also use the Read File command to add an existing class to a nib file, or you create a header file in Project Builder using the New in Project command and read it into a nib file using this command.