Copyright ©1996 by NeXT Software, Inc. All Rights Reserved.

;MakingYourClassADelegate.rtfd;;¬ Next Section ;GeneratingSourceCodeFiles.rtfd;;¬ Previous Section

ImplementingasubclassofNSObject;¬Implementing a subclass of NSObject

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

This task summarizes the steps that you must completeÐand can optionally completeÐto 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:

SquareBullet.eps ¬Naming a class, positioning it in the class hierarchy
SquareBullet.eps ¬Specifying outlets and actions for the class
SquareBullet.eps ¬Creating an instance of the class
SquareBullet.eps ¬Connecting the instance to other objects through the outlets and actions
SquareBullet.eps ¬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**).

_ImplementingSubclassNSObject.eps ¬

For more on the NSObject class, see its description in the *Foundation Framework Reference.* *;/NextLibrary/Frameworks/Foundation.framework/Resources/English.lproj/Documentation/Reference/Classes/NSObject.rtfd;;¬*

The book *Object-Oriented Programming and the Objective-C Language* describes in detail many topics related to the NSObject class and class creation. ;/NextLibrary/Documentation/NextDev/TasksAndConcepts/ObjectiveC/0_IntroObjC.rtf;;¬

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 interfaceÐthat 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 ªset.º 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*.
> ;/NextLibrary/Documentation/NextDev/TasksAndConcepts/DeveloperTutorial/0_Contents.rtfd;;¬

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.

> For more on designated initializers, see the description of the **init** method in the NSObject class specification in the *Foundation Framework Reference*
> *;/NextLibrary/Frameworks/Foundation.framework/Resources/English.lproj/Documentation/Reference/Classes/NSObject.rtfd;;¬* or see *Object-Oriented Programming and the Objective-C Language*.
> ;/NextLibrary/Documentation/NextDev/TasksAndConcepts/ObjectiveC/0_IntroObjC.rtf;;¬

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.

```
- (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];
}
```

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.

> Remember, if you create an object (such as a 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 ªCreating and Deallocating Different Types of Objectsº in this chapter for some background. ;SubclassingConcepts.rtfd;CreatingandDeallocatingDifferentTypesofObjects;¬ For complete details, read the introduction to the *Foundation Framework Reference*.

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 **id**s). If the **id**s 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 **id**s, 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 stateÐthe values of its instance variablesÐis likely to change. ªOffº might change to ªon,º 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 NSCoding. 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 NSCoding protocol, it receives a message requesting that it encode itself and a message asking that it decode and initialize itself. You implement two NSCoding 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 NSCoding, 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 **archiveRootObject:toFile:** 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 NSCoding.

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 **/NextDeveloper/Examples**.

Related Concept: ;SubclassingConcepts.rtfd;TheStructureofHeaderFilesandImplementationFiles;¬

The Structure of Header Files and Implementation Files

Related Concept: **;SubclassingConcepts.rtfd;OtherNSObjectMethodsYouCouldOverride;¬** Other NSObject Methods You Could Override