Concepts in Objective-C Programming

Developer

Contents

About the Basic Programming Concepts for Cocoa and Cocoa Touch 7
At a Glance 7
How to Use This Document 7
Prerequisites 8
See Also 8

Class Clusters 9
Without Class Clusters: Simple Concept but Complex Interface 9
With Class Clusters: Simple Concept and Simple Interface 10
Creating Instances 10
Class Clusters with Multiple Public Superclasses 11
Creating Subclasses Within a Class Cluster 12
    A True Subclass 12
    True Subclasses: An Example 14
    A Composite Object 16
    A Composite Object: An Example 16

Class Factory Methods 20

Delegates and Data Sources 22
How Delegation Works 22
The Form of Delegation Messages 24
Delegation and the Application Frameworks 25
    Becoming the Delegate of a Framework Class 26
    Locating Objects Through the delegate Property 27
Data Sources 27
Implementing a Delegate for a Custom Class 27

Introspection 29
Evaluating Inheritance Relationships 29
Method Implementation and Protocol Conformance 30
Object Comparison 31

Object Allocation 34

Object Initialization 35
The Form of Initializers 35
Issues with Initializers 36
Implementing an Initializer 38
Multiple Initializers and the Designated Initializer 40

Model-View-Controller 43
Roles and Relationships of MVC Objects 43
Model Objects Encapsulate Data and Basic Behaviors 43
View Objects Present Information to the User 44
Controller Objects Tie the Model to the View 44
Combining Roles 45
Types of Cocoa Controller Objects 45
MVC as a Compound Design Pattern 47
Design Guidelines for MVC Applications 50
Model-View-Controller in Cocoa (OS X) 52

Object Modeling 53
Entities 53
Attributes 54
Relationships 55
Relationship Cardinality and Ownership 56
Accessing Properties 57
Keys 57
Values 57
Key Paths 58

Object Mutability 60
Why Mutable and Immutable Object Variants? 60
Programming with Mutable Objects 62
Creating and Converting Mutable Objects 62
Storing and Returning Mutable Instance Variables 63
Receiving Mutable Objects 64
Mutable Objects in Collections 66

Outlets 67

Receptionist Pattern 68
The Receptionist Design Pattern in Practice 68
When to Use the Receptionist Pattern 71

Target-Action 73
The Target 73
The Action 74
Target-Action in the AppKit Framework 75
Controls, Cells, and Menu Items 75
Setting the Target and Action 77
Actions Defined by AppKit 78
Target-Action in UIKit 78

Toll-Free Bridging 80

Document Revision History 83

Figures, Tables, and Listings

Class Clusters 9

Figure 1-1 A simple hierarchy for number classes 9
Figure 1-2 A more complete number class hierarchy 10
Figure 1-3 Class cluster architecture applied to number classes 10
Figure 1-4 An object that embeds a cluster object 16
Table 1-1 Class clusters and their public superclasses 11
Table 1-2 Derived methods and their possible implementations 13

Delegates and Data Sources 22

Figure 3-1 The mechanism of delegation 23
Figure 3-2 A more realistic sequence involving a delegate 23
Listing 3-1 Sample delegation methods with return values 24
Listing 3-2 Sample delegation methods returning `void` 24

Introspection 29

Listing 4-1 Using the `class` and `superclass` methods 29
Listing 4-2 Using `isKindOfClass:` 30
Listing 4-3 Using `respondsToSelector:` 31
Listing 4-4 Using `conformsToProtocol:` 31
Listing 4-5 Using isEqual: 32
Listing 4-6 Overriding `isEqual:` 32

Object Initialization 35

Figure 6-1 Initialization up the inheritance chain 39
Figure 6-2 Interactions of secondary and designated initializers 41

Model-View-Controller 43

Figure 7-1 Traditional version of MVC as a compound pattern 48
Figure 7-2 Cocoa version of MVC as a compound design pattern 48
Figure 7-3 Coordinating controller as the owner of a nib file 50

Object Modeling 53

Figure 8-1 Employee management application object diagram 54
Figure 8-2 Employees table view 55

Figure 8-3 Relationships in the employee management application 56
Figure 8-4 Relationship cardinality 56
Figure 8-5 Object graph for the employee management application 58
Figure 8-6 Employees table view showing department name 59

Object Mutability 60
Listing 9-1 Returning an immutable copy of a mutable instance variable 63
Listing 9-2 Making a snapshot of a potentially mutable object 65

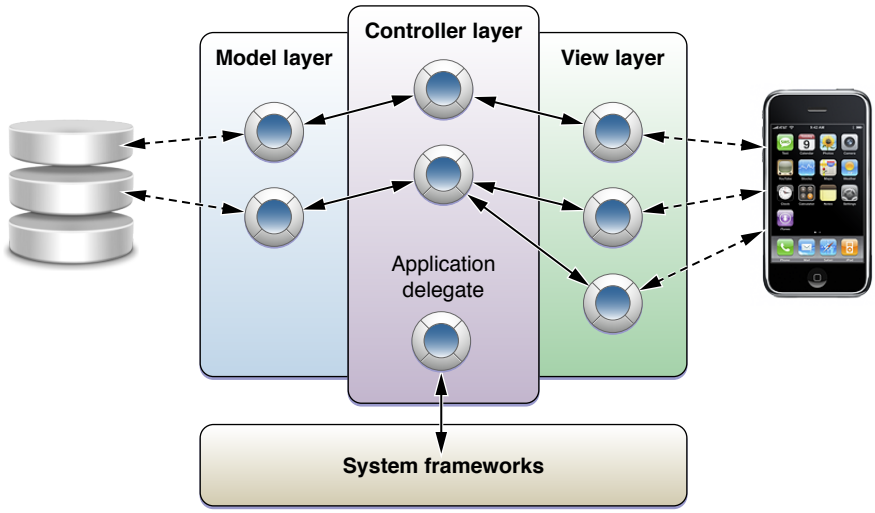
Receptionist Pattern 68
Figure 11-1 Bouncing KVO updates to the main operation queue 69
Listing 11-1 Declaring the receptionist class 69
Listing 11-2 The class factory method for creating a receptionist object 70
Listing 11-3 Handling the KVO notification 71
Listing 11-4 Creating a receptionist object 71

Target-Action 73
Figure 12-1 How the target-action mechanism works in the control-cell architecture 76

Toll-Free Bridging 80
Table 13-1 Data types that can be used interchangeably between Core Foundation and Foundation 81

About the Basic Programming Concepts for Cocoa and Cocoa Touch

Many of the programmatic interfaces of the Cocoa and Cocoa Touch frameworks only make sense only if you are aware of the concepts on which they are based. These concepts express the rationale for many of the core designs of the frameworks. Knowledge of these concepts will illuminate your software-development practices.



At a Glance

This document contains articles that explain central concepts, design patterns, and mechanisms of the Cocoa and Cocoa Touch frameworks. The articles are arranged in alphabetical order.

How to Use This Document

If you read this document cover-to-cover, you learn important information about Cocoa and Cocoa Touch application development. However, most readers come to the articles in this document in one of two ways:

- Other documents—especially those that are intended for novice iOS and OS X developers—which link to these articles.
- In-line mini-articles (which appear when you click a dash-underlined word or phrase) that have a link to an article as a “Definitive Discussion.”

Prerequisites

Prior programming experience, especially with object-oriented languages, is recommended.

See Also

Programming with Objective-C offers further discussion of many of the language-related concepts covered in this document.

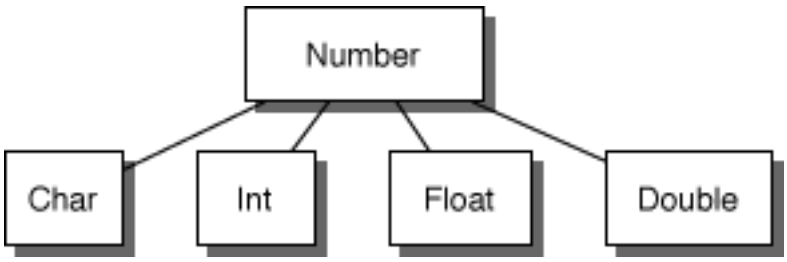
Class Clusters

Class clusters are a design pattern that the Foundation framework makes extensive use of. Class clusters group a number of private concrete subclasses under a public abstract superclass. The grouping of classes in this way simplifies the publicly visible architecture of an object-oriented framework without reducing its functional richness. Class clusters are based on the Abstract Factory design pattern.

Without Class Clusters: Simple Concept but Complex Interface

To illustrate the class cluster architecture and its benefits, consider the problem of constructing a class hierarchy that defines objects to store numbers of different types (`char`, `int`, `float`, `double`). Because numbers of different types have many features in common (they can be converted from one type to another and can be represented as strings, for example), they could be represented by a single class. However, their storage requirements differ, so it's inefficient to represent them all by the same class. Taking this fact into consideration, one could design the class architecture depicted in Figure 1-1 to solve the problem.

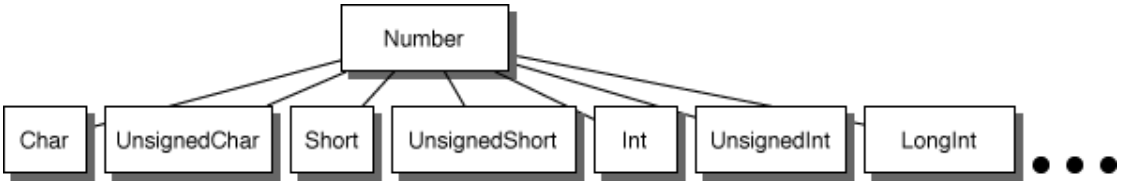
Figure 1-1 A simple hierarchy for number classes



`Number` is the abstract superclass that declares in its methods the operations common to its subclasses. However, it doesn't declare an instance variable to store a number. The subclasses declare such instance variables and share in the programmatic interface declared by `Number`.

So far, this design is relatively simple. However, if the commonly used modifications of these basic C types are taken into account, the class hierarchy diagram looks more like Figure 1-2.

Figure 1-2 A more complete number class hierarchy

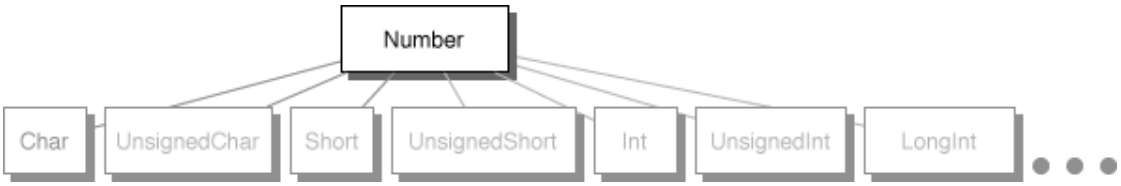


The simple concept—creating a class to hold number values—can easily burgeon to over a dozen classes. The class cluster architecture presents a design that reflects the simplicity of the concept.

With Class Clusters: Simple Concept and Simple Interface

Applying the class cluster design pattern to this problem yields the class hierarchy in Figure 1-3 (private classes are in gray).

Figure 1-3 Class cluster architecture applied to number classes



Users of this hierarchy see only one public class, `Number`, so how is it possible to allocate instances of the proper subclass? The answer is in the way the abstract superclass handles instantiation.

Creating Instances

The abstract superclass in a class cluster must declare methods for creating instances of its private subclasses. It's the superclass's responsibility to dispense an object of the proper subclass based on the creation method that you invoke—you don't, and can't, choose the class of the instance.

In the Foundation framework, you generally create an object by invoking a *+className* `...` method or the `alloc...` and `init...` methods. Taking the Foundation framework's `NSNumber` class as an example, you could send these messages to create number objects:

```
NSNumber *aChar = [NSNumber numberWithChar:'a'];
NSNumber *anInt = [NSNumber numberWithInt:1];
NSNumber *aFloat = [NSNumber numberWithFloat:1.0];
NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

You are not responsible for releasing the objects returned from factory methods. Many classes also provide the standard `alloc...` and `init...` methods to create objects that require you to manage their deallocation.

Each object returned—`aChar`, `anInt`, `aFloat`, and `aDouble`—may belong to a different private subclass (and in fact does). Although each object's class membership is hidden, its interface is public, being the interface declared by the abstract superclass, `NSNumber`. Although it is not precisely correct, it's convenient to consider the `aChar`, `anInt`, `aFloat`, and `aDouble` objects to be instances of the `NSNumber` class, because they're created by `NSNumber` class methods and accessed through instance methods declared by `NSNumber`.

Class Clusters with Multiple Public Superclasses

In the example above, one abstract public class declares the interface for multiple private subclasses. This is a class cluster in the purest sense. It's also possible, and often desirable, to have two (or possibly more) abstract public classes that declare the interface for the cluster. This is evident in the Foundation framework, which includes the clusters listed in Table 1-1.

Table 1-1 Class clusters and their public superclasses

Class cluster	Public superclasses
`NSData`	`NSData`
	`NSMutableData`
`NSArray`	`NSArray`
	`NSMutableArray`
`NSDictionary`	`NSDictionary`
	`NSMutableDictionary`
`NSString`	`NSString`
	`NSMutableString`

Other clusters of this type also exist, but these clearly illustrate how two abstract nodes cooperate in declaring the programmatic interface to a class cluster. In each of these clusters, one public node declares methods that all cluster objects can respond to, and the other node declares methods that are only appropriate for cluster objects that allow their contents to be modified.

This factoring of the cluster's interface helps make an object-oriented framework's programmatic interface more expressive. For example, imagine an object representing a book that declares this method:

```
- (NSString *)title;
```

The book object could return its own instance variable or create a new string object and return that—it doesn't matter. It's clear from this declaration that the returned string can't be modified. Any attempt to modify the returned object will elicit a compiler warning.

Creating Subclasses Within a Class Cluster

The class cluster architecture involves a trade-off between simplicity and extensibility: Having a few public classes stand in for a multitude of private ones makes it easier to learn and use the classes in a framework but somewhat harder to create subclasses within any of the clusters. However, if it's rarely necessary to create a subclass, then the cluster architecture is clearly beneficial. Clusters are used in the Foundation framework in just these situations.

If you find that a cluster doesn't provide the functionality your program needs, then a subclass may be in order. For example, imagine that you want to create an array object whose storage is file-based rather than memory-based, as in the `NSArray` class cluster. Because you are changing the underlying storage mechanism of the class, you'd have to create a subclass.

On the other hand, in some cases it might be sufficient (and easier) to define a class that embeds within it an object from the cluster. Let's say that your program needs to be alerted whenever some data is modified. In this case, creating a simple class that wraps a data object that the Foundation framework defines may be the best approach. An object of this class could intervene in messages that modify the data, intercepting the messages, acting on them, and then forwarding them to the embedded data object.

In summary, if you need to manage your object's storage, create a true subclass. Otherwise, create a composite object, one that embeds a standard Foundation framework object in an object of your own design. The following sections give more detail on these two approaches.

A True Subclass

A new class that you create within a class cluster must:

- Be a subclass of the cluster's abstract superclass
- Declare its own storage
- Override all initializer methods of the superclass
- Override the superclass's primitive methods (described below)

Because the cluster's abstract superclass is the only publicly visible node in the cluster's hierarchy, the first point is obvious. This implies that the new subclass will inherit the cluster's interface but no instance variables, because the abstract superclass declares none. Thus the second point: The subclass must declare any instance variables it needs. Finally, the subclass must override any method it inherits that directly accesses an object's instance variables. Such methods are called *primitive methods* .

A class's primitive methods form the basis for its interface. For example, take the `NSArray` class, which declares the interface to objects that manage arrays of objects. In concept, an array stores a number of data items, each of which is accessible by index. `NSArray` expresses this abstract notion through its two primitive methods, `count` and `objectAtIndex:`. With these methods as a base, other methods—*derived methods* —can be implemented; Table 1-2 gives two examples of derived methods.

Table 1-2 Derived methods and their possible implementations

Derived Method	Possible Implementation
`lastObject`	Find the last object by sending the array object this message: `[self objectAtIndex: ([self count] –1)]`.
`containsObject:`	Find an object by repeatedly sending the array object an `objectAtIndex:` message, each time incrementing the index until all objects in the array have been tested.

The division of an interface between primitive and derived methods makes creating subclasses easier. Your subclass must override inherited primitives, but having done so can be sure that all derived methods that it inherits will operate properly.

The primitive-derived distinction applies to the interface of a fully initialized object. The question of how `init...` methods should be handled in a subclass also needs to be addressed.

In general, a cluster's abstract superclass declares a number of `init...` and `+ className` methods. As described in "Creating Instances" (page 10), the abstract class decides which concrete subclass to instantiate based your choice of `init...` or `+ className` method. You can consider that the abstract class declares these methods for the convenience of the subclass. Since the abstract class has no instance variables, it has no need of initialization methods.

Your subclass should declare its own `init...` (if it needs to initialize its instance variables) and possibly `+ className` methods. It should not rely on any of those that it inherits. To maintain its link in the initialization chain, it should invoke its superclass's designated initializer within its own designated initializer method. It should also override all other inherited initializer methods and implement them to behave in a reasonable manner. (See "Multiple Initializers and the Designated Initializer" (page 40) for a discussion of designated initializers.) Within a class cluster, the designated initializer of the abstract superclass is always `init`.

True Subclasses: An Example

Let's say that you want to create a subclass of `NSArray`, named `MonthArray`, that returns the name of a month given its index position. However, a `MonthArray` object won't actually store the array of month names as an instance variable. Instead, the method that returns a name given an index position (`objectAtIndex:`) will return constant strings. Thus, only twelve string objects will be allocated, no matter how many `MonthArray` objects exist in an application.

The `MonthArray` class is declared as:

```
#import <foundation/foundation.h>
@interface MonthArray : NSArray
{
}

+ monthArray;
- (unsigned)count;
- (id)objectAtIndex:(unsigned)index;

@end
```

Note that the `MonthArray` class doesn't declare an `init...` method because it has no instance variables to initialize. The `count` and `objectAtIndex:` methods simply cover the inherited primitive methods, as described above.

The implementation of the `MonthArray` class looks like this:

```
#import "MonthArray.h"

@implementation MonthArray
```

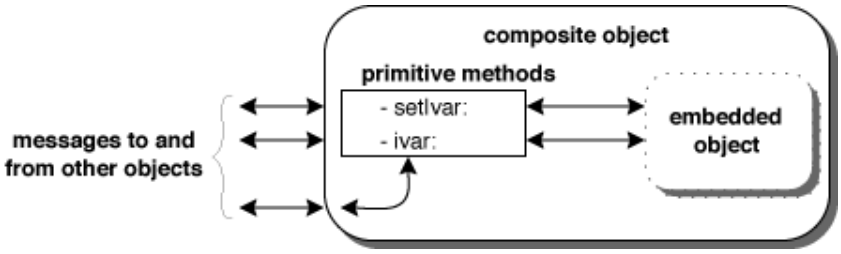
```
static MonthArray *sharedMonthArray = nil;
static NSString *months[] = { @"January", @"February", @"March",
    @"April", @"May", @"June", @"July", @"August", @"September",
    @"October", @"November", @"December" };

+ monthArray
{
    if (!sharedMonthArray) {
        sharedMonthArray = [[MonthArray alloc] init];
    }
    return sharedMonthArray;
}

- (unsigned)count
{
 return 12;
}

- objectAtIndex:(unsigned)index
{
    if (index >= [self count])
        [NSException raise:NSRangeException format:@"***%s: index
            (%d) beyond bounds (%d)", sel_getName(_cmd), index,
            [self count] - 1];
    else
        return months[index];
}

@end
```

Because `MonthArray` overrides the inherited primitive methods, the derived methods that it inherits will work properly without being overridden. `NSArray`'s `lastObject`, `containsObject:`, `sortedArrayUsingSelector:`, `objectEnumerator`, and other methods work without problems for `MonthArray` objects.

A Composite Object

By embedding a private cluster object in an object of your own design, you create a composite object. This composite object can rely on the cluster object for its basic functionality, only intercepting messages that the composite object wants to handle in some particular way. This architecture reduces the amount of code you must write and lets you take advantage of the tested code provided by the Foundation Framework. Figure 1-4 depicts this architecture.

Figure 1-4 An object that embeds a cluster object



The composite object must declare itself to be a subclass of the cluster's abstract superclass. As a subclass, it must override the superclass's primitive methods. It can also override derived methods, but this isn't necessary because the derived methods work through the primitive ones.

The `count` method of the `NSArray` class is an example; the intervening object's implementation of a method it overrides can be as simple as:

```
- (unsigned)count {
    return [embeddedObject count];
}
```

However, your object could put code for its own purposes in the implementation of any method it overrides.

A Composite Object: An Example

To illustrate the use of a composite object, imagine you want a mutable array object that tests changes against some validation criteria before allowing any modification to the array's contents. The example that follows describes a class called `ValidatingArray`, which contains a standard mutable array object. `ValidatingArray` overrides all of the primitive methods declared in its superclasses, `NSArray` and `NSMutableArray`. It also declares the `array`, `validatingArray`, and `init` methods, which can be used to create and initialize an instance:

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

```
@interface ValidatingArray : NSMutableArray
{
    NSMutableArray *embeddedArray;
}

+ validatingArray;
- init;
- (unsigned)count;
- objectAtIndex:(unsigned)index;
- (void)addObject:object;
- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
- (void)removeLastObject;
- (void)insertObject:object atIndex:(unsigned)index;
- (void)removeObjectAtIndex:(unsigned)index;

@end
```

The implementation file shows how, in an `init` method of the `ValidatingArray`class , the embedded object is created and assigned to the `embeddedArray` variable. Messages that simply access the array but don't modify its contents are relayed to the embedded object. Messages that could change the contents are scrutinized (here in pseudocode) and relayed only if they pass the hypothetical validation test.

```
#import "ValidatingArray.h"

@implementation ValidatingArray

- init
{
    self = [super init];
    if (self) {
        embeddedArray = [[NSMutableArray allocWithZone:[self zone]] init];
    }
    return self;
}
```

```
+ validatingArray
{
    return [[[self alloc] init] autorelease];
}

- (unsigned)count
{
    return [embeddedArray count];
}

- objectAtIndex:(unsigned)index
{
    return [embeddedArray objectAtIndex:index];
}

- (void)addObject:object
{
    if (/* modification is valid */) {
        [embeddedArray addObject:object];
    }
}

- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
{
    if (/* modification is valid */) {
        [embeddedArray replaceObjectAtIndex:index withObject:object];
    }
}

- (void)removeLastObject;
{
    if (/* modification is valid */) {
        [embeddedArray removeLastObject];
```

```
    }
}
- (void)insertObject:object atIndex:(unsigned)index;
{
    if (/* modification is valid */) {
        [embeddedArray insertObject:object atIndex:index];
    }
}
- (void)removeObjectAtIndex:(unsigned)index;
{
    if (/* modification is valid */) {
        [embeddedArray removeObjectAtIndex:index];
    }
}
```

Class Factory Methods

Class factory methods are implemented by a class as a convenience for clients. They combine allocation and initialization in one step and return the created object. However, the client receiving this object does not own the object and thus (per the object-ownership policy) is not responsible for releasing it. These methods are of the form `+ (`*type*`)`*className*`...` (where *className* excludes any prefix).

Cocoa provides plenty of examples, especially among the "value" classes. `NSDate` includes the following class factory methods:

```
+ (id)dateWithTimeIntervalSinceNow:(NSTimeInterval)secs;
+ (id)dateWithTimeIntervalSinceReferenceDate:(NSTimeInterval)secs;
+ (id)dateWithTimeIntervalSince1970:(NSTimeInterval)secs;
```

And `NSData` offers the following factory methods:

```
+ (id)dataWithBytes:(const void *)bytes length:(unsigned)length;
+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length;
+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length
        freeWhenDone:(BOOL)b;
+ (id)dataWithContentsOfFile:(NSString *)path;
+ (id)dataWithContentsOfURL:(NSURL *)url;
+ (id)dataWithContentsOfMappedFile:(NSString *)path;
```

Factory methods can be more than a simple convenience. They can not only combine allocation and initialization, but the allocation can inform the initialization. As an example, let's say you must initialize a collection object from a property-list file that encodes any number of elements for the collection (`NSString` objects, `NSData` objects, `NSNumber` objects, and so on). Before the factory method can know how much memory to allocate for the collection, it must read the file and parse the property list to determine how many elements there are and what object type these elements are.

Another purpose for a class factory method is to ensure that a certain class (`NSWorkspace`, for example) vends a singleton instance. Although an `init...` method could verify that only one instance exists at any one time in a program, it would require the prior allocation of a "raw" instance and then, in memory-managed code, would have to release that instance. A factory method, on the other hand, gives you a way to avoid blindly allocating memory for an object that you might not use, as in the following example:

```
static AccountManager *DefaultManager = nil;

+ (AccountManager *)defaultManager {
    if (!DefaultManager) DefaultManager = [[self allocWithZone:NULL] init];
    return DefaultManager;
}
```

Delegates and Data Sources

A delegate is an object that acts on behalf of, or in coordination with, another object when that object encounters an event in a program. The delegating object is often a responder object—that is, an object inheriting from `NSResponder` in AppKit or `UIResponder` in UIKit—that is responding to a user event. The delegate is an object that is delegated control of the user interface for that event, or is at least asked to interpret the event in an application-specific manner.

To better appreciate the value of delegation, it helps to consider an off-the-shelf Cocoa object such as a text field (an instance of `NSTextField` or `UITextField`) or a table view (an instance of `NSTableView` or `UITableView`). These objects are designed to fulfill a specific role in a generic fashion; a window object in the AppKit framework, for example, responds to mouse manipulations of its controls and handles such things as closing, resizing, and moving the physical window. This restricted and generic behavior necessarily limits what the object can know about how an event affects (or will affect) something elsewhere in the application, especially when the affected behavior is specific to your application. Delegation provides a way for your custom object to communicate application-specific behavior to the off-the-shelf object.

The programming mechanism of delegation gives objects a chance to coordinate their appearance and state with changes occurring elsewhere in a program, changes usually brought about by user actions. More importantly, delegation makes it possible for one object to alter the behavior of another object without the need to inherit from it. The delegate is almost always one of your custom objects, and by definition it incorporates application-specific logic that the generic and delegating object cannot possibly know itself.

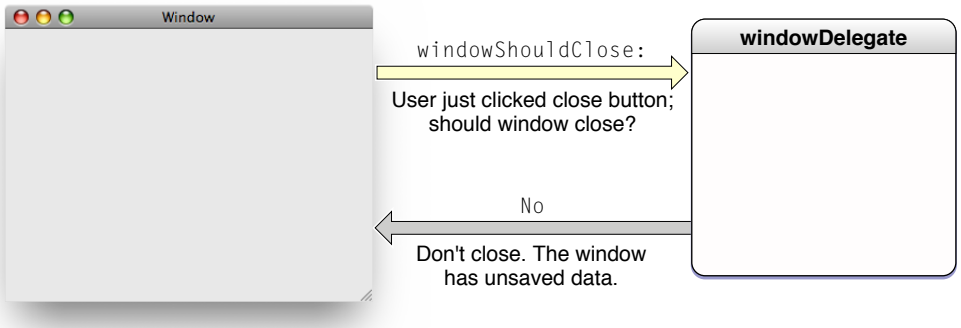
How Delegation Works

The design of the delegation mechanism is simple—see Figure 3-1 (page 23). The delegating class has an outlet or property, usually one that is named `delegate`; if it is an outlet, it includes methods for setting and accessing the value of the outlet. It also declares, without implementing, one or more methods that constitute a formal protocol or an informal protocol. A formal protocol that uses optional methods—a feature of Objective-C 2.0—is the preferred approach, but both kinds of protocols are used by the Cocoa frameworks for delegation.

In the informal protocol approach, the delegating class declares methods on a category of `NSObject`, and the delegate implements only those methods in which it has an interest in coordinating itself with the delegating object or affecting that object's default behavior. If the delegating class declares a formal protocol, the delegate may choose to implement those methods marked optional, but it must implement the required ones.

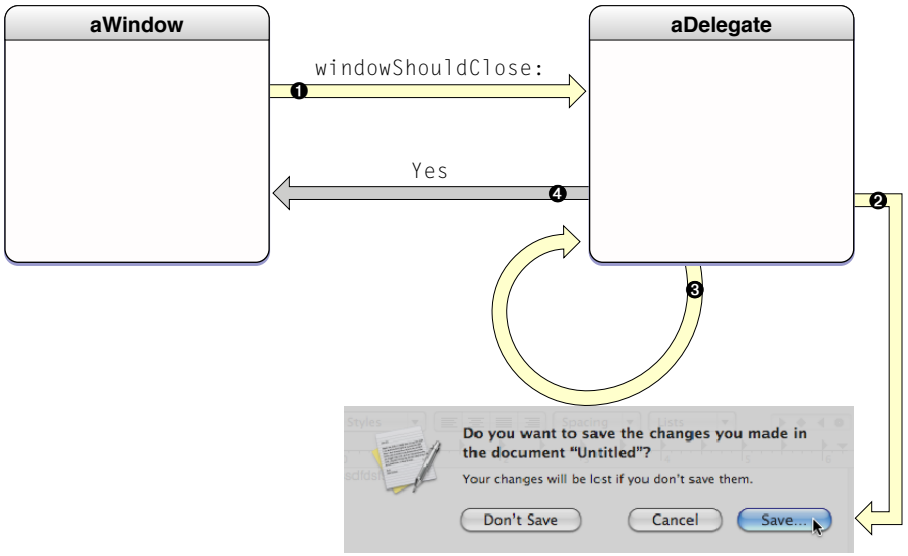
Delegation follows a common design, illustrated by Figure 3-1.

Figure 3-1 The mechanism of delegation



The methods of the protocol mark significant events handled or anticipated by the delegating object. This object wants either to communicate these events to the delegate or, for impending events, to request input or approval from the delegate. For example, when a user clicks the close button of a window in OS X, the window object sends the `windowShouldClose:` message to its delegate; this gives the delegate the opportunity to veto or defer the closing of the window if, for example, the window has associated data that must be saved (see Figure 3-2).

Figure 3-2 A more realistic sequence involving a delegate



The delegating object sends a message only if the delegate implements the method. It makes this discovery by invoking the `NSObject`method `respondsToSelector:` in the delegate first.

The Form of Delegation Messages

Delegation methods have a conventional form. They begin with the name of the AppKit or UIKit object doing the delegating—application, window, control, and so on; this name is in lower-case and without the "NS" or "UI" prefix. Usually (but not always) this object name is followed by an auxiliary verb indicative of the temporal status of the reported event. This verb, in other words, indicates whether the event is about to occur ("Should" or "Will") or whether it has just occurred ("Did" or "Has"). This temporal distinction helps to categorize those messages that expect a return value and those that don't. Listing 3-1 includes a few AppKit delegation methods that expect a return value.

Listing 3-1 Sample delegation methods with return values

```
- (BOOL)application:(NSApplication *)sender
    openFile:(NSString *)filename;                          // NSApplication
- (BOOL)application:(UIApplication *)application
    handleOpenURL:(NSURL *)url;                            // UIApplicationDelegate
- (UITableRowIndexSet *)tableView:(NSTableView *)tableView
    willSelectRows:(UITableRowIndexSet *)selection;        // UITableViewDelegate
- (NSRect)windowWillUseStandardFrame:(NSWindow *)window
    defaultFrame:(NSRect)newFrame;                          // NSWindow
```

The delegate that implements these methods can block the impending event (by returning `NO` in the first two methods) or alter a suggested value (the index set and the frame rectangle in the last two methods). It can even defer an impending event; for example, the delegate implementing the `applicationShouldTerminate:` method can delay application termination by returning `NSTerminateLater`.

Other delegation methods are invoked by messages that don't expect a return value and so are typed to return `void`. These messages are purely informational, and the method names often contain "Did", "Will", or some other indication of a transpired or impending event. Listing 3-2 shows a few examples of these kinds of delegation method.

Listing 3-2 Sample delegation methods returning `void`

```
- (void) tableView:(NSTableView*)tableView
    mouseDownInHeaderOfTableColumn:(NSTableColumn *)tableColumn;     // NSTableView
- (void)windowDidMove:(NSNotification *)notification;                  // NSWindow
- (void)application:(UIApplication *)application
    willChangeStatusBarFrame:(CGRect)newStatusBarFrame;                //
UIApplication
```

```
- (void)applicationWillBecomeActive:(NSNotification *)notification;   //
NSApplication
```

There are a couple of things to note about this last group of methods. The first is that an auxiliary verb of "Will" (as in the third method) does not necessarily mean that a return value is expected. In this case, the event is imminent and cannot be blocked, but the message gives the delegate an opportunity to prepare the program for the event.

The other point of interest concerns the second and last method declarations in Listing 3-2 . The sole parameter of each of these methods is an `NSNotification` object, which means that these methods are invoked as the result of the posting of a particular notification. For example, the `windowDidMove:` method is associated with the `NSWindow` notification `NSWindowDidMoveNotification`. It's important to understand the relationship of notifications to delegation messages in AppKit. The delegating object automatically makes its delegate an observer of all notifications it posts. All the delegate needs to do is implement the associated method to get the notification.

To make an instance of your custom class the delegate of an AppKit object, simply connect the instance to the `delegate` outlet or property in Interface Builder. Or you can set it programmatically through the delegating object's `setDelegate:` method or `delegate` property, preferably early on, such as in the `awakeFromNib` or `applicationDidFinishLaunching:` method.

Delegation and the Application Frameworks

The delegating object in a Cocoa or Cocoa Touch application is often a responder object such as a `UIApplication`, `NSWindow`, or `NSTableView` object. The delegate object itself is typically, but not necessarily, an object, often a custom object, that controls some part of the application (that is, a coordinating controller object). The following AppKit classes define a delegate:

- `NSApplication`
- `NSBrowser`
- `NSControl`
- `NSDrawer`
- `NSFontManager`
- `NSFontPanel`
- `NSMatrix`
- `NSOutlineView`
- `NSSplitView`

- `NSTableView`
- `NSTabView`
- `NSText`
- `NSTextField`
- `NSTextView`
- `NSWindow`

The UIKit framework also uses delegation extensively and always implements it using formal protocols. The application delegate is extremely important in an application running in iOS because it must respond to application-launch, application-quit, low-memory, and other messages from the application object. The application delegate must adopt the `UIApplicationDelegate` protocol.

Delegating objects do not (and should not) retain their delegates. However, clients of delegating objects (applications, usually) are responsible for ensuring that their delegates are around to receive delegation messages. To do this, they may have to retain the delegate in memory-managed code. This precaution applies equally to data sources, notification observers, and targets of action messages. Note that in a garbage-collection environment, the reference to the delegate is strong because the retain-cycle problem does not apply.

Some AppKit classes have a more restricted type of delegate called a *modal delegate*. Objects of these classes (`NSOpenPanel`, for example) run modal dialogs that invoke a handler method in the designated delegate when the user clicks the dialog's OK button. Modal delegates are limited in scope to the operation of the modal dialog.

Becoming the Delegate of a Framework Class

A framework class or any other class that implements delegation declares a `delegate` property and a protocol (usually a formal protocol). The protocol lists the required and optional methods that the delegate implements. For an instance of your class to function as the delegate of a framework object, it must do the following:

- Set your object as the delegate (by assigning it to the `delegate` property). You can do this programmatically or through Interface Builder.
- If the protocol is formal, declare that your class adopts the protocol in the class definition. For example:

```
@interface MyControllerClass : UIViewController <UIAlertViewDelegate> {
```

- Implement all required methods of the protocol and any optional methods that you want to participate in.

Locating Objects Through the delegate Property

The existence of delegates has other programmatic uses. For example, with delegates it is easy for two coordinating controllers in the same program to find and communicate with each other. For example, the object controlling the application overall can find the controller of the application's inspector window (assuming it's the current key window) using code similar to the following:

```
id winController = [[NSApp keyWindow] delegate];
```

And your code can find the application-controller object—by definition, the delegate of the global application instance—by doing something similar to the following:

```
id appController = [NSApp delegate];
```

Data Sources

A data source is like a delegate except that, instead of being delegated control of the user interface, it is delegated control of data. A data source is an outlet held by `NSView` and `UIView` objects such as table views and outline views that require a source from which to populate their rows of visible data. The data source for a view is usually the same object that acts as its delegate, but it can be any object. As with the delegate, the data source must implement one or more methods of an informal protocol to supply the view with the data it needs and, in more advanced implementations, to handle data that users directly edit in such views.

As with delegates, data sources are objects that must be present to receive messages from the objects requesting data. The application that uses them must ensure their persistence, retaining them if necessary in memory-managed code.

Data sources are responsible for the persistence of the objects they hand out to user-interface objects. In other words, they are responsible for the memory management of those objects. However, whenever a view object such as an outline view or table view accesses the data from a data source, it retains the objects as long as it uses the data. But it does not use the data for very long. Typically it holds on to the data only long enough to display it.

Implementing a Delegate for a Custom Class

To implement a delegate for your custom class, complete the following steps:

- Declare the delegate accessor methods in your class header file.

```
- (id)delegate;
- (void)setDelegate:(id)newDelegate;
```

- Implement the accessor methods. In a memory-managed program, to avoid retain cycles, the setter method should not retain or copy your delegate.

```
- (id)delegate {
    return delegate;
}

- (void)setDelegate:(id)newDelegate {
    delegate = newDelegate;
}
```

In a garbage-collected environment, where retain cycles are not a problem, you should not make the delegate a weak reference (by using the `__weak` type modifier). For more on retain cycles, see *Advanced Memory Management Programming Guide*. For more on weak references in garbage collection, see "Garbage Collection for Cocoa Essentials" in *Garbage Collection Programming Guide*.

- Declare a formal or informal protocol containing the programmatic interface for the delegate. Informal protocols are categories on the `NSObject` class. If you declare a formal protocol for your delegate, make sure you mark groups of optional methods with the `@optional` directive.

  "The Form of Delegation Messages" (page 24) gives advice for naming your own delegation methods.

- Before invoking a delegation method, make sure the delegate implements it by sending it a `respondsToSelector:` message.

```
- (void)someMethod {
    if ( [delegate respondsToSelector:@selector(operationShouldProceed)]
 ) {
        if ( [delegate operationShouldProceed] ) {
            // do something appropriate
        }
    }
}
```

The precaution is necessary only for optional methods in a formal protocol or methods of an informal protocol.

Introspection

Introspection is a powerful feature of object-oriented languages and environments, and introspection in Objective-C and Cocoa is no exception. Introspection refers to the capability of objects to divulge details about themselves as objects at runtime. Such details include an object's place in the inheritance tree, whether it conforms to a specific protocol, and whether it responds to a certain message. The `NSObject` protocol and class define many introspection methods that you can use to query the runtime in order to characterize objects.

Used judiciously, introspection makes an object-oriented program more efficient and robust. It can help you avoid message-dispatch errors, erroneous assumptions of object equality, and similar problems. The following sections show how you might effectively use the `NSObject` introspection methods in your code.

Evaluating Inheritance Relationships

Once you know the class an object belongs to, you probably know quite a bit about the object. You might know what its capabilities are, what attributes it represents, and what kinds of messages it can respond to. Even if after introspection you are unfamiliar with the class to which an object belongs, you now know enough to not send it certain messages.

The `NSObject` protocol declares several methods for determining an object's position in the class hierarchy. These methods operate at different granularities. The `class` and `superclass` instance methods, for example, return the `Class` objects representing the class and superclass, respectively, of the receiver. These methods require you to compare one `Class` object with another. Listing 4-1 gives a simple (one might say trivial) example of their use.

Listing 4-1 Using the `class` and `superclass` methods

```
// ...
while ( id anObject = [objectEnumerator nextObject] ) {
    if ( [self class] == [anObject superclass] ) {
        // do something appropriate...
    }
}
```

> **Note:** Sometimes you use the `class` or `superclass` methods to obtain an appropriate receiver for a class message.

More commonly, to check an object's class affiliation, you would send it a `isKindOfClass:` or `isMemberOfClass:` message. The former method returns whether the receiver is an instance of a given class or an instance of any class that inherits from that class. A `isMemberOfClass:` message, on the other hand, tells you if the receiver is an instance of the specified class. The `isKindOfClass:` method is generally more useful because from it you can know at once the complete range of messages you can send to an object. Consider the code snippet in Listing 4-2.

Listing 4-2 Using `isKindOfClass:`

```
if ([item isKindOfClass:[NSData class]]) {
    const unsigned char *bytes = [item bytes];
    unsigned int length = [item length];
    // ...
}
```

By learning that the object *item* inherits from the `NSData` class, this code knows it can send it the `NSDatabytes` and `length` messages. The difference between `isKindOfClass:` and `isMemberOfClass:` becomes apparent if you assume that *item* is an instance of `NSMutableData`. If you use `isMemberOfClass:` instead of `isKindOfClass:`, the code in the conditionalized block is never executed because *item* is not an instance of `NSData` but rather of `NSMutableData`, a subclass of `NSData`.

Method Implementation and Protocol Conformance

Two of the more powerful introspection methods of `NSObject` are `respondsToSelector:` and `conformsToProtocol:`. These methods tell you, respectively, whether an object implements a certain method and whether an object conforms to a specified formal protocol (that is, adopts the protocol, if necessary, and implements all the methods of the protocol).

You use these methods in a similar situation in your code. They enable you to discover whether some potentially anonymous object can respond appropriately to a particular message or set of messages *before* you send it any of those messages. By making this check before sending a message, you can avoid the risk of runtime exceptions resulting from unrecognized selectors. The AppKit framework implements informal protocols—the basis of delegation—by checking whether delegates implement a delegation method (using `respondsToSelector:`) prior to invoking that method.

Listing 4-3 illustrates how you might use the `respondsToSelector:` method in your code.

Listing 4-3 Using `respondsToSelector:`

```
- (void)doCommandBySelector:(SEL)aSelector {
    if ([self respondsToSelector:aSelector]) {
        [self performSelector:aSelector withObject:nil];
    } else {
        [_client doCommandBySelector:aSelector];
    }
}
```

Listing 4-4 illustrates how you might use the `conformsToProtocol:` method in your code.

Listing 4-4 Using `conformsToProtocol:`

```
// ...
if (![((id)testObject) conformsToProtocol:@protocol(NSMenuItem)]) {
    NSLog(@"Custom MenuItem, '%@', not loaded; it must conform to the
        'NSMenuItem' protocol.\n", [testObject class]);
    [testObject release];
    testObject = nil;
}
```

Object Comparison

Although they are not strictly introspection methods, the `hash` and `isEqual:` methods fulfill a similar role. They are indispensable runtime tools for identifying and comparing objects. But instead of querying the runtime for information about an object, they rely on class-specific comparison logic.

The `hash` and `isEqual:` methods, both declared by the `NSObject` protocol, are closely related. The `hash` method must be implemented to return an integer that can be used as a table address in a hash table structure. If two objects are equal (as determined by the `isEqual:` method), they must have the same hash value. If your object could be included in collections such as `NSSet` objects, you need to define `hash` and verify the invariant that if two objects are equal, they return the same hash value. The default `NSObject` implementation of `isEqual:` simply checks for pointer equality.

Using the `isEqual:` method is straightforward; it compares the receiver against the object supplied as a parameter. Object comparison frequently informs runtime decisions about what should be done with an object. As Listing 4-5 illustrates, you can use `isEqual:` to decide whether to perform an action, in this case to save user preferences that have been modified.

Listing 4-5 Using isEqual:

```
- (void)saveDefaults {
    NSDictionary *prefs = [self preferences];
    if (![origValues isEqual:prefs])
        [Preferences savePreferencesToDefaults:prefs];
}
```

If you are creating a subclass, you might need to override `isEqual:` to add further checks for points of equality. The subclass might define an extra attribute that has to be the same value in two instances for them to be considered equal. For example, say you create a subclass of `NSObject` called `MyWidget` that contains two instance variables, `name` and `data`. Both of these must be the same value for two instances of `MyWidget` to be considered equal. Listing 4-6 illustrates how you might implement `isEqual:` for the `MyWidget` class.

Listing 4-6 Overriding `isEqual:`

```
- (BOOL)isEqual:(id)other {
    if (other == self)
        return YES;
    if (!other || ![other isKindOfClass:[self class]])
        return NO;
    return [self isEqualToWidget:other];
}

- (BOOL)isEqualToWidget:(MyWidget *)aWidget {
    if (self == aWidget)
        return YES;
    if (![(id)[self name] isEqual:[aWidget name]])
        return NO;
    if (![[self data] isEqualToData:[aWidget data]])
        return NO;
    return YES;
```

```
}
```

This `isEqual:` method first checks for pointer equality, then class equality, and finally invokes an object comparator whose name indicates the class of object involved in the comparison. This type of comparator, which forces type checking of the object passed in, is a common convention in Cocoa; the `isEqualToString:` method of the `NSString` class and the `isEqualToTimeZone:` method of the `NSTimeZone` class are but two examples. The class-specific comparator—`isEqualToWidget:` in this case—performs the checks for name and data equality.

In all `isEqualTo`*Type* `:` methods of the Cocoa frameworks, `nil` is not a valid parameter and implementations of these methods may raise an exception upon receiving a `nil`. However, for backward compatibility, `isEqual:` methods of the Cocoa frameworks do accept `nil`, returning `NO`.

Object Allocation

When you allocate an object, part of what happens is what you might expect, given the term. Cocoa allocates enough memory for the object from a region of application virtual memory. To calculate how much memory to allocate, it takes the object's instance variables into account—including their types and order—as specified by the object's class.

To allocate an object, you send the message `alloc` or `allocWithZone:` to the object's class. In return, you get a "raw" (uninitialized) instance of the class. The `alloc` variant of the method uses the application's default zone. A zone is a page-aligned area of memory for holding related objects and data allocated by an application. See *Advanced Memory Management Programming Guide* for more information on zones.

An allocation message does other important things besides allocating memory:

- It sets the object's retain count to one.
- It initializes the object's `isa`instance variable to point to the object's class, a runtime object in its own right that is compiled from the class definition.
- It initializes all other instance variables to zero (or to the equivalent type for zero, such as `nil`, `NULL`, and `0.0`).

An object's `isa` instance variable is inherited from `NSObject`, so it is common to all Cocoa objects. After allocation sets `isa` to the object's class, the object is integrated into the runtime's view of the inheritance hierarchy and the current network of objects (class and instance) that constitute a program. Consequently an object can find whatever information it needs at runtime, such as another object's place in the inheritance hierarchy, the protocols that other objects conform to, and the location of the method implementations it can perform in response to messages.

In summary, allocation not only allocates memory for an object but initializes two small but very important attributes of any object: its `isa` instance variable and its retain count. It also sets all remaining instance variables to zero. But the resulting object is not yet usable. Initializing methods such as `init` must yet initialize objects with their particular characteristics and return a functional object.

Object Initialization

Initialization sets the instance variables of an object to reasonable and useful initial values. It can also allocate and prepare other global resources needed by the object, loading them if necessary from an external source such as a file. Every object that declares instance variables should implement an initializing method—unless the default set-everything-to-zero initialization is sufficient. If an object does not implement an initializer, Cocoa invokes the initializer of the nearest ancestor instead.

The Form of Initializers

`NSObject` declares the `init` prototype for initializers; it is an instance method typed to return an object of type `id`. Overriding `init` is fine for subclasses that require no additional data to initialize their objects. But often initialization depends on external data to set an object to a reasonable initial state. For example, say you have an `Account` class; to initialize an `Account` object appropriately requires a unique account number, and this must be supplied to the initializer. Thus initializers can take one or more parameters; the only requirement is that the initializing method begins with the letters "init". (The stylistic convention `init...` is sometimes used to refer to initializers.)

> **Note:** Instead of implementing an initializer with parameters, a subclass can implement only a simple `init` method and then use "set" accessor methods immediately after initialization to set the object to a useful initial state. (Accessor methods enforce encapsulation of object data by setting and getting the values of instance variables.) Or, if the subclass uses properties and the related access syntax, it may assign values to the properties immediately after initialization.

Cocoa has plenty of examples of initializers with parameters. Here are a few (with the defining class in parentheses):

- `- (id)initWithArray:(NSArray *)array;` (from `NSSet`)
- `- (id)initWithTimeInterval:(NSTimeInterval)secsToBeAdded sinceDate:(NSDate *)anotherDate;` (from `NSDate`)
- `- (id)initWithContentRect:(NSRect)contentRect styleMask:(unsigned int)aStyle backing:(NSBackingStoreType)bufferingType defer:(BOOL)flag;` (from `NSWindow`)
- `- (id)initWithFrame:(NSRect)frameRect;` (from `NSControl` and `NSView`)

These initializers are instance methods that begin with "init" and return an object of the dynamic type `id`. Other than that, they follow the Cocoa conventions for multiparameter methods, often using `With`*Type*`:` or `From`*Source*`:` before the first and most important parameter.

Issues with Initializers

Although `init...` methods are required by their method signature to return an object, that object is not necessarily the one that was most recently allocated—the receiver of the `init...` message. In other words, the object you get back from an initializer might not be the one you thought was being initialized.

Two conditions prompt the return of something other than the just-allocated object. The first involves two related situations: when there must be a singleton instance or when the defining attribute of an object must be unique. Some Cocoa classes—`NSWorkspace`, for instance—allow only one instance in a program; a class in such a case must ensure (in an initializer or, more likely, in a class factory method) that only one instance is created, returning this instance if there is any further request for a new one.

A similar situation arises when an object is required to have an attribute that makes it unique. Recall the hypothetical `Account` class mentioned earlier. An account of any sort must have a unique identifier. If the initializer for this class—say, `initWithAccountID:`—is passed an identifier that has already been associated with an object, it must do two things:

- Release the newly allocated object (in memory-managed code)
- Return the `Account` object previously initialized with this unique identifier

By doing this, the initializer ensures the uniqueness of the identifier while providing what was asked for: an `Account` instance with the requested identifier.

Sometimes an `init...` method cannot perform the initialization requested. For example, an `initFromFile:` method expects to initialize an object from the contents of a file, the path to which is passed as a parameter. But if no file exists at that location, the object cannot be initialized. A similar problem happens if an `initWithArray:` initializer is passed an `NSDictionary` object instead of an `NSArray` object. When an `init...` method cannot initialize an object, it should:

- Release the newly allocated object (in memory-managed code)
- Return `nil`

Returning `nil` from an initializer indicates that the requested object cannot be created. When you create an object, you should generally check whether the returned value is `nil` before proceeding:

```
id anObject = [[MyClass alloc] init];
if (anObject) {
    [anObject doSomething];
    // more messages...
} else {
    // handle error
}
```

Because an `init...` method might return `nil` or an object other than the one explicitly allocated, it is dangerous to use the instance returned by `alloc` or `allocWithZone:` instead of the one returned by the initializer. Consider the following code:

```
id myObject = [MyClass alloc];
[myObject init];
[myObject doSomething];
```

The `init` method in this example could have returned `nil` or could have substituted a different object. Because you can send a message to `nil` without raising an exception, nothing would happen in the former case except (perhaps) a debugging headache. But you should always rely on the initialized instance instead of the "raw" just-allocated one. Therefore, you should nest the allocation message inside the initialization message and test the object returned from the initializer before proceeding.

```
id myObject = [[MyClass alloc] init];
if ( myObject ) {
    [myObject doSomething];
} else {
    // error recovery...
}
```

Once an object is initialized, you should not initialize it again. If you attempt a reinitialization, the framework class of the instantiated object often raises an exception. For example, the second initialization in this example would result in `NSInvalidArgumentException` being raised.

```
NSString *aStr = [[NSString alloc] initWithString:@"Foo"];
aStr = [aStr initWithString:@"Bar"];
```

Implementing an Initializer

There are several critical rules to follow when implementing an `init...` method that serves as a class's sole initializer or, if there are multiple initializers, its *designated initializer* (described in "Multiple Initializers and the Designated Initializer" (page 40)):

- Always invoke the superclass (`super`) initializer *first*.
- Check the object returned by the superclass. If it is `nil`, then initialization cannot proceed; return `nil` to the receiver.
- When initializing instance variables that are references to objects, retain or copy the object as necessary (in memory-managed code).
- After setting instance variables to valid initial values, return `self` unless:
  - It was necessary to return a substituted object, in which case release the freshly allocated object first (in memory-managed code).
  - A problem prevented initialization from succeeding, in which case return `nil`.

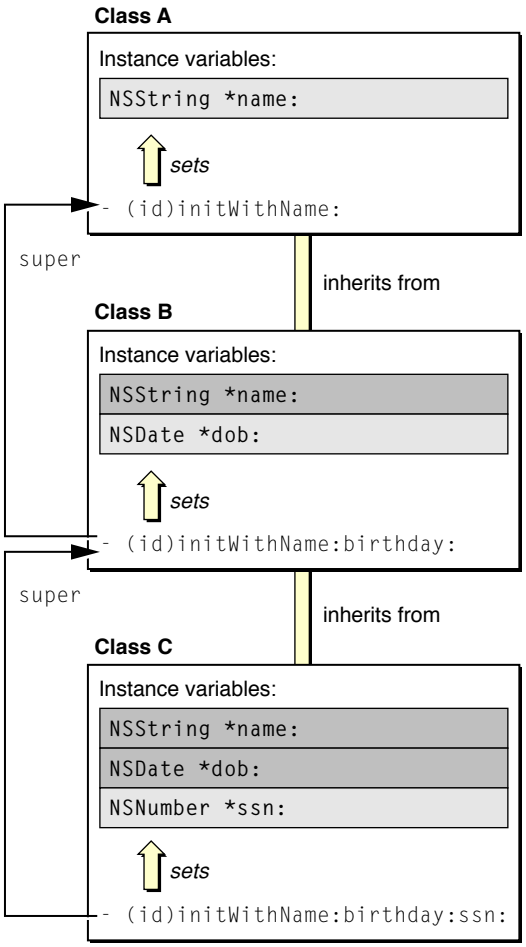
```
- (id)initWithAccountID:(NSString *)identifier {
    if ( self = [super init] ) {
        Account *ac = [accountDictionary objectForKey:identifier];
        if (ac) { // object with that ID already exists
            [self release];
            return [ac retain];
        }
        if (identifier) {
            accountID = [identifier copy]; // accountID is instance variable
            [accountDictionary setObject:self forKey:identifier];
            return self;
        } else {
            [self release];
            return nil;
        }
    } else
        return nil;
}
```

> **Note:** Although, for the sake of simplicity, this example returns `nil` if the parameter is `nil`, the better Cocoa practice is to raise an exception.

It isn't necessary to initialize all instance variables of an object explicitly, just those that are necessary to make the object functional. The default set-to-zero initialization performed on an instance variable during allocation is often sufficient. Make sure that you retain or copy instance variables, as required for memory management.

The requirement to invoke the superclass's initializer as the first action is important. Recall that an object encapsulates not only the instance variables defined by its class but the instance variables defined by all of its ancestor classes. By invoking the initializer of `super` first, you help to ensure that the instance variables defined by classes up the inheritance chain are initialized first. The immediate superclass, in its initializer, invokes the initializer of its superclass, which invokes the main `init...` method of its superclass, and so on (see Figure 6-1). The proper order of initialization is critical because the later initializations of subclasses may depend on superclass-defined instance variables being initialized to reasonable values.

Figure 6-1 Initialization up the inheritance chain



Inherited initializers are a concern when you create a subclass. Sometimes a superclass `init...` method sufficiently initializes instances of your class. But because it is more likely it won't, you should override the superclass's initializer. If you don't, the superclass's implementation is invoked, and because the superclass knows nothing about your class, your instances may not be correctly initialized.

Multiple Initializers and the Designated Initializer

A class can define more than one initializer. Sometimes multiple initializers let clients of the class provide the input for the same initialization in different forms. The `NSSet` class, for example, offers clients several initializers that accept the same data in different forms; one takes an `NSArray` object, another a counted list of elements, and another a `nil`-terminated list of elements:

```
- (id)initWithArray:(NSArray *)array;
- (id)initWithObjects:(id *)objects count:(unsigned)count;
- (id)initWithObjects:(id)firstObj, ...;
```

Some subclasses provide convenience initializers that supply default values to an initializer that takes the full complement of initialization parameters. This initializer is usually the designated initializer, the most important initializer of a class. For example, assume there is a `Task` class and it declares a designated initializer with this signature:

```
- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate;
```

The `Task` class might include secondary, or convenience, initializers that simply invoke the designated initializer, passing it default values for those parameters the secondary initializer doesn't explicitly request. This example shows a designated initializer and a secondary initializer.

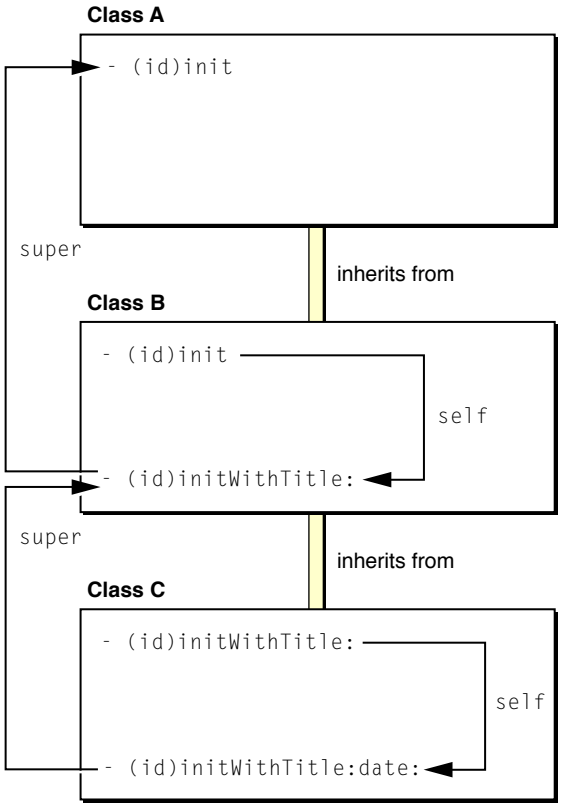
```
- (id)initWithTitle:(NSString *)aTitle {
    return [self initWithTitle:aTitle date:[NSDate date]];
}

- (id)init {
    return [self initWithTitle:@"Task"];
}
```

The designated initializer plays an important role for a class. It ensures that inherited instance variables are initialized by invoking the designated initializer of the superclass. It is typically the `init...` method that has the most parameters and that does most of the initialization work, and it is the initializer that secondary initializers of the class invoke with messages to `self`.

When you define a subclass, you must be able to identify the designated initializer of the superclass and invoke it in your subclass's designated initializer through a message to `super`. You must also make sure that inherited initializers are covered in some way. And you may provide as many convenience initializers as you deem necessary. When designing the initializers of your class, keep in mind that designated initializers are chained to each other through messages to `super`; whereas other initializers are chained to the designated initializer of their class through messages to `self`.

An example will make this clearer. Let's say there are three classes, A, B, and C; class B inherits from class A, and class C inherits from class B. Each subclass adds an attribute as an instance variable and implements an `init...` method—the designated initializer—to initialize this instance variable. They also define secondary initializers and ensure that inherited initializers are overridden, if necessary. Figure 6-2 illustrates the initializers of all three classes and their relationships.

Figure 6-2 Interactions of secondary and designated initializers



The designated initializer for each class is the initializer with the most coverage; it is the method that initializes the attribute added by the subclass. The designated initializer is also the `init...` method that invokes the designated initializer of the superclass in a message to `super`. In this example, the designated initializer of class C, `initWithTitle:date:`, invokes the designated initializer of its superclass, `initWithTitle:`, which in turn invokes the `init` method of class A. When creating a subclass, it's always important to know the designated initializer of the superclass.

Although designated initializers are thus connected up the inheritance chain through messages to `super`, secondary initializers are connected to their class's designated initializer through messages to `self`. Secondary initializers (as in this example) are frequently overridden versions of inherited initializers. Class C overrides `initWithTitle:` to invoke its designated initializer, passing it a default date. This designated initializer, in turn, invokes the designated initializer of class B, which is the overridden method, `initWithTitle:`. If you sent an `initWithTitle:` message to objects of class B and class C, you'd be invoking different method implementations. On the other hand, if class C did *not* override `initWithTitle:` and you sent the message to an instance of class C, the class B implementation would be invoked. Consequently, the C instance would be incompletely initialized (since it would lack a date). When creating a subclass, it's important to make sure that all inherited initializers are adequately covered.

Sometimes the designated initializer of a superclass may be sufficient for the subclass, and so there is no need for the subclass to implement its own designated initializer. Other times, a class's designated initializer may be an overridden version of its superclass's designated initializer. This is frequently the case when the subclass needs to supplement the work performed by the superclass's designated initializer, even though the subclass does not add any instance variables of its own (or the instance variables it does add don't require explicit initialization).

Model-View-Controller

The Model-View-Controller design pattern (MVC) is quite old. Variations of it have been around at least since the early days of Smalltalk. It is a high-level pattern in that it concerns itself with the global architecture of an application and classifies objects according to the general roles they play in an application. It is also a compound pattern in that it comprises several, more elemental patterns.

Object-oriented programs benefit in several ways by adapting the MVC design pattern for their designs. Many objects in these programs tend to be more reusable and their interfaces tend to be better defined. The programs overall are more adaptable to changing requirements—in other words, they are more easily extensible than programs that are not based on MVC. Moreover, many technologies and architectures in Cocoa—such as bindings, the document architecture, and scriptability—are based on MVC and require that your custom objects play one of the roles defined by MVC.

Roles and Relationships of MVC Objects

The MVC design pattern considers there to be three types of objects: model objects, view objects, and controller objects. The MVC pattern defines the roles that these types of objects play in the application and their lines of communication. When designing an application, a major step is choosing—or creating custom classes for—objects that fall into one of these three groups. Each of the three types of objects is separated from the others by abstract boundaries and communicates with objects of the other types across those boundaries.

Model Objects Encapsulate Data and Basic Behaviors

Model objects represent special knowledge and expertise. They hold an application's data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects. Any data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects once the data is loaded into the application. Because they represent knowledge and expertise related to a specific problem domain, they tend to be reusable.

Ideally, a model object has no explicit connection to the user interface used to present and edit it. For example, if you have a model object that represents a person (say you are writing an address book), you might want to store a birthdate. That's a good thing to store in your Person model object. However, storing a date format string or other information on how that date is to be presented is probably better off somewhere else.

In practice, this separation is not always the best thing, and there is some room for flexibility here, but in general a model object should not be concerned with interface and presentation issues. One example where a bit of an exception is reasonable is a drawing application that has model objects that represent the graphics displayed. It makes sense for the graphic objects to know how to draw themselves because the main reason for their existence is to define a visual thing. But even in this case, the graphic objects should not rely on living in a particular view or any view at all, and they should not be in charge of knowing when to draw themselves. They should be asked to draw themselves by the view object that wants to present them.

View Objects Present Information to the User

A view object knows how to display, and might allow users to edit, the data from the application's model. The view should not be responsible for storing the data it is displaying. (This does not mean the view never actually stores data it's displaying, of course. A view can cache data or do similar tricks for performance reasons). A view object can be in charge of displaying just one part of a model object, or a whole model object, or even many different model objects. Views come in many different varieties.

View objects tend to be reusable and configurable, and they provide consistency between applications. In Cocoa, the AppKit framework defines a large number of view objects and provides many of them in the Interface Builder library. By reusing the AppKit's view objects, such as `NSButton` objects, you guarantee that buttons in your application behave just like buttons in any other Cocoa application, assuring a high level of consistency in appearance and behavior across applications.

A view should ensure it is displaying the model correctly. Consequently, it usually needs to know about changes to the model. Because model objects should not be tied to specific view objects, they need a generic way of indicating that they have changed.

Controller Objects Tie the Model to the View

A controller object acts as the intermediary between the application's view objects and its model objects. Controllers are often in charge of making sure the views have access to the model objects they need to display and act as the conduit through which views learn about changes to the model. Controller objects can also perform set-up and coordinating tasks for an application and manage the life cycles of other objects.

In a typical Cocoa MVC design, when users enter a value or indicate a choice through a view object, that value or choice is communicated to a controller object. The controller object might interpret the user input in some application-specific way and then either may tell a model object what to do with this input—for example, "add a new value" or "delete the current record"—or it may have the model object reflect a changed value in one of its properties. Based on this same user input, some controller objects might also tell a view object to change an aspect of its appearance or behavior, such as telling a button to disable itself. Conversely, when a model object changes—say, a new data source is accessed—the model object usually communicates that change to a controller object, which then requests one or more view objects to update themselves accordingly.

Controller objects can be either reusable or nonreusable, depending on their general type. "Types of Cocoa Controller Objects" (page 45) describes the different types of controller objects in Cocoa.

Combining Roles

One can merge the MVC roles played by an object, making an object, for example, fulfill both the controller and view roles—in which case, it would be called a *view controller*. In the same way, you can also have model-controller objects. For some applications, combining roles like this is an acceptable design.

A *model controller* is a controller that concerns itself mostly with the model layer. It "owns" the model; its primary responsibilities are to manage the model and communicate with view objects. Action methods that apply to the model as a whole are typically implemented in a model controller. The document architecture provides a number of these methods for you; for example, an `NSDocument` object (which is a central part of the document architecture) automatically handles action methods related to saving files.

A view controller is a controller that concerns itself mostly with the view layer. It "owns" the interface (the views); its primary responsibilities are to manage the interface and communicate with the model. Action methods concerned with data displayed in a view are typically implemented in a view controller. An `NSWindowController` object (also part of the document architecture) is an example of a view controller.

"Design Guidelines for MVC Applications" (page 50) offers some design advice concerning objects with merged MVC roles.

> **Further Reading:** *Document-Based Applications Overview* discusses the distinction between a model controller and a view controller from another perspective.

Types of Cocoa Controller Objects

"Controller Objects Tie the Model to the View" (page 44) sketches the abstract outline of a controller object, but in practice the picture is far more complex. In Cocoa there are two general kinds of controller objects: mediating controllers and coordinating controllers. Each kind of controller object is associated with a different set of classes and each provides a different range of behaviors.

A *mediating controller* is typically an object that inherits from the `NSController`class. Mediating controller objects are used in the Cocoa bindings technology. They facilitate—or mediate—the flow of data between view objects and model objects.

> **iOS Note:** AppKit implements the `NSController` class and its subclasses. These classes and the bindings technology are not available in iOS.

Mediating controllers are typically ready-made objects that you drag from the Interface Builder library. You can configure these objects to establish the bindings between properties of view objects and properties of the controller object, and then between those controller properties and specific properties of a model object. As a result, when users change a value displayed in a view object, the new value is automatically communicated to a model object for storage—via the mediating controller; and when a property of a model changes its value, that change is communicated to a view for display. The abstract `NSController` class and its concrete subclasses—`NSObjectController`, `NSArrayController`, `NSUserDefaultsController`, and `NSTreeController`—provide supporting features such as the ability to commit and discard changes and the management of selections and placeholder values.

A *coordinating controller* is typically an `NSWindowController` or `NSDocumentController`object (available only in AppKit), or an instance of a custom subclass of `NSObject`. Its role in an application is to oversee—or coordinate—the functioning of the entire application or of part of the application, such as the objects unarchived from a nib file. A coordinating controller provides services such as:

- Responding to delegation messages and observing notifications
- Responding to action messages
- Managing the life cycle of owned objects (for example, releasing them at the proper time)
- Establishing connections between objects and performing other set-up tasks

`NSWindowController` and `NSDocumentController` are classes that are part of the Cocoa architecture for document-based applications. Instances of these classes provide default implementations for several of the services listed above, and you can create subclasses of them to implement more application-specific behavior. You can even use `NSWindowController` objects to manage windows in an application that is not based on the document architecture.

A coordinating controller frequently owns the objects archived in a nib file. As File's Owner, the coordinating controller is external to the objects in the nib file and manages those objects. These owned objects include mediating controllers as well as window objects and view objects. See "MVC as a Compound Design Pattern" (page 47) for more on coordinating controllers as File's Owner.

Instances of custom `NSObject` subclasses can be entirely suitable as coordinating controllers. These kinds of controller objects combine both mediating and coordinating functions. For their mediating behavior, they make use of mechanisms such as target-action, outlets, delegation, and notifications to facilitate the movement of data between view objects and model objects. They tend to contain a lot of glue code and, because that code is exclusively application-specific, they are the least reusable kind of object in an application.

Further Reading: For more on the Cocoa bindings technology, see *Cocoa Bindings Programming Topics*.

MVC as a Compound Design Pattern

Model-View-Controller is a design pattern that is composed of several more basic design patterns. These basic patterns work together to define the functional separation and paths of communication that are characteristic of an MVC application. However, the traditional notion of MVC assigns a set of basic patterns different from those that Cocoa assigns. The difference primarily lies in the roles given to the controller and view objects of an application.

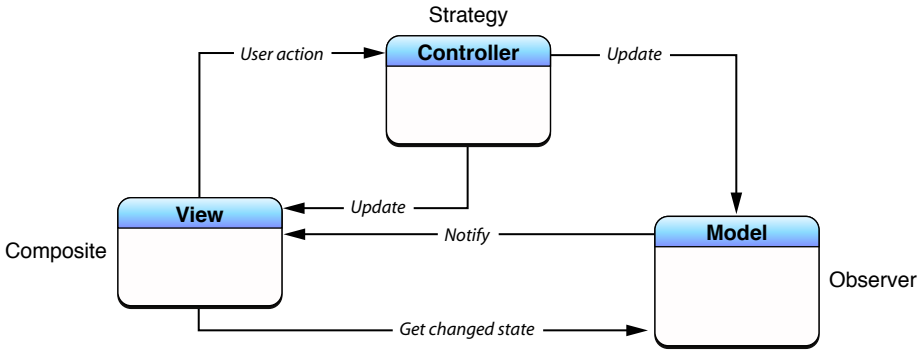
In the original (Smalltalk) conception, MVC is made up of the Composite, Strategy, and Observer patterns.

- Composite—The view objects in an application are actually a composite of nested views that work together in a coordinated fashion (that is, the view hierarchy). These display components range from a window to compound views, such as a table view, to individual views, such as buttons. User input and display can take place at any level of the composite structure.
- Strategy—A controller object implements the strategy for one or more view objects. The view object confines itself to maintaining its visual aspects, and it delegates to the controller all decisions about the application-specific meaning of the interface behavior.
- Observer—A model object keeps interested objects in an application—usually view objects—advised of changes in its state.

The traditional way the Composite, Strategy, and Observer patterns work together is depicted by Figure 7-1: The user manipulates a view at some level of the composite structure and, as a result, an event is generated. A controller object receives the event and interprets it in an application-specific way—that is, it applies a strategy. This strategy can be to request (via message) a model object to change its state or to request a view

object (at some level of the composite structure) to change its behavior or appearance. The model object, in turn, notifies all objects who have registered as observers when its state changes; if the observer is a view object, it may update its appearance accordingly.

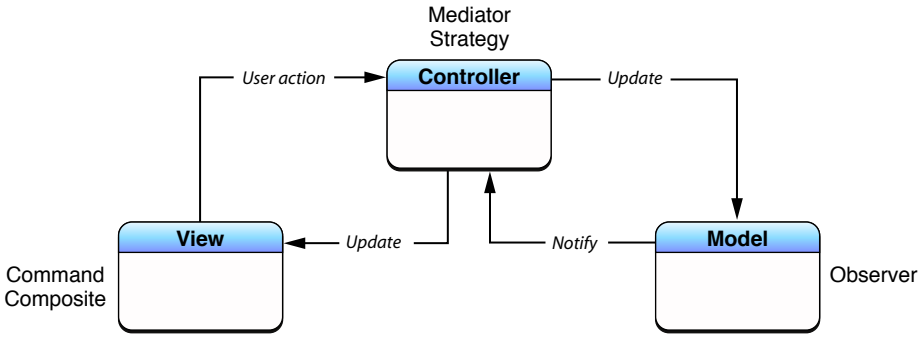
Figure 7-1 Traditional version of MVC as a compound pattern



The Cocoa version of MVC as a compound pattern has some similarities to the traditional version, and in fact it is quite possible to construct a working application based on the diagram in Figure 7-1. By using the bindings technology, you can easily create a Cocoa MVC application whose views directly observe model objects to receive notifications of state changes. However, there is a theoretical problem with this design. View objects and model objects should be the most reusable objects in an application. View objects represent the "look and feel" of an operating system and the applications that system supports; consistency in appearance and behavior is essential, and that requires highly reusable objects. Model objects by definition encapsulate the data associated with a problem domain and perform operations on that data. Design-wise, it's best to keep model and view objects separate from each other, because that enhances their reusability.

In most Cocoa applications, notifications of state changes in model objects are communicated to view objects *through* controller objects. Figure 7-2 shows this different configuration, which appears much cleaner despite the involvement of two more basic design patterns.

Figure 7-2 Cocoa version of MVC as a compound design pattern



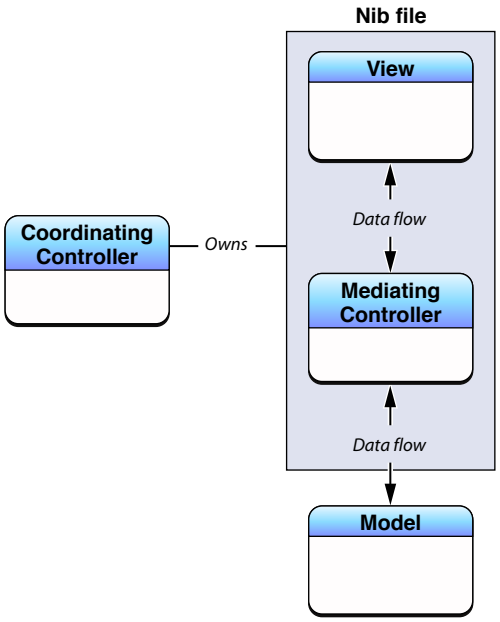
The controller object in this compound design pattern incorporates the Mediator pattern as well as the Strategy pattern; it mediates the flow of data between model and view objects in both directions. Changes in model state are communicated to view objects through the controller objects of an application. In addition, view objects incorporate the Command pattern through their implementation of the target-action mechanism.

> **Note:** The target-action mechanism, which enables view objects to communicate user input and choices, can be implemented in both coordinating and mediating controller objects. However, the design of the mechanism differs in each controller type. For coordinating controllers, you connect the view object to its target (the controller object) in Interface Builder and specify an action selector that must conform to a certain signature. Coordinating controllers, by virtue of being delegates of windows and the global application object, can also be in the responder chain. The bindings mechanism used by mediating controllers also connects view objects to targets and allows action signatures with a variable number of parameters of arbitrary types. Mediating controllers, however, aren't in the responder chain.

There are practical reasons as well as theoretical ones for the revised compound design pattern depicted in Figure 7-2, especially when it comes to the Mediator design pattern. Mediating controllers derive from concrete subclasses of `NSController`, and these classes, besides implementing the Mediator pattern, offer many features that applications should take advantage of, such as the management of selections and placeholder values. And if you opt not to use the bindings technology, your view object could use a mechanism such as the Cocoa notification center to receive notifications from a model object. But this would require you to create a custom view subclass to add the knowledge of the notifications posted by the model object.

In a well-designed Cocoa MVC application, coordinating controller objects often own mediating controllers, which are archived in nib files. Figure 7-3 shows the relationships between the two types of controller objects.

Figure 7-3 Coordinating controller as the owner of a nib file



Design Guidelines for MVC Applications

The following guidelines apply to Model-View-Controller considerations in the design of applications:

- Although you can use an instance of a custom subclass of `NSObject` as a mediating controller, there's no reason to go through all the work required to make it one. Use instead one of the ready-made `NSController` objects designed for the Cocoa bindings technology; that is, use an instance of `NSObjectController`, `NSArrayController`, `NSUserDefaultsController`, or `NSTreeController`—or a custom subclass of one of these concrete `NSController` subclasses.

  However, if the application is very simple and you feel more comfortable writing the glue code needed to implement mediating behavior using outlets and target-action, feel free to use an instance of a custom `NSObject` subclass as a mediating controller. In a custom `NSObject` subclass, you can also implement a mediating controller in the `NSController` sense, using key-value coding, key-value observing, and the editor protocols.

- Although you can combine MVC roles in an object, the best overall strategy is to keep the separation between roles. This separation enhances the reusability of objects and the extensibility of the program they're used in. If you are going to merge MVC roles in a class, pick a predominant role for that class and then (for maintenance purposes) use categories in the same implementation file to extend the class to play other roles.
- A goal of a well-designed MVC application should be to use as many objects as possible that are (theoretically, at least) reusable. In particular, view objects and model objects should be highly reusable. (The ready-made mediating controller objects, of course, are reusable.) Application-specific behavior is frequently concentrated as much as possible in controller objects.
- Although it is possible to have views directly observe models to detect changes in state, it is best not to do so. A view object should always go through a mediating controller object to learn about changes in an model object. The reason is two-fold:
    - If you use the bindings mechanism to have view objects directly observe the properties of model objects, you bypass all the advantages that `NSController` and its subclasses give your application: selection and placeholder management as well as the ability to commit and discard changes.
    - If you don't use the bindings mechanism, you have to subclass an existing view class to add the ability to observe change notifications posted by a model object.
- Strive to limit code dependency in the classes of your application. The greater the dependency a class has on another class, the less reusable it is. Specific recommendations vary by the MVC roles of the two classes involved:
    - A view class shouldn't depend on a model class (although this may be unavoidable with some custom views).
    - A view class shouldn't have to depend on a mediating controller class.
    - A model class shouldn't depend on anything other than other model classes.
    - A mediating controller class shouldn’t depend on a model class (although, like views, this may be necessary if it's a custom controller class).
    - A mediating controller class shouldn't depend on view classes or on coordinating controller classes.
    - A coordinating controller class depends on classes of all MVC role types.
- If Cocoa offers an architecture that solves a programming problem, and this architecture assigns MVC roles to objects of specific types, use that architecture. It will be much easier to put your project together if you do. The document architecture, for example, includes an Xcode project template that configures an `NSDocument` object (per-nib model controller) as File's Owner.

Model-View-Controller in Cocoa (OS X)

The Model-View-Controller design pattern is fundamental to many Cocoa mechanisms and technologies. As a consequence, the importance of using MVC in object-oriented design goes beyond attaining greater reusability and extensibility for your own applications. If your application is to incorporate a Cocoa technology that is MVC-based, your application will work best if its design also follows the MVC pattern. It should be relatively painless to use these technologies if your application has a good MVC separation, but it will take more effort to use such a technology if you don't have a good separation.

Cocoa in OS X includes the following architectures, mechanisms, and technologies that are based on Model-View-Controller:

- **Document architecture**. In this architecture, a document-based application consists of a controller object for the entire application (`NSDocumentController`), a controller object for each document window (`NSWindowController`), and an object that combines controller and model roles for each document (`NSDocument`).
- **Bindings**. MVC is central to the bindings technology of Cocoa. The concrete subclasses of the abstract `NSController` provide ready-made controller objects that you can configure to establish bindings between view objects and properly designed model objects.
- **Application scriptability**. When designing an application to make it scriptable, it is essential not only that it follow the MVC design pattern but that your application's model objects are properly designed. Scripting commands that access application state and request application behavior should usually be sent to model objects or controller objects.
- **Core Data**. The Core Data framework manages graphs of model objects and ensures the persistence of those objects by saving them to (and retrieving them from) a persistent store. Core Data is tightly integrated with the Cocoa bindings technology. The MVC and object modeling design patterns are essential determinants of the Core Data architecture.
- **Undo**. In the undo architecture, model objects once again play a central role. The primitive methods of model objects (which are usually its accessor methods) are often where you implement undo and redo operations. The view and controller objects of an action may also be involved in these operations; for example, you might have such objects give specific titles to the undo and redo menu items, or you might have them undo selections in a text view.

Object Modeling

This section defines terms and presents examples of object modeling and key-value coding that are specific to Cocoa bindings and the Core Data framework. Understanding terms such as key paths is fundamental to using these technologies effectively. This section is recommended reading if you are new to object-oriented design or key-value coding.

When using the Core Data framework, you need a way to describe your model objects that does not depend on views and controllers. In a good reusable design, views and controllers need a way to access model properties without imposing dependencies between them. The Core Data framework solves this problem by borrowing concepts and terms from database technology—specifically, the entity-relationship model.

Entity-relationship modeling is a way of representing objects typically used to describe a data source's data structures in a way that allows those data structures to be mapped to objects in an object-oriented system. Note that entity-relationship modeling isn't unique to Cocoa; it's a popular discipline with a set of rules and terms that are documented in database literature. It is a representation that facilitates storage and retrieval of objects in a data source. A data source can be a database, a file, a web service, or any other persistent store. Because it is not dependent on any type of data source it can also be used to represent any kind of object and its relationship to other objects.

In the entity-relationship model, the objects that hold data are called *entities*, the components of an entity are called *attributes*, and the references to other data-bearing objects are called *relationships*. Together, attributes and relationships are known as *properties*. With these three simple components (entities, attributes, and relationships), you can model systems of any complexity.

Cocoa uses a modified version of the traditional rules of entity-relationship modeling referred to in this document as *object modeling*. Object modeling is particularly useful in representing model objects in the Model-View-Controller (MVC) design pattern. This is not surprising because even in a simple Cocoa application, models are typically persistent—that is, they are stored in a data container such as a file.

Entities

Entities are model objects. In the MVC design pattern, model objects are the objects in your application that encapsulate specified data and provide methods that operate on that data. They are usually persistent but more importantly, model objects are not dependent on how the data is displayed to the user.

For example, a structured collection of model objects (an object model) can be used to represent a company's customer base, a library of books, or a network of computers. A library book has attributes—such as the book title, ISBN number, and copyright date—and relationships to other objects—such as the author and library member. In theory, if the parts of a system can be identified, the system can be expressed as an object model.

Figure 8-1 shows an example object model used in an employee management application. In this model, Department models a department and Employee models an employee.

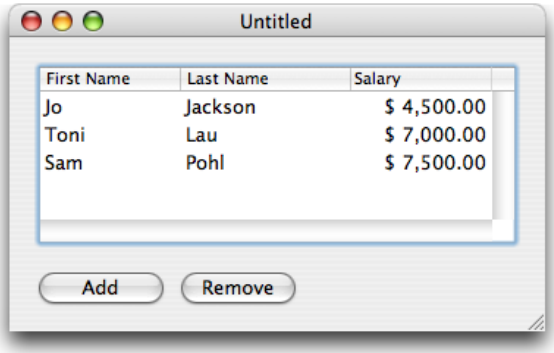
Figure 8-1 Employee management application object diagram

Attributes

Attributes represent structures that contain data. An attribute of an object may be a simple value, such as a scalar (for example, an `integer`, `float`, or `double` value), but can also be a C structure (for example an array of `char` values or an `NSPoint` structure) or an instance of a primitive class (such as, `NSNumber`, `NSData`, or `NSColor` in Cocoa). Immutable objects such as `NSColor` are usually considered attributes too. (Note that Core Data natively supports only a specific set of attribute types, as described in *NSAttributeDescription Class Reference*. You can, however, use additional attribute types, as described in "Non-Standard Persistent Attributes" in *Core Data Programming Guide*.)

In Cocoa, an attribute typically corresponds to a model's instance variable or accessor method. For example, Employee has `firstName`, `lastName`, and `salary` instance variables. In an employee management application, you might implement a table view to display a collection of Employee objects and some of their attributes, as shown in Figure 8-2. Each row in the table corresponds to an instance of Employee, and each column corresponds to an attribute of Employee.

Figure 8-2 Employees table view



Relationships

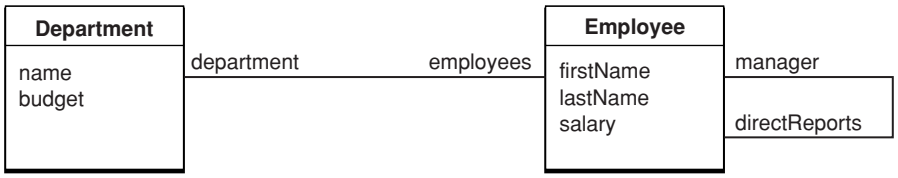
Not all properties of a model are attributes—some properties are relationships to other objects. Your application is typically modeled by multiple classes. At runtime, your object model is a collection of related objects that make up an object graph. These are typically the persistent objects that your users create and save to some data container or file before terminating the application (as in a document-based application). The relationships between these model objects can be traversed at runtime to access the properties of the related objects.

For example, in the employee management application, there are relationships between an employee and the department in which the employee works, and between an employee and the employee's manager. Because a manager is also an employee, the employee–manager relationship is an example of a reflexive relationship—a relationship from an entity to itself.

Relationships are inherently bidirectional, so conceptually at least there are also relationships between a department and the employees that work in the department, and an employee and the employee's direct reports. Figure 8-3 (page 56) illustrates the relationships between a Department and an Employee entity, and the Employee reflexive relationship. In this example, the Department entity's "employees" relationship is the inverse of the Employee entity's "department" relationship. It is possible, however, for relationships to be navigable in only one direction—for there to be no inverse relationship. If, for example, you are never interested in finding out from a department object what employees are associated with it, then you do not have to model

that relationship. (Note that although this is true in the general case, Core Data may impose additional constraints over general Cocoa object modeling—not modeling the inverse should be considered an extremely advanced option.)

Figure 8-3 Relationships in the employee management application



Relationship Cardinality and Ownership

Every relationship has a *cardinality*; the cardinality tells you how many destination objects can (potentially) resolve the relationship. If the destination object is a single object, then the relationship is called a *to-one relationship*. If there may be more than one object in the destination, then the relationship is called a *to-many relationship*.

Relationships can be mandatory or optional. A mandatory relationship is one where the destination is required—for example, every employee must be associated with a department. An optional relationship is, as the name suggests, optional—for example, not every employee has direct reports. So the `directReports` relationship depicted in Figure 8-4 (page 56) is optional.

It is also possible to specify a range for the cardinality. An optional to-one relationship has a range 0-1. An employee may have any number of direct reports, or a range that specifies a minimum and a maximum, for example, 0-15, which also illustrates an optional to-many relationship.

Figure 8-4 illustrates the cardinalities in the employee management application. The relationship between an Employee object and a Department object is a mandatory to-one relationship—an employee must belong to one, and only one, department. The relationship between a Department and its Employee objects is an optional to-many relationship (represented by a "*"). The relationship between an employee and a manager is an optional to-one relationship (denoted by the range 0-1)—top-ranking employees do not have managers.

Figure 8-4 Relationship cardinality

Note also that destination objects of relationships are sometimes owned and sometimes shared.

Accessing Properties

In order for models, views, and controllers to be independent of each other, you need to be able to access properties in a way that is independent of a model's implementation. This is accomplished by using key-value pairs.

Keys

You specify properties of a model using a simple key, often a string. The corresponding view or controller uses the key to look up the corresponding attribute value. This design enforces the notion that the attribute itself doesn't necessarily contain the data—the value can be indirectly obtained or derived.

Key-value coding is used to perform this lookup; it is a mechanism for accessing an object's properties indirectly and, in certain contexts, automatically. Key-value coding works by using the names of the object's properties—typically its instance variables or accessor methods—as keys to access the values of those properties.

For example, you might obtain the name of a Department object using a `name` key. If the Department object either has an instance variable or a method called `name` then a value for the key can be returned (if it doesn't have either, an error is returned). Similarly, you might obtain Employee attributes using the `firstName`, `lastName`, and `salary` keys.

Values

All values for a particular attribute of a given entity are of the same data type. The data type of an attribute is specified in the declaration of its corresponding instance variable or in the return value of its accessor method. For example, the data type of the Department object `name` attribute may be an `NSString` object in Objective-C.

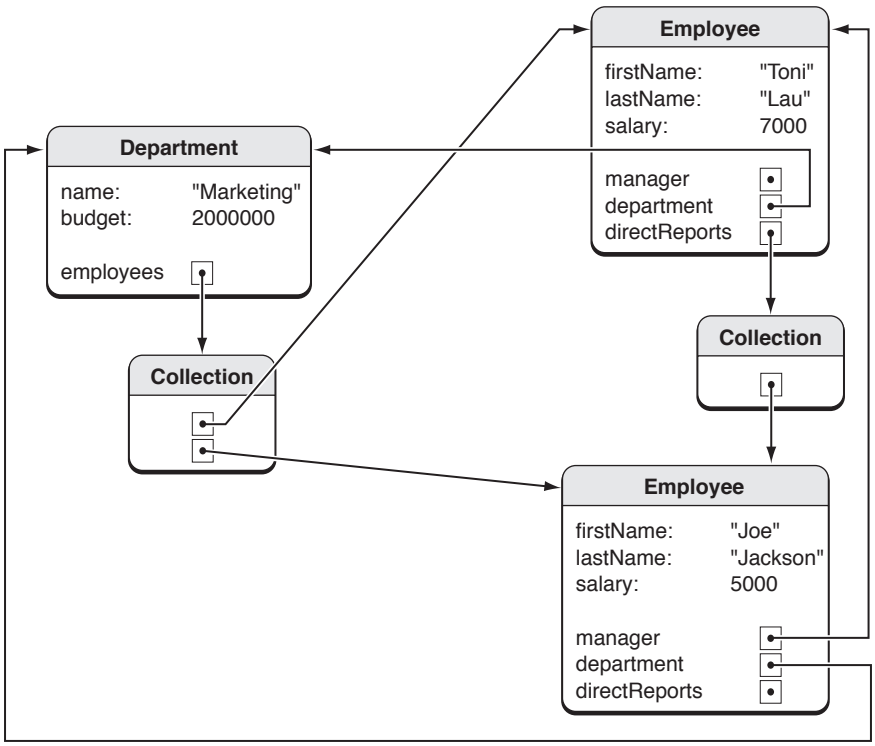
Note that key-value coding returns only object values. If the return type or the data type for the specific accessor method or instance variable used to supply the value for a specified key is not an object, then an `NSNumber` or `NSValue` object is created for that value and returned in its place. If the `name` attribute of Department is of type `NSString`, then, using key-value coding, the value returned for the `name` key of a Department object is an `NSString` object. If the `budget` attribute of Department is of type `float`, then, using key-value coding, the value returned for the `budget` key of a Department object is an `NSNumber` object.

Similarly, when you set a value using key-value coding, if the data type required by the appropriate accessor or instance variable for the specified key is not an object, then the value is extracted from the passed object using the appropriate *–type* `Value` method.

The value of a to-one relationship is simply the destination object of that relationship. For example, the value of the `department` property of an Employee object is a Department object. The value of a to-many relationship is the collection object. The collection can be a set or an array. If you use Core Data it is a set; otherwise, it is

typically an array) that contains the destination objects of that relationship. For example, the value of the `employees` property of an Department object is a collection containing Employee objects. Figure 8-5 shows an example object graph for the employee management application.

Figure 8-5 Object graph for the employee management application



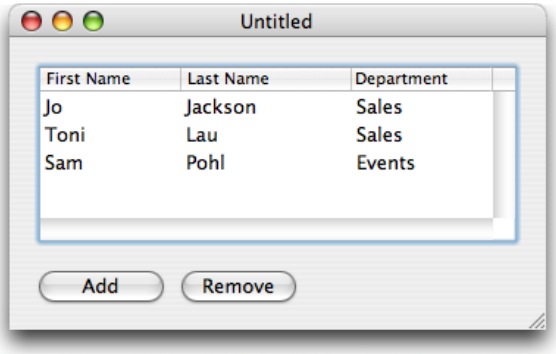
Key Paths

A *key path* is a string of dot-separated keys that specify a sequence of object properties to traverse. The property of the first key is determined by, and each subsequent key is evaluated relative to, the previous property. Key paths allow you to specify the properties of related objects in a way that is independent of the model implementation. Using key paths you can specify the path through an object graph, of whatever depth, to a specific attribute of a related object.

The key-value coding mechanism implements the lookup of a value given a key path similar to key-value pairs. For example, in the employee-management application you might access the name of a Department via an Employee object using the `department.name` key path where `department` is a relationship of Employee and `name` is an attribute of Department. Key paths are useful if you want to display an attribute of a destination

entity. For example, the employee table view in Figure 8-6 is configured to display the name of the employee's department object, not the department object itself. Using Cocoa bindings, the value of the Department column is bound to `department.name` of the Employee objects in the displayed array.

Figure 8-6 Employees table view showing department name



Not every relationship in a key path necessarily has a value. For example, the `manager` relationship can be `nil` if the employee is the CEO. In this case, the key-value coding mechanism does not break—it simply stops traversing the path and returns an appropriate value, such as `nil`.

Object Mutability

Cocoa objects are either mutable or immutable. You cannot change the encapsulated values of immutable objects; once such an object is created, the value it represents remains the same throughout the object's life. But you can change the encapsulated value of a mutable object at any time. The following sections explain the reasons for having mutable and immutable variants of an object type, describe the characteristics and side-effects of object mutability, and recommend how best to handle objects when their mutability is an issue.

Why Mutable and Immutable Object Variants?

Objects by default are mutable. Most objects allow you to change their encapsulated data through setter accessor methods. For example, you can change the size, positioning, title, buffering behavior, and other characteristics of an `NSWindow` object. A well-designed model object—say, an object representing a customer record—*requires* setter methods to change its instance data.

The Foundation framework adds some nuance to this picture by introducing classes that have mutable and immutable variants. The mutable subclasses are typically subclasses of their immutable superclass and have "Mutable" embedded in the class name. These classes include the following:

- `NSMutableArray`
- `NSMutableDictionary`
- `NSMutableSet`
- `NSMutableIndexSet`
- `NSMutableCharacterSet`
- `NSMutableData`
- `NSMutableString`
- `NSMutableAttributedString`
- `NSMutableURLRequest`

> **Note:** Except for `NSMutableParagraphStyle` in the AppKit framework, the Foundation framework currently defines all explicitly named mutable classes. However, any Cocoa framework can potentially have its own mutable and immutable class variants.

Although these classes have atypical names, they are closer to the mutable norm than their immutable counterparts. Why this complexity? What purpose does having an immutable variant of a mutable object serve?

Consider a scenario where all objects are capable of being mutated. In your application you invoke a method and are handed back a reference to an object representing a string. You use this string in your user interface to identify a particular piece of data. Now another subsystem in your application gets its own reference to that same string and decides to mutate it. Suddenly your label has changed out from under you. Things can become even more dire if, for instance, you get a reference to an array that you use to populate a table view. The user selects a row corresponding to an object in the array that has been removed by some code elsewhere in the program, and problems ensue. Immutability is a guarantee that an object won't unexpectedly change in value while you're using it.

Objects that are good candidates for immutability are ones that encapsulate collections of discrete values or contain values that are stored in buffers (which are themselves kinds of collections, either of characters or bytes). But not all such value objects necessarily benefit from having mutable versions. Objects that contain a single simple value, such as instances of `NSNumber` or `NSDate`, are not good candidates for mutability. When the represented value changes in these cases, it makes more sense to replace the old instance with a new instance.

Performance is also a reason for immutable versions of objects representing things such as strings and dictionaries. Mutable objects for basic entities such as strings and dictionaries bring some overhead with them. Because they must dynamically manage a changeable backing store—allocating and deallocating chunks of memory as needed—mutable objects can be less efficient than their immutable counterparts.

Although in theory immutability guarantees that an object's value is stable, in practice this guarantee isn't always assured. A method may choose to hand out a mutable object under the return type of its immutable variant; later, it may decide to mutate the object, possibly violating assumptions and choices the recipient has made based on the earlier value. The mutability of an object itself may change as it undergoes various transformations. For example, serializing a property list (using the `NSPropertyListSerialization` class) does not preserve the mutability aspect of objects, only their general kind—a dictionary, an array, and so on. Thus, when you deserialize this property list, the resulting objects might not be of the same class as the original objects. For instance, what was once an `NSMutableDictionary` object might now be a `NSDictionary` object.

Programming with Mutable Objects

When the mutability of objects is an issue, it's best to adopt some defensive programming practices. Here are a few general rules or guidelines:

- Use a mutable variant of an object when you need to modify its contents frequently and incrementally after it has been created.
- Sometimes it's preferable to replace one immutable object with another; for example, most instance variables that hold string values should be assigned immutable `NSString` objects that are replaced with setter methods.
- Rely on the return type for indications of mutability.
- If you have any doubts about whether an object is, or should be, mutable, go with immutable.

This section explores the gray areas in these guidelines, discussing typical choices you have to make when programming with mutable objects. It also gives an overview of methods in the Foundation framework for creating mutable objects and for converting between mutable and immutable object variants.

Creating and Converting Mutable Objects

You can create a mutable object through the standard nested `alloc-init` message—for example:

```
NSMutableDictionary *mutDict = [[NSMutableDictionary alloc] init];
```

However, many mutable classes offer initializers and factory methods that let you specify the initial or probable capacity of the object, such as the `arrayWithCapacity:` class method of `NSMutableArray`:

```
NSMutableArray *mutArray = [NSMutableArray arrayWithCapacity:[timeZones count]];
```

The capacity hint enables more efficient storage of the mutable object's data. (Because the convention for class factory methods is to return autoreleased instances, be sure to retain the object if you wish to keep it viable in your code.)

You can also create a mutable object by making a mutable copy of an existing object of that general type. To do so, invoke the `mutableCopy` method that each immutable super class of a Foundation mutable class implements:

```
NSMutableSet *mutSet = [aSet mutableCopy];
```

In the other direction, you can send `copy` to a mutable object to make an immutable copy of the object.

Many Foundation classes with immutable and mutable variants include methods for converting between the variants, including:

- *type*`WithType:`—for example, `arrayWithArray:`
- `set`*Type*`:`—for example, `setString:` (mutable classes only)
- `initWith`*Type*`:copyItems:`—for example, `initWithDictionary:copyItems:`

Storing and Returning Mutable Instance Variables

In Cocoa development you often have to decide whether to make an instance variable mutable or immutable. For an instance variable whose value can change, such as a dictionary or string, when is it appropriate to make the object mutable? And when is it better to make the object immutable and replace it with another object when its represented value changes?

Generally, when you have an object whose contents change wholesale, it's better to use an immutable object. Strings (`NSString`) and data objects (`NSData`) usually fall into this category. If an object is likely to change incrementally, it is a reasonable approach to make it mutable. Collections such as arrays and dictionaries fall into this category. However, the frequency of changes and the size of the collection should be factors in this decision. For example, if you have a small array that seldom changes, it's better to make it immutable.

There are a couple of other considerations when deciding on the mutability of a collection held as an instance variable:

- If you have a mutable collection that is frequently changed and that you frequently hand out to clients (that is, you return it directly in a getter accessor method), you run the risk of mutating something that your clients might have a reference to. If this risk is probable, the instance variable should be immutable.
- If the value of the instance variable frequently changes but you rarely return it to clients in getter methods, you can make the instance variable mutable but return an immutable copy of it in your accessor method; in memory-managed programs, this object would be autoreleased (Listing 9-1).

Listing 9-1 Returning an immutable copy of a mutable instance variable

```
@interface MyClass : NSObject {
    // ...
    NSMutableSet *widgets;
}
// ...
@end
```

```
@implementation MyClass
- (NSSet *)widgets {
    return (NSSet *)[[widgets copy] autorelease];
}
```

One sophisticated approach for handling mutable collections that are returned to clients is to maintain a flag that records whether the object is currently mutable or immutable. If there is a change, make the object mutable and apply the change. When handing out the collection, make the object immutable (if necessary) before returning it.

Receiving Mutable Objects

The invoker of a method is interested in the mutability of a returned object for two reasons:

- It wants to know if it can change the object's value.
- It wants to know if the object's value will change unexpectedly while it has a reference to it.

Use Return Type, Not Introspection

To determine whether it can change a received object, the receiver of a message must rely on the formal type of the return value. If it receives, for example, an array object typed as immutable, it should not attempt to mutate it. It is not an acceptable programming practice to determine if an object is mutable based on its class membership—for example:

```
if ( [anArray isKindOfClass:[NSMutableArray class]] ) {
    // add, remove objects from anArray
}
```

For reasons related to implementation, what `isKindOfClass:` returns in this case may not be accurate. But for reasons other than this, you should not make assumptions about whether an object is mutable based on class membership. Your decision should be guided solely by what the signature of the method vending the object says about its mutability. If you are not sure whether an object is mutable or immutable, assume it's immutable.

A couple of examples might help clarify why this guideline is important:

- You read a property list from a file. When the Foundation framework processes the list, it notices that various subsets of the property list are identical, so it creates a set of objects that it shares among all those subsets. Afterward you look at the created property list objects and decide to mutate one subset. Suddenly, and without being aware of it, you've changed the tree in multiple places.
- You ask `NSView` for its subviews (with the `subviews` method) and it returns an object that is declared to be an `NSArray` but which could be an `NSMutableArray` internally. Then you pass that array to some other code that, through introspection, determines it to be mutable and changes it. By changing this array, the code is mutating internal data structures of the `NSView` class.

So don't make an assumption about object mutability based on what introspection tells you about an object. Treat objects as mutable or not based on what you are handed at the API boundaries (that is, based on the return type). If you need to unambiguously mark an object as mutable or immutable when you pass it to clients, pass that information as a flag along with the object.

Make Snapshots of Received Objects

If you want to ensure that a supposedly immutable object received from a method does not mutate without your knowing about it, you can make snapshots of the object by copying it locally. Then occasionally compare the stored version of the object with the most recent version. If the object has mutated, you can adjust anything in your program that is dependent on the previous version of the object. Listing 9-2 shows a possible implementation of this technique.

Listing 9-2 Making a snapshot of a potentially mutable object

```
static NSArray *snapshot = nil;
- (void)myFunction {
    NSArray *thingArray = [otherObj things];
    if (snapshot) {
        if ( ![thingArray isEqualToArray:snapshot] ) {
            [self updateStateWith:thingArray];
        }
    }
    snapshot = [thingArray copy];
}
```

A problem with making snapshots of objects for later comparison is that it is expensive. You're required to make multiple copies of the same object. A more efficient alternative is to use key-value observing. See *Key-Value Observing Programming Guide* for a description of this protocol.

Mutable Objects in Collections

Storing mutable objects in collection objects can cause problems. Certain collections can become invalid or even corrupt if objects they contain mutate because, by mutating, these objects can affect the way they are placed in the collection. First, the properties of objects that are keys in hashing collections such as `NSDictionary` objects or `NSSet` objects will, if changed, corrupt the collection if the changed properties affect the results of the object's `hash` or `isEqual:` methods. (If the `hash` method of the objects in the collection does not depend on their internal state, corruption is less likely.) Second, if an object in an ordered collection such as a sorted array has its properties changed, this might affect how the object compares to other objects in the array, thus rendering the ordering invalid.

Outlets

An outlet is a property of an object that references another object. The reference is archived through Interface Builder. The connections between the containing object and its outlets are reestablished every time the containing object is unarchived from its nib file. The containing object holds an outlet declared as a property with the type qualifier of `IBOutlet` and a `weak` option. For example:

```
@interface AppController : NSObject
{
}
@property (weak) IBOutlet NSArray *keywords;
```

Because it is a property, an outlet becomes part of an object's encapsulated data and is backed by an instance variable. But an outlet is more than a simple property. The connection between an object and its outlets is archived in a nib file; when the nib file is loaded, each connection is unarchived and reestablished, and is thus always available whenever it becomes necessary to send messages to the other object. The type qualifier `IBOutlet` is a tag applied to an property declaration so that the Interface Builder application can recognize the property as an outlet and synchronize the display and connection of it with Xcode.

An outlet is declared as a weak reference (`weak`) to prevent strong reference cycles.

You create and connect an outlet in the Interface Builder feature of Xcode.The property declaration for the outlet must be tagged with the `IBOutlet` qualifier.

An application typically sets outlet connections between its custom controller objects and objects on the user interface, but they can be made between any objects that can be represented as instances in Interface Builder, even between two custom objects. As with any item of object state, you should be able to justify its inclusion in a class; the more outlets an object has, the more memory it takes up. If there are other ways to obtain a reference to an object, such as finding it through its index position in a matrix, or through its inclusion as a function parameter, or through use of a tag (an assigned numeric identifier), you should do that instead.

Outlets are a form of object composition, which is a dynamic pattern that requires an object to somehow acquire references to its constituent objects so that it can send messages to them. It typically holds these other objects as properties backed by instance variables. These variables must be initialized with the appropriate references at some point during the execution of the program.

Receptionist Pattern

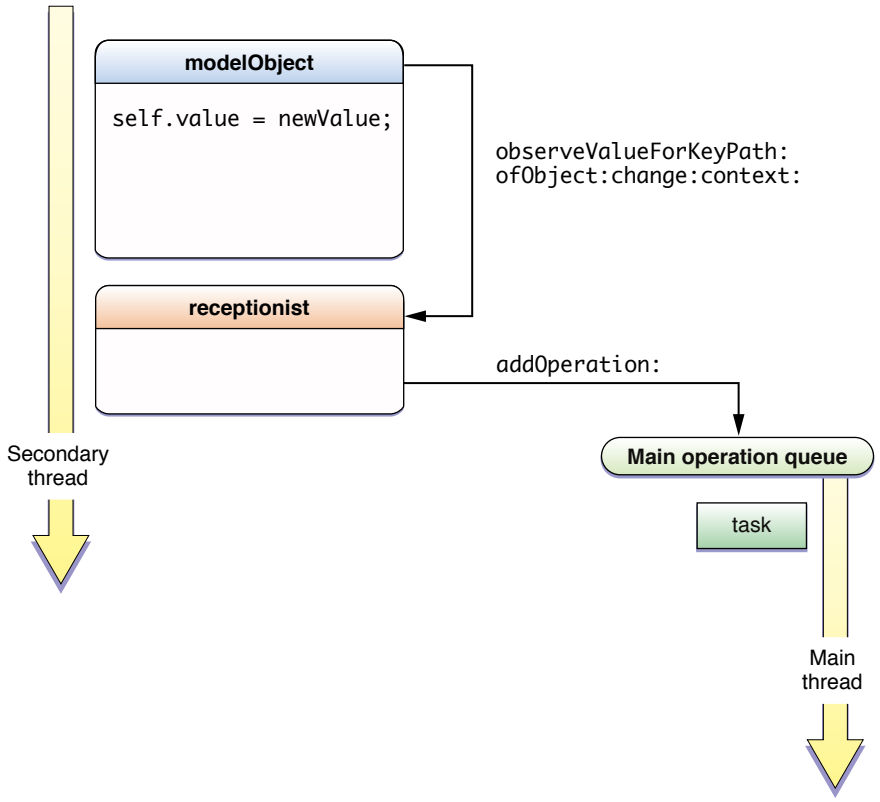
The Receptionist design pattern addresses the general problem of redirecting an event occurring in one execution context of an application to another execution context for handling. It is a hybrid pattern. Although it doesn't appear in the "Gang of Four" book, it combines elements of the Command, Memo, and Proxy design patterns described in that book. It is also a variant of the Trampoline pattern (which also doesn't appear in the book); in this pattern, an event initially is received by a trampoline object, so-called because it immediately bounces, or redirects, the event to a target object for handling.

The Receptionist Design Pattern in Practice

A KVO notification invokes the `observeValueForKeyPath:ofObject:change:context:` method implemented by an observer. If the change to the property occurs on a secondary thread, the `observeValueForKeyPath:ofObject:change:context:` code executes on that same thread. There the central object in this pattern, the receptionist, acts as a thread intermediary. As Figure 11-1 illustrates, a receptionist object is assigned as the observer of a model object's property. The receptionist implements `observeValueForKeyPath:ofObject:change:context:` to redirect the notification received on a secondary thread to another execution context—the main operation queue, in this case. When the property

changes, the receptionist receives a KVO notification. The receptionist immediately adds a block operation to the main operation queue; the block contains code—specified by the client—that updates the user interface appropriately.

Figure 11-1 Bouncing KVO updates to the main operation queue



You define a receptionist class so that it has the elements it needs to add itself as an observer of a property and then convert a KVO notification into an update task. Thus it must know what object it's observing, the property of the object that it's observing, what update task to execute, and what queue to execute it on. Listing 11-1 shows the initial declaration of the `RCReceptionist` class and its instance variables.

Listing 11-1 Declaring the receptionist class

```
@interface RCReceptionist : NSObject {
    id observedObject;
    NSString *observedKeyPath;
    RCTaskBlock task;
    NSOperationQueue *queue;
}
```

The `RCTaskBlock` instance variable is a block object of the following declared type:

```
typedef void (^RCTaskBlock)(NSString *keyPath, id object, NSDictionary *change);
```

These parameters are similar to those of the `observeValueForKeyPath:ofObject:change:context:` method. Next, the parameter class declares a single class factory method in which an `RCTaskBlock` object is a parameter:

```
+ (id)receptionistForKeyPath:(NSString *)path
        object:(id)obj
         queue:(NSOperationQueue *)queue
          task:(RCTaskBlock)task;
```

It implements this method to assign the passed-in value to instance variables of the created receptionist object and to add that object as an observer of the model object's property, as shown in Listing 11-2.

Listing 11-2 The class factory method for creating a receptionist object

```
+ (id)receptionistForKeyPath:(NSString *)path object:(id)obj queue:(NSOperationQueue
 *)queue task:(RCTaskBlock)task {
    RCReceptionist *receptionist = [RCReceptionist new];
    receptionist->task = [task copy];
    receptionist->observedKeyPath = [path copy];
    receptionist->observedObject = [obj retain];
    receptionist->queue = [queue retain];
    [obj addObserver:receptionist forKeyPath:path
             options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
context:0];
    return [receptionist autorelease];
}
```

Note that the code copies the block object instead of retaining it. Because the block was probably created on the stack, it must be copied to the heap so it exists in memory when the KVO notification is delivered.

Finally, the parameter class implements the `observeValueForKeyPath:ofObject:change:context:` method. The implementation (see Listing 11-3) is simple.

Listing 11-3 Handling the KVO notification

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
        change:(NSDictionary *)change context:(void *)context {
    [queue addOperationWithBlock:^{
        task(keyPath, object, change);
    }];
}
```

This code simply enqueues the task onto the given operation queue, passing the task block the observed object, the key path for the changed property, and the dictionary containing the new value. The task is encapsulated in an `NSBlockOperation` object that executes the task on the queue.

The client object supplies the block code that updates the user interface when it creates a receptionist object, as shown in Listing 11-4. Note that when it creates the receptionist object, the client passes in the operation queue on which the block is to be executed, in this case the main operation queue.

Listing 11-4 Creating a receptionist object

```
        RCReceptionist *receptionist = [RCReceptionist
receptionistForKeyPath:@"value" object:model queue:mainQueue task:^(NSString
*keyPath, id object, NSDictionary *change) {
            NSView *viewForModel = [modelToViewMap objectForKey:model];
            NSColor *newColor = [change objectForKey:NSKeyValueChangeNewKey];
            [[[viewForModel subviews] objectAtIndex:0] setFillColor:newColor];
        }];
```

When to Use the Receptionist Pattern

You can adopt the Receptionist design pattern whenever you need to bounce off work to another execution context for handling. When you observe a notification, or implement a block handler, or respond to an action message and you want to ensure that your code executes in the appropriate execution context, you can implement the Receptionist pattern to redirect the work that must be done to that execution context. With the Receptionist pattern, you might even perform some filtering or coalescing of the incoming data before you bounce off a task to process the data. For example, you could collect data into batches, and then at intervals dispatch those batches elsewhere for processing.

One common situation where the Receptionist pattern is useful is key-value observing. In key-value observing, changes to the value of an model object's property are communicated to observers via KVO notifications. However, changes to a model object can occur on a background thread. This results in a thread mismatch, because changes to a model object's state typically result in updates to the user interface, and these must occur on the main thread. In this case, you want to redirect the KVO notifications to the main thread. where the updates to an application's user interface can occur.

Target-Action

Although delegation, bindings, and notification are useful for handling certain forms of communication between objects in a program, they are not particularly suitable for the most visible sort of communication. A typical application's user interface consists of a number of graphical objects, and perhaps the most common of these objects are controls. A control is a graphical analog of a real-world or logical device (button, slider, checkboxes, and so on); as with a real-world control, such as a radio tuner, you use it to convey your intent to some system of which it is a part—that is, an application.

The role of a control on a user interface is simple: It interprets the intent of the user and instructs some other object to carry out that request. When a user acts on the control by, say, clicking it or pressing the Return key, the hardware device generates a raw event. The control accepts the event (as appropriately packaged for Cocoa) and translates it into an instruction that is specific to the application. However, events by themselves don't give much information about the user's intent; they merely tell you that the user clicked a mouse button or pressed a key. So some mechanism must be called upon to provide the translation between event and instruction. This mechanism is called *target-action* .

Cocoa uses the target-action mechanism for communication between a control and another object. This mechanism allows the control and, in OS X its cell or cells, to encapsulate the information necessary to send an application-specific instruction to the appropriate object. The receiving object—typically an instance of a custom class—is called the *target* . The *action* is the message that the control sends to the target. The object that is interested in the user event—the target—is the one that imparts significance to it, and this significance is usually reflected in the name it gives to the action.

The Target

A target is a receiver of an action message. A control or, more frequently, its cell holds the target of its action message as an outlet (see "Outlets" (page 67)). The target usually is an instance of one of your custom classes, although it can be any Cocoa object whose class implements the appropriate action method.

You can also set a cell's or control's target outlet to `nil` and let the target object be determined at runtime. When the target is `nil`, the application object (`NSApplication` or `UIApplication`) searches for an appropriate receiver in a prescribed order:

1. It begins with the first responder in the key window and follows `nextResponder` links up the responder chain to the window object's (`NSWindow` or `UIWindow`) content view.

> **Note:** A key window in OS X responds to key presses for an application and is the receiver of messages from menus and dialogs. An application's main window is the principal focus of user actions and often has key status as well.

2. It tries the window object and then the window object's delegate.
3. If the main window is different from the key window, it then starts over with the first responder in the main window and works its way up the main window's responder chain to the window object and its delegate.
4. Next, the application object tries to respond. If it can't respond, it tries its delegate. The application object and its delegate are the receivers of last resort.

Control objects do not (and should not) retain their targets. However, clients of controls sending action messages (applications, usually) are responsible for ensuring that their targets are available to receive action messages. To do this, they may have to retain their targets in memory-managed environments. This precaution applies equally to delegates and data sources.

The Action

An action is the message a control sends to the target or, from the perspective of the target, the method the target implements to respond to the action message. A control or—as is frequently the case in AppKit—a control's cell stores an action as an instance variable of type `SEL`. `SEL` is an Objective-C data type used to specify the signature of a message. An action message must have a simple, distinct signature. The method it invokes returns nothing and usually has a sole parameter of type `id`. This parameter, by convention, is named `sender`. Here is an example from the `NSResponder` class, which defines a number of action methods:

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

Action methods declared by some Cocoa classes can also have the equivalent signature:

```
- (IBAction) deleteRecord:(id)sender;
```

In this case, `IBAction` does not designate a data type for a return value; no value is returned. `IBAction` is a type qualifier that Interface Builder notices during application development to synchronize actions added programmatically with its internal list of action methods defined for a project.

> **iOS Note:** In UIKit, action selectors can also take two other forms. See "Target-Action in UIKit" (page 78) for details.

The `sender`parameter usually identifies the control sending the action message (although it can be another object substituted by the actual sender). The idea behind this is similar to a return address on a postcard. The target can query the sender for more information if it needs to. If the actual sending object substitutes another object as sender, you should treat that object in the same way. For example, say you have a text field and when the user enters text, the action method `nameEntered:` is invoked in the target:

```
- (void)nameEntered:(id) sender {
    NSString *name = [sender stringValue];
    if (![name isEqualToString:@""]) {
        NSMutableArray *names = [self nameList];
        [names addObject:name];
        [sender setStringValue:@""];
    }
}
```

Here the responding method extracts the contents of the text field, adds the string to an array cached as an instance variable, and clears the field. Other possible queries to the sender would be asking an `NSMatrix` object for its selected row (`[sender selectedRow]`), asking an `NSButton` object for its state (`[sender state]`), and asking any cell associated with a control for its tag (`[[sender cell] tag]`), a tag being a numeric identifier.

Target-Action in the AppKit Framework

The AppKit framework uses specific architectures and conventions in implementing target-action.

Controls, Cells, and Menu Items

Most controls in AppKit are objects that inherit from the `NSControl` class. Although a control has the initial responsibility for sending an action message to its target, it rarely carries the information needed to send the message. For this, it usually relies on its cell or cells.

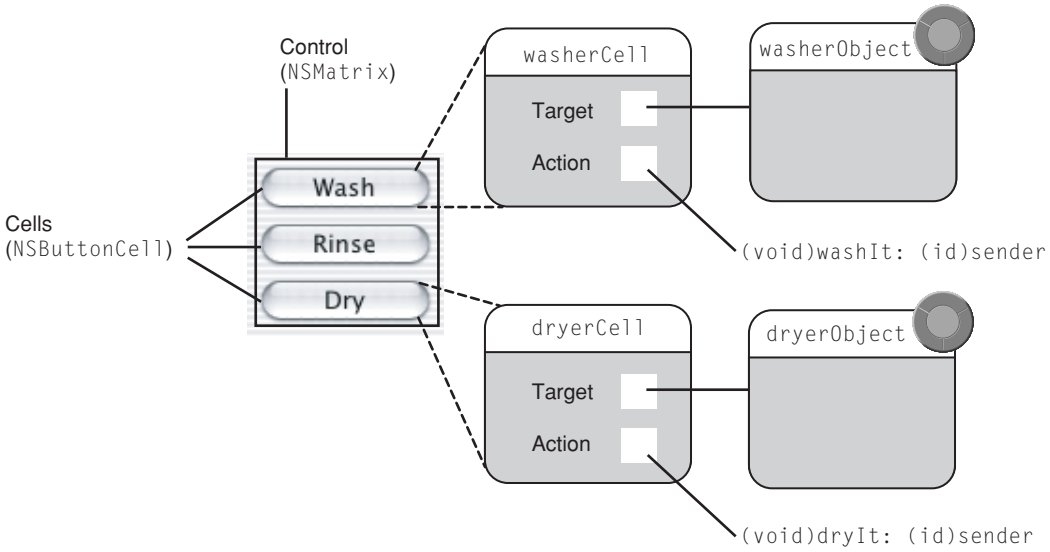
A control almost always has one or more cells—objects that inherit from `NSCell`—associated with it. Why is there this association? A control is a relatively "heavy" object because it inherits all the combined instance variables of its ancestors, which include the `NSView` and `NSResponder` classes. Because controls are expensive,

cells are used to subdivide the screen real estate of a control into various functional areas. Cells are lightweight objects that can be thought of as overlaying all or part of the control. But it's not only a division of area, it's a division of labor. Cells do some of the drawing that controls would otherwise have to do, and cells hold some of the data that controls would otherwise have to carry. Two items of this data are the instance variables for target and action. Figure 12-1 (page 76) depicts the control-cell architecture.

Being abstract classes, `NSControl` and `NSCell` both incompletely handle the setting of the target and action instance variables. By default, `NSControl` simply sets the information in its associated cell, if one exists. (`NSControl` itself supports only a one-to-one mapping between itself and a cell; subclasses of `NSControl` such as `NSMatrix` support multiple cells.) In its default implementation, `NSCell` simply raises an exception. You must go one step further down the inheritance chain to find the class that really implements the setting of target and action: `NSActionCell`.

Objects derived from `NSActionCell` provide target and action values to their controls so the controls can compose and send an action message to the proper receiver. An `NSActionCell` object handles mouse (cursor) tracking by highlighting its area and assisting its control in sending action messages to the specified target. In most cases, the responsibility for an `NSControl` object's appearance and behavior is completely given over to a corresponding `NSActionCell` object. (`NSMatrix`, and its subclass `NSForm`, are subclasses of `NSControl` that don't follow this rule.)

Figure 12-1 How the target-action mechanism works in the control-cell architecture



When users choose an item from a menu, an action is sent to a target. Yet menus (`NSMenu` objects) and their items (`NSMenuItem` objects) are completely separate, in an architectural sense, from controls and cells. The `NSMenuItem` class implements the target-action mechanism for its own instances; an `NSMenuItem` object has both target and action instance variables (and related accessor methods) and sends the action message to the target when a user chooses it.

> **Note:** See *Control and Cell Programming Topics for Cocoa* and *Application Menu and Pop-up List Programming Topics* for more information about the control-cell architecture.

Setting the Target and Action

You can set the targets and actions of cells and controls programmatically or by using Interface Builder. For most developers and most situations, Interface Builder is the preferred approach. When you use it to set controls and targets, Interface Builder provides visual confirmation, allows you to lock the connections, and archives the connections to a nib file. The procedure is simple:

1. Declare an action method in the header file of your custom class that has the `IBAction` qualifier.
2. In Interface Builder, connect the control sending the message to the action method of the target.

If the action is handled by a superclass of your custom class or by an off-the-shelf AppKit or UIKit class, you can make the connection without declaring any action method. Of course, if you declare an action method yourself, you must be sure to implement it.

To set the action and the target programmatically, use the following methods to send messages to a control or cell object:

```
- (void)setTarget:(id)anObject;
- (void)setAction:(SEL)aSelector;
```

The following example shows how you might use these methods:

```
[aCell setTarget:myController];
[aControl setAction:@selector(deleteRecord:)];
[aMenuItem setAction:@selector(showGuides:)];
```

Programmatically setting the target and action does have its advantages and in certain situations it is the only possible approach. For example, you might want the target or action to vary according to some runtime condition, such as whether a network connection exists or whether an inspector window has been loaded. Another example is when you are dynamically populating the items of a pop-up menu, and you want each pop-up item to have its own action.

Actions Defined by AppKit

The AppKit framework not only includes many `NSActionCell`-based controls for sending action messages, it defines action methods in many of its classes. Some of these actions are connected to default targets when you create a Cocoa application project. For example, the Quit command in the application menu is connected to the `terminate:` method in the global application object (`NSApp`).

The `NSResponder` class also defines many default action messages (also known as *standard commands*) for common operations on text. This allows the Cocoa text system to send these action messages up an application's responder chain—a hierarchical sequence of event-handling objects—where it can be handled by the first `NSView`, `NSWindow`, or `NSApplication` object that implements the corresponding method.

Target-Action in UIKit

The UIKit framework also declares and implements a suite of control classes; the control classes in this framework inherit from the `UIControl` class, which defines most of the target-action mechanism for iOS. However there are some fundamental differences in how the AppKit and UIKit frameworks implement target-action. One of these differences is that UIKit does not have any true cell classes. Controls in UIKit do not rely upon their cells for target and action information.

A larger difference in how the two frameworks implement target-action lies in the nature of the event model. In the AppKit framework, the user typically uses a mouse and keyboard to register events for handling by the system. These events—such as clicking on a button—are limited and discrete. Consequently, a control object in AppKit usually recognizes a single physical event as the trigger for the action it sends to its target. (In the case of buttons, this is a mouse-up event.) In iOS, the user's fingers are what originate events instead of mouse clicks, mouse drags, or physical keystrokes. There can be more than one finger touching an object on the screen at one time, and these touches can even be going in different directions.

To account for this multitouch event model, UIKit declares a set of control-event constants in `UIControl.h` that specify various physical gestures that users can make on controls, such as lifting a finger from a control, dragging a finger into a control, and touching down within a text field. You can configure a control object so that it responds to one or more of these touch events by sending an action message to a target. Many of the

control classes in UIKit are implemented to generate certain control events; for example, instances of the `UISlider` class generate a `UIControlEventValueChanged` control event, which you can use to send an action message to a target object.

You set up a control so that it sends an action message to a target object by associating both target and action with one or more control events. To do this, send `addTarget:action:forControlEvents:` to the control for each target-action pair you want to specify. When the user touches the control in a designated fashion, the control forwards the action message to the global `UIApplication` object in a `sendAction:to:from:forEvent:` message. As in AppKit, the global application object is the centralized dispatch point for action messages. If the control specifies a `nil` target for an action message, the application queries objects in the responder chain until it finds one that is willing to handle the action message—that is, one implementing a method corresponding to the action selector.

In contrast to the AppKit framework, where an action method may have only one or perhaps two valid signatures, the UIKit framework allows three different forms of action selector:

```
- (void)action
- (void)action:(id)sender
- (void)action:(id)sender forEvent:(UIEvent *)event
```

To learn more about the target-action mechanism in UIKit, read *UIControl Class Reference*.

Toll-Free Bridging

There are a number of data types in the Core Foundation framework and the Foundation framework that can be used interchangeably. This capability, called *toll-free bridging* , means that you can use the same data type as the parameter to a Core Foundation function call or as the receiver of an Objective-C message. For example, `NSLocale` (see *NSLocale Class Reference*) is interchangeable with its Core Foundation counterpart, CFLocale (see *CFLocale Reference*). Therefore, in a method where you see an `NSLocale *` parameter, you can pass a `CFLocaleRef`, and in a function where you see a `CFLocaleRef` parameter, you can pass an `NSLocale` instance. You cast one type to the other to suppress compiler warnings, as illustrated in the following example.

```
NSLocale *gbNSLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_GB"];
CFLocaleRef gbCFLocale = (CFLocaleRef) gbNSLocale;
CFStringRef cfIdentifier = CFLocaleGetIdentifier (gbCFLocale);
NSLog(@"cfIdentifier: %@", (NSString *)cfIdentifier);
// logs: "cfIdentifier: en_GB"
CFRelease((CFLocaleRef) gbNSLocale);

CFLocaleRef myCFLocale = CFLocaleCopyCurrent();
NSLocale * myNSLocale = (NSLocale *) myCFLocale;
[myNSLocale autorelease];
NSString *nsIdentifier = [myNSLocale localeIdentifier];
CFShow((CFStringRef) [@"nsIdentifier: " stringByAppendingString:nsIdentifier]);
// logs identifier for current locale
```

Note from the example that the memory management functions and methods are also interchangeable—you can use `CFRelease` with a Cocoa object and `release` and `autorelease` with a Core Foundation object.

Note: When using garbage collection, there are important differences to how memory management works for Cocoa objects and Core Foundation objects. See "Using Core Foundation with Garbage Collection" for details.

Toll-free bridging has been available since OS X v10.0. Table 13-1 provides a list of the data types that are interchangeable between Core Foundation and Foundation. For each pair, the table also lists the version of OS X in which toll-free bridging between them became available.

Table 13-1 Data types that can be used interchangeably between Core Foundation and Foundation

Core Foundation type	Foundation class	Availability
`CFArrayRef`	`NSArray`	OS X v10.0
`CFAttributedStringRef`	`NSAttributedString`	OS X v10.4
`CFCalendarRef`	`NSCalendar`	OS X v10.4
`CFCharacterSetRef`	`NSCharacterSet`	OS X v10.0
`CFDataRef`	`NSData`	OS X v10.0
`CFDateRef`	`NSDate`	OS X v10.0
`CFDictionaryRef`	`NSDictionary`	OS X v10.0
`CFErrorRef`	`NSError`	OS X v10.5
`CFLocaleRef`	`NSLocale`	OS X v10.4
`CFMutableArrayRef`	`NSMutableArray`	OS X v10.0
`CFMutableAttributedStringRef`	`NSMutableAttributedString`	OS X v10.4
`CFMutableCharacterSetRef`	`NSMutableCharacterSet`	OS X v10.0
`CFMutableDataRef`	`NSMutableData`	OS X v10.0
`CFMutableDictionaryRef`	`NSMutableDictionary`	OS X v10.0
`CFMutableSetRef`	`NSMutableSet`	OS X v10.0
`CFMutableStringRef`	`NSMutableString`	OS X v10.0
`CFNumberRef`	`NSNumber`	OS X v10.0
`CFReadStreamRef`	`NSInputStream`	OS X v10.0

Core Foundation type	Foundation class	Availability
CFRunLoopTimerRef	NSTimer	OS X v10.0
CFSetRef	NSSet	OS X v10.0
CFStringRef	NSString	OS X v10.0
CFTimeZoneRef	NSTimeZone	OS X v10.0
CFURLRef	NSURL	OS X v10.0
CFWriteStreamRef	NSOutputStream	OS X v10.0

Note: Not all data types are toll-free bridged, even though their names might suggest that they are. For example, NSRunLoop is not toll-free bridged to CFRunLoop, NSBundle is not toll-free bridged to CFBundle, and NSDateFormatter is not toll-free bridged to CFDateFormatter.

Document Revision History

This table describes the changes to *Concepts in Objective-C Programming* .

Date	Notes
2012-01-09	Descriptions of design patterns, architectures, and other concepts important in Cocoa and Cocoa Touch development.

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