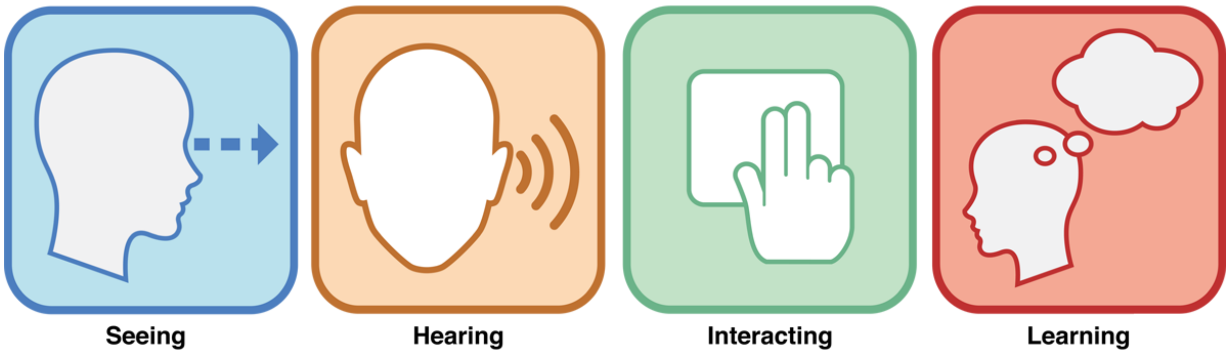
# Cocoa Core Competencies

## Accessibility

Accessible apps can be used by everyone, regardless of their limitations or disabilities. By making your app accessible, you can reach broader markets and expand your user base.

Your users do not necessarily have the same abilities as you. Consider your app's user experience from the perspective of someone with a seeing, hearing, interacting, or learning impairment. Is your app still operable if someone with one of these impairments tries to use it?

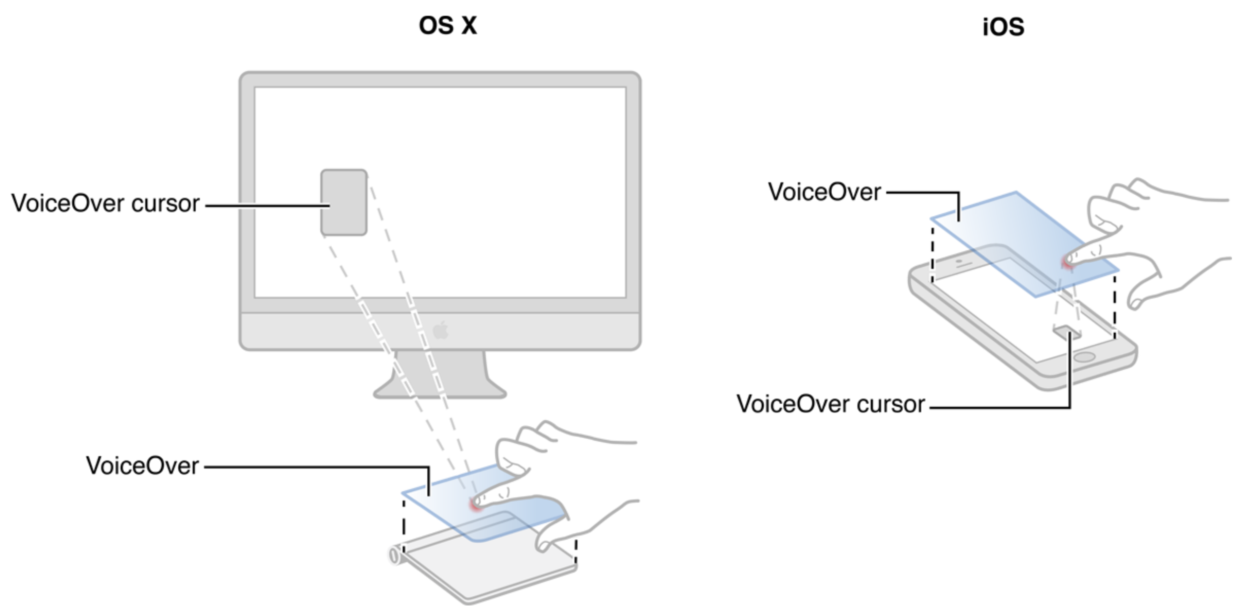


Apple has done the majority of the work on the system level to make your app accessible—by providing features like screen zoom, visual alerts, AssistiveTouch, and Guided Access—but there is more you can do as a developer to enhance the accessibility of your app.

### Working with VoiceOver

The primary way OS X and iOS apps present information is through a graphical user interface (**GUI**), so you need to pay particular attention to visual accessibility. Blind and low-vision users interact with your app using VoiceOver, a screen-reading technology built into the operating system that speaks your app's user interface aloud. VoiceOver users use special gestures or keyboard commands to explore and control the GUI. Standard inputs—such as mouse movements and finger swipes—are translated by VoiceOver to move the VoiceOver cursor, which reads an element's accessibility information.

VoiceOver helps low-vision users navigate the GUI by interpreting touches differently



The standard UI elements provided by AppKit and UIKit are accessible to VoiceOver by default. You only need to provide a description for visual UI elements such as images or icons that don't already have text associated with them. You can change an element's accessibility description and other default accessibility behavior—such as the role of an element, or whether VoiceOver should read the element—directly in Interface Builder.

Custom UI elements and views, on the other hand, must conform to the `NSAccessibility` or `UIAccessibility` protocol so that they can describe themselves to VoiceOver to be read aloud. These are the same protocols that standard controls in AppKit and UIKit adopt. By adopting these protocols and implementing their methods, you provide VoiceOver the information it needs to make your custom UI elements accessible.

The best way to confirm that your app works well with VoiceOver is to interact with your app using VoiceOver. Enable VoiceOver on iOS in Settings > General > Accessibility and on OS X in System Preferences > Accessibility (or hit Command-F5). VoiceOver is a sophisticated tool, but it only takes a few minutes to learn the basics. Navigate through your user interface using VoiceOver to make sure all your features are accessible and all of your UI elements have appropriate descriptions. Verify that your app promotes a positive accessibility experience by following the advice in "Testing the Accessibility of Your iPhone Application".

## Related Articles

Internationalization

## Definitive Discussion

Accessibility Programming Guide for iOS

## Sample Code Projects

ImageMapExample

# Accessor method

An accessor method is an instance method that gets or sets the value of a property of an object. In Cocoa's terminology, a method that retrieves the value of an object's property is referred to as a getter method, or "getter;" a method that changes the value of an object's property is referred to as a setter method, or "setter." These methods are often found in pairs, providing API for getting and setting the property values of an object.

You should use accessor methods rather than directly accessing state data because they provide an abstraction layer. Here are just two of the benefits that accessor methods provide:

- You don't need to rewrite your code if the manner in which a property is represented or stored changes.
- Accessor methods often implement important behavior that occurs whenever a value is retrieved or set. For example, setter methods frequently implement memory management code and notify other objects when a value is changed.

## Naming Conventions

Because of the importance of this pattern, Cocoa defines some conventions for naming accessor methods. Given a property of type `type` and called `name`, you should typically implement accessor methods with the following form:

```
- (type)name;


- (void)setName:(type)newName;
```

The one exception is a property that is a Boolean value. Here the getter method name may be `isName`. For example:

```
- (BOOL)isHidden;


- (void)setHidden:(BOOL)newHidden;
```

This naming convention is important because much other functionality in Cocoa relies upon it, in particular key-value coding. Cocoa does not use `getName` because methods that start with “get” in Cocoa indicate that the method will return values by reference.

### Prerequisite Articles

Key-value coding

### Related Articles

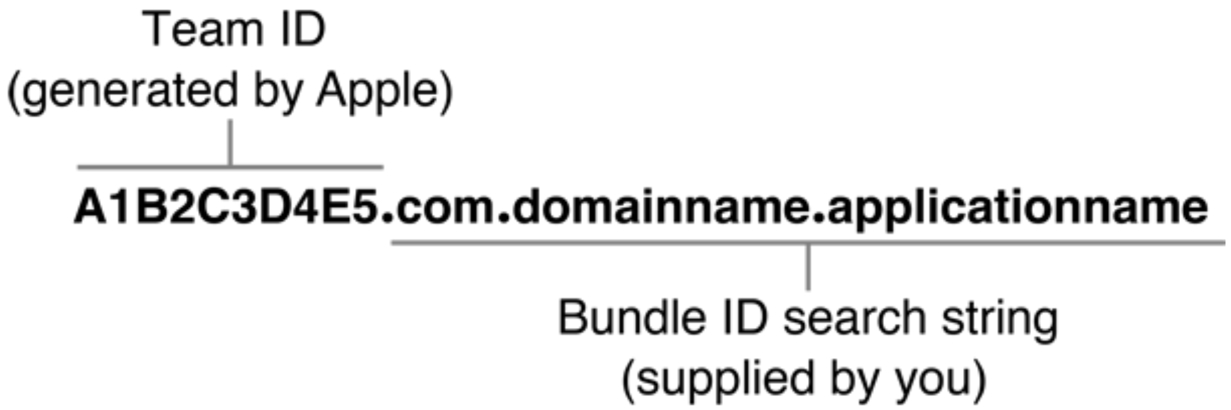
Memory management
Declared property

### Definitive Discussion

“Use Accessor Methods to Get or Set Property Values” in Programming with Objective-C

## App ID

An App ID is a two-part string used to identify one or more apps from a single development team. The string consists of a **Team ID** and a **bundle ID search string**, with a period (.) separating the two parts. The Team ID is supplied by Apple and is unique to a specific development team, while the bundle ID search string is supplied by you to match either the bundle ID of a single app or a set of bundle IDs for a group of your apps.



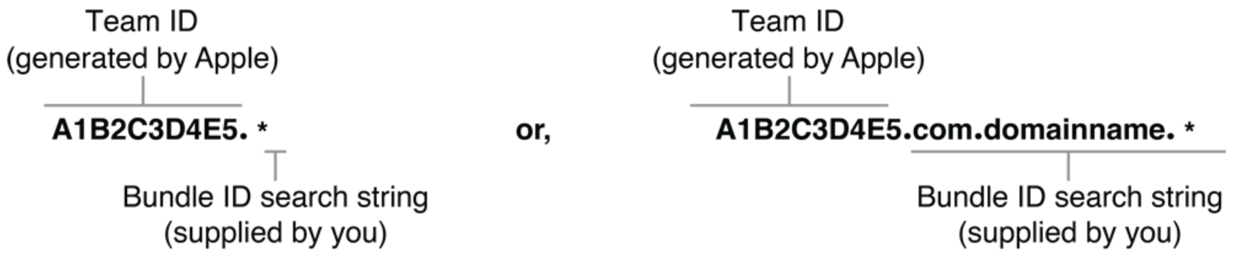
There are two types of App IDs: an explicit App ID, used for a single app, and wildcard App IDs, used for a set of apps.

## An Explicit App ID Matches a Single App

For an explicit App ID to match an app, the Team ID in the App ID must equal the Team ID associated with the app, and the bundle ID search string must equal the bundle ID for the app. The bundle ID is a unique identifier that identifies a single app and cannot be used by other teams.

## Wildcard App IDs Match Multiple Apps

A wildcard App ID contains an asterisk as the last part of its bundle ID search string. The asterisk replaces some or all of the bundle ID in the search string.



The asterisk is treated as a wildcard when matching the bundle ID search string with bundle IDs. For a wildcard App ID to match a set of apps, the bundle ID must exactly match all of the characters preceding the asterisk in the bundle ID search string. The asterisk matches all remaining characters in the bundle ID. The asterisk must match at least one character in the bundle ID. The table below shows a bundle ID search string and some matching and nonmatching bundle IDs.

| com.domain.* | | (bundle id search string) |
|---|---|---|
| com.domain.text | ✓ | * matches text. |
| com.domain.icon | ✓ | * matches icon |
| com.otherdomain.database | ✗ | The d in the pattern fails to find a match. |
| com.domain | ✗ | The . in the pattern fails to find a match. |
| com.domain. | ✗ | The * in the pattern fails to match a character. |

For a wildcard App ID to match an app, the Team ID must match exactly and the bundle ID must match the bundle ID search string using the wildcard matching rules.

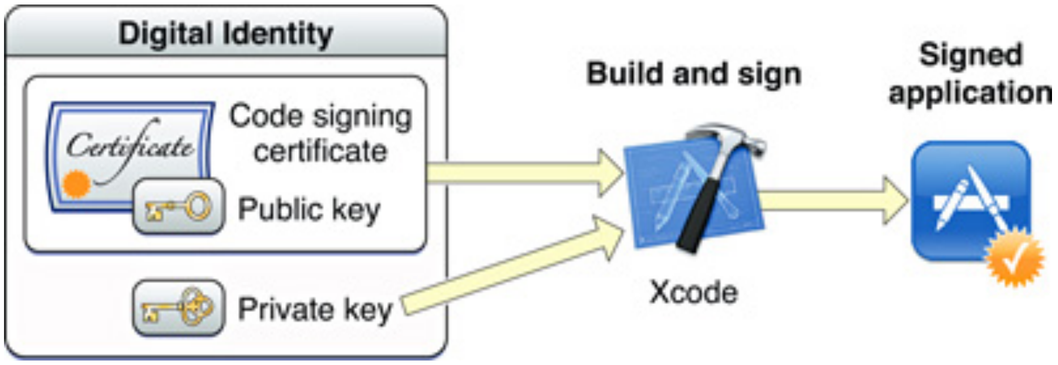
## Related Articles

## Definitive Discussion

App Distribution Guide

# Application Code Signing

Signing an application allows the system to identify who signed the application and to verify that the application has not been modified since it was signed. Signing is a requirement for submitting to the App Store (both for iOS and Mac apps). OS X and iOS verify the signature of applications downloaded from the App Store or the Mac App Store to ensure that they they do not run applications with invalid signatures. This lets users trust that the application was signed by an Apple source and hasn't been modified since it was signed.



Xcode uses your digital identity to sign your application during the build process. This digital identity consists of a public–private key pair and a certificate. The private key is used by cryptographic functions to generate the signature. The certificate is issued by Apple; it contains the public key and identifies you as the owner of the key pair.

In order to sign applications, you must have both parts of your digital identity installed. Use Xcode or Keychain Access to manage your digital identities. Depending on your role in your development team, you may have multiple digital identities for use in different contexts. For example, the identity you use for signing during development is different from the identity you user for distribution on the App Store or the Mac App Store. Different digital identities are also used for development on OS X and iOS.

An application's executable code is protected by its signature because the signature becomes invalid if any of the executable code in the app bundle changes.

An application's signature can be removed, and the application can be re-signed using another digital identity. For example, Apple re-signs all applications sold on the App Store and the Mac App Store. Also, a fully-tested development build of your application can be re-signed for submission to the App Store or the Mac App Store. Thus the signature is best understood not as indelible proof of the application's origins but as a verifiable mark placed by the signer.

### Prerequisite Articles

(None)

### Related Articles

App ID

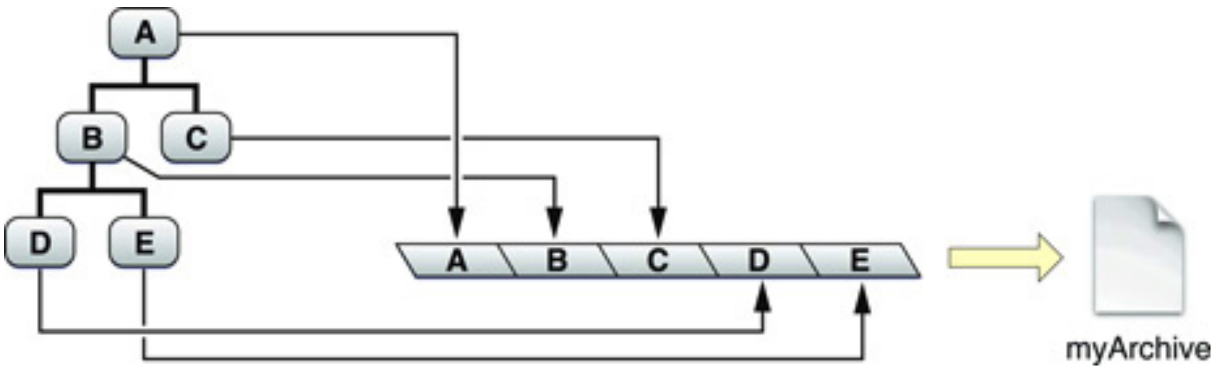
### Definitive Discussion

"Creating Your Team's Signing Certificates" in Application Development Process Overview

## Archiving

Archiving is the process of converting a group of related objects to a form that can be stored or transferred between applications. The end result of archiving—an archive—is a stream of bytes that records the identity of objects, their encapsulated values, and their relationships with other objects. Unarchiving, the reverse process, takes an archive and reconstitutes an identical network of objects.

The main value of archiving is that it provides a generic way to make objects persistent. Instead of writing object data out in a special file format, applications frequently store their model objects in archives that they can write out as files. An application can also transfer a network of objects—commonly known as an object graph—to another application using archiving. Applications often do this for pasteboard operations such as copy and paste.



For its instances to be included in an archive, a class must adopt the `NSCoding` protocol and implement the required methods for encoding and decoding objects. Cocoa archives can hold Objective-C objects, scalar values, C arrays, structures, and strings. Archives store the types of objects along with the encapsulated data, so an object decoded from a stream of bytes is of the same class as the object that was originally encoded into the stream.

## Keyed and Sequential Archivers

The Foundation framework provides two sets of classes for archiving and unarchiving networks of objects. They include methods for initiating the archiving and unarchiving processes and for encoding and decoding the instance data of your objects. Objects of these classes are sometimes referred to as archivers and unarchivers.

- Keyed archivers and unarchivers (`NSKeyedArchiver` and `NSKeyedUnarchiver`). These objects use string keys as identifiers of the data to be encoded and decoded. They are the preferred objects for archiving and unarchiving objects, especially with new applications.
- Sequential archivers and unarchivers (`NSArchiver` and `NSUnarchiver`). This “old-style” archiver encodes object state in a certain order; the unarchiver expects to decode object state in the same order. Their intended use is for legacy code; new applications should use keyed archives instead.

## Creating and Decoding Keyed Archives

An application creates an archive by invoking the `archiveRootObject:toFile:` class method of `NSKeyedArchiver`. The first parameter of this method takes a reference to the root object of an object graph. Starting with this root object, each object in the graph that conforms to the `NSCoding` protocol is given an opportunity to encode itself into the archive. The resulting byte stream is written to the specified file.

Decoding an archive proceeds in the opposite direction. An application calls the `NSKeyedUnarchiver` class method `unarchiveObjectWithFile:`. Given an archive file, the method recreates the object graph, asking the class of each object in the graph to decode the relevant data in the byte stream and recreate the object. The method ends by returning a reference to the root object.

The `NSKeyedArchiver` class methods `archivedDataWithRootObject:` and `unarchiveObjectWithData:` are equivalent to the above methods, except they work with a data object rather than a file.

### Prerequisite Articles

Object graph
Object encoding

### Related Articles

Model object
Property list
Object life cycle

### Definitive Discussion

Archives and Serializations Programming Guide

### Sample Code Projects

iSpend

## Block object

Block objects are a C-level syntactic and runtime feature that allow you to compose function expressions that can be passed as arguments, optionally stored, and used by multiple threads. The function expression can reference and can preserve access to local variables. In other languages and environments, a block object is sometimes also called a closure or a lambda. You use a block when you want to create units of work (that is, code segments) that can be passed around as though they are values. Blocks offer more flexible programming and more power. You might use them, for example, to write callbacks or to perform an operation on all the items in a collection.

## Declaring a Block

In many situations, you use blocks inline so you don't need to declare them. The declaration syntax, however, is similar to the standard syntax for function pointers, except that you use a caret (^) instead of an asterisk pointer (*). For example, the following declares a variable `aBlock` that references a block that requires three parameters and returns a `float` value:

```
float (^aBlock)(const int*, int, float);
```

## Creating a Block

You use the caret (^) operator to indicate the beginning, and a semicolon to indicate the end of a block expression. The following example declares a simple block and assigns it to a previously declared block variable (`oneFrom`):

```
int (^oneFrom)(int);



oneFrom = ^(int anInt) {

    return anInt - 1;

};
```

The closing semicolon is required as a standard C end-of-line marker.

If you don't explicitly declare the return value of a block expression, it can be automatically inferred from the contents of the block.

## Block-Mutable Variables

You can use the `__block` storage modifier with variables declared locally to the enclosing lexical scope to denote that such variables should be provided by reference in a block and so are mutable. Any changes are reflected in the enclosing lexical scope, including any other blocks defined within the same enclosing lexical scope.

## Using Blocks

If you declare a block as a variable, you can use it as you would a function. The following example prints 9 as output.

```
printf("%d\n", oneFrom(10));
```

Typically, however, you pass a block as the argument to a function or a method. In these cases, you often create a block inline.

The following example determines whether an `NSSet` object contains a word specified by a local variable and sets the value of another local variable (`found`) to `YES` (and stops the search) if it does. In this example, `found` is declared as a `__block` variable.

```
__block BOOL found = NO;


NSSet *aSet = [NSSet setWithObjects: @"Alpha", @"Beta",
@"Gamma", @"X", nil];


NSString *string = @"gamma";




[aSet enumerateObjectsUsingBlock:^(id obj, BOOL *stop) {


    if ([obj localizedCaseInsensitiveCompare:string] ==
NSOrderedSame) {


        *stop = YES;


        found = YES;


    }


}];
```

```
// At this point, found == YES
```

In this example, the block is contained within the method's argument list. The block also makes use of a stack local variable.

### Comparison Operations

One of the more common operations you perform with blocks in a Cocoa environment is comparison of two objects—to sort the contents of an array, for example. The Objective-C runtime defines a block type—`NSComparator`—to use for these comparisons.

### Prerequisite Articles

Message

### Related Articles

Enumeration

### Definitive Discussion

"Working with Blocks"

# Bundle

A bundle is a directory in the file system that groups executable code and related resources such as images and sounds together in one place. In iOS and OS X, applications, frameworks, plug-ins, and other types of software are bundles. A bundle is a directory with a standardized hierarchical structure that holds executable code and resources used by that code. Foundation and Core Foundation include facilities for locating and loading code and resources in bundles.

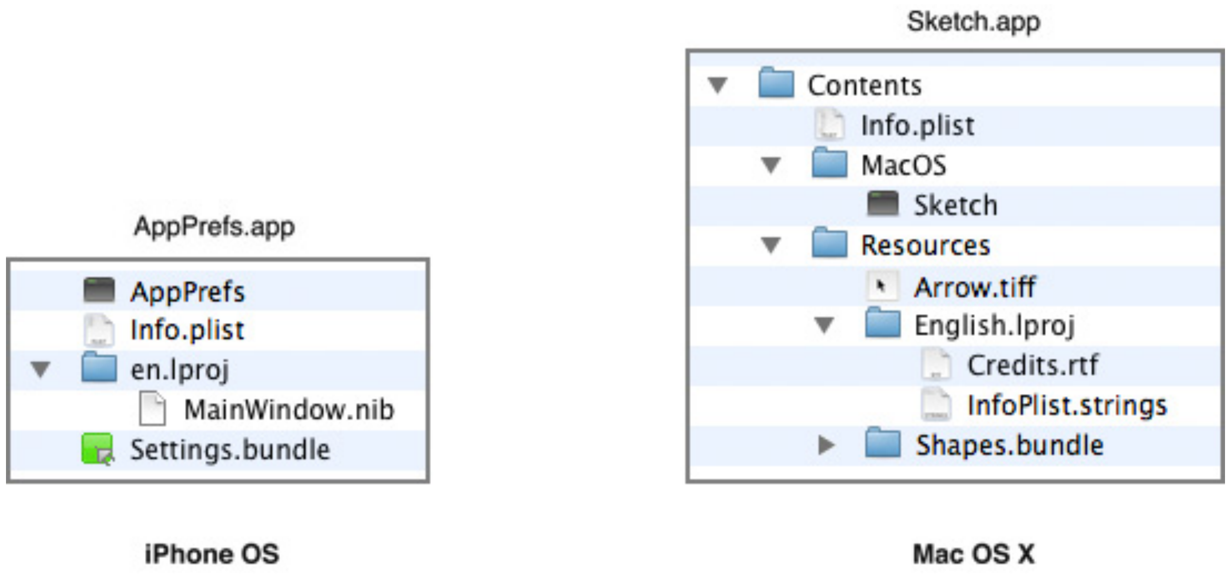
**Note:** Applications are the only type of bundle that third-party developers can create on iOS.

Bundles bring several advantages to users and developers. They make it easy to install or relocate an application or other piece of software by simply moving it from one location to another. Bundles are also an important factor in internationalization. You store localized resources in specially named subdirectories of a bundle; programmatic facilities look for localized resources in the location that corresponds with a user's language preferences.

Most types of Xcode projects create a bundle for you when you build the executable. Therefore you rarely need to construct a bundle by hand. Even so, it is important to understand their structure and how to access the code and resources inside them.

## Structure and Content of Bundles

A bundle can contain executable code, images, sounds, nib files, private frameworks and libraries, plug-ins, loadable bundles, or any other type of code or resource. It also contains a runtime-configuration file called the information property list (`Info.plist`). Each of these items has its proper place in the bundle structure. Resources such as images, sounds, and nib files are deposited in the `Resources` subdirectory. They can be either localized or nonlocalized. Localized files (including strings files, which are collections of localized strings) are put in subdirectories of `Resources` that have the extension of `lproj` and a name corresponding to a language and possibly a locale.



## Accessing Bundle Resources

Each application has a main bundle, which is the bundle that contains the application code. When a user launches an application, it finds the code and resources in the main bundle that it immediately needs and loads them into memory. Thereafter, the application can dynamically (and lazily) load code and resources from the main bundle or subordinate bundles as required.

The `NSBundle` class and, for procedural code, the `CFBundleRef` opaque type of Core Foundation give your application the means to locate resources in a bundle. In Objective-C, you first must obtain an instance of `NSBundle` that corresponds to the physical bundle. To get an application's main bundle, call the class method `mainBundle`. Other `NSBundle` methods return paths to bundle resources when given a filename, extension, and (optionally) a bundle subdirectory. After you have a path to a resource, you can load it into memory using the appropriate class.

### Loadable Bundles

As with application bundles, loadable bundles package executable code and related resources, but you explicitly load these bundles at runtime. You can use loadable bundles to design applications that are highly modular, customizable, and extensible. Every loadable bundle has a principal class that is the entry point for the bundle; when you load the bundle, you must ask `NSBundle` for the principal class and use the returned `Class` object to create an instance of the class.

### Prerequisite Articles

Message

### Related Articles

Property list
Nib file

### Definitive Discussion

"About Bundles"

## Category

You use categories to define additional methods of an existing class—even one whose source code is unavailable to you—without subclassing. You typically use a category to add methods to an existing class, such as one defined in the Cocoa frameworks. The added methods are inherited by subclasses and are indistinguishable at runtime from the original methods of the class. You can also use categories of your own classes to:

- Distribute the implementation of your own classes into separate source files—for example, you could group the methods of a large class into several categories and put each category in a different

file.
- Declare private methods.

You add methods to a class by declaring them in an interface file under a category name and defining them in an implementation file under the same name. The category name indicates that the methods are an extension to a class declared elsewhere, not a new class.

## Declaration

The declaration of a category interface looks very much like a class interface declaration—except that the category name is listed within parentheses after the class name and the superclass isn't mentioned. A category must import the interface file for the class it extends:

```
#import "SystemClass.h"




@interface SystemClass (CategoryName)


// method declarations


@end
```

A common naming convention is that the base file name of the category is the name of the class the category extends followed by "+" followed by the name of the category. This category might be declared in a file named `SystemClass+CategoryName.h`.

If you use a category to declare private methods of one of your own classes, you can put the declaration in the implementation file before the `@implementation` block:

```
#import "MyClass.h"




@interface MyClass (PrivateMethods)
```

```
// method declarations

@end



@implementation MyClass

// method definitions

@end
```

## Implementation

If you use a category to declare private methods of one of your own classes, you can put the implementation in your class's `@implementation` block. If you use a category to extend a class to which you don't have source code, or to distribute the implementation of your own class, you put the implementation in a file named `<ClassName>+CategoryName.m` The implementation, as usual, imports its own interface. A category implementation might therefore look like this:

```
#import "SystemClass+CategoryName.h"



@implementation SystemClass ( CategoryName )

// method definitions

@end
```

## Prerequisite Articles

Class definition

## Related Articles

(None)

### Definitive Discussion

“Categories Add Methods to Existing Classes” in Programming with Objective-C

## Class cluster

A class cluster is an architecture that groups a number of private, concrete subclasses under a public, abstract superclass. The grouping of classes in this way provides a simplified interface to the user, who sees only the publicly visible architecture. Behind the scenes, though, the abstract class is calling up the private subclass most suited for performing a particular task. For example, several of the common Cocoa classes are implemented as class clusters, including `NSArray`, `NSString`, and `NSDictionary`. There are many ways in which they can represent their internal data storage. For any particular instance, the abstract class chooses the most efficient class to use based on the data that instance is being initialized with.

You create and interact with instances of the cluster just as you would any other class. Behind the scenes, though, when you create an instance of the public class, the class returns an object of the appropriate subclass based on the creation method that you invoke. (You don’t, and can’t, choose the actual class of the instance.)

Taking the Foundation framework’s `NSString` class as an example, you could create three different string objects:

```
NSString *string1 = @"UTF32.txt";


NSString *string2 = [NSHomeDirectory()
stringByAppendingPathComponent:string1];


NSTextStorage *storage = [[NSTextStorage alloc]
initWithString:string2];


NSString *string3 = [storage string];
```

Each string may be an instance of a different private subclass (and in fact, on OS X v10.5, each is). Although each of the objects is of a private subclass of `NSString`, it's convenient to consider each of the objects to be instances of the `NSString` class. You use the instance methods declared by `NSString` just as you would if they were instances of `NSString` itself.

## Benefits

The benefit of a class cluster is primarily efficiency. The internal representation of the data that an instance manages can be tailored to the way it's created or being used. Moreover, the code you write will continue to work even if the underlying implementation changes.

## Considerations

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.

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 the superclass's primitive methods

If it's rarely necessary to create a subclass—as is the case with class clusters in the Foundation framework—then the cluster architecture is clearly beneficial. You might also be able to avoid subclassing by using composition; 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 it wants to handle in some particular way. Using this approach reduces the amount of code you must write and lets you take advantage of the tested code provided by the Foundation Framework.

## Prerequisite Articles

(None)

## Related Articles

(None)

## Definitive Discussion

"Class Clusters" in Cocoa Fundamentals Guide

## Class definition

A class definition is the specification of a class of objects through the use of certain files and syntax. A class definition minimally consists of two parts: a public interface, and a private implementation. You typically split the interface and implementation into two separate files—the header file and the implementation file. By separating the public and private parts of your code, you retain the class interface as an independent entity.

You usually name the interface and implementation files after the class. Because it's included in other source files, the name of the interface file usually has the `.h` extension typical of header files. The name of the implementation file has a `.m` extension, indicating that it contains Objective-C source code. For example, the `MyClass` class would be declared in `MyClass.h` and defined in `MyClass.m`.

### Interface

In the interface, you do several things:

- You name the class and its superclass.You may also specify any protocols that your class conforms to (see Protocol).
- You specify the class's instance variables.
- You specify the methods and declared properties (see Declared property) that are available for the class.

In the interface file, you first import any required frameworks. (This will often be just `Cocoa/Cocoa.h`.) You start the declaration of the class interface itself with the compiler directive `@interface` and finish it with the directive `@end`.

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


@interface MyClass : SuperClass {


    int integerInstanceVariable;


}
```

```
+ (void)aClassMethod;


- (void)anInstanceMethod;



@end
```

## Implementation

Whereas you declare a class's methods in the interface, you define those methods (that is, write the code for implementing them) in the implementation.

In the interface file, you first import any required header files. (Minimally this will be your class's header file.) You start the implementation of the class with the compiler directive `@implementation` and finish it with the directive `@end`.

```
#import "MyClass.h"



@implementation MyClass

+ (void)aClassMethod {


    printf("This is a class method\n");


}

- (void)anInstanceMethod {


    printf("This is an instance method\n");


    printf("The value of integerInstanceVariable is %d\n",
integerInstanceVariable);


}

@end
```

## Prerequisite Articles

(None)

### Related Articles

Category
Accessor method

### Definitive Discussion

“Defining Classes”

## Class method

A class method is a method that operates on class objects rather than instances of the class. In Objective-C, a class method is denoted by a plus (+) sign at the beginning of the method declaration and implementation:

```
+ (void)classMethod;
```

To send a message to a class, you put the name of the class as the receiver in the message expression:

```
[MyClass classMethod];
```

### Subclasses

You can send class messages to subclasses of the class that declared the method. For example, `NSArray` declares the class method `array` that returns a new instance of an array object. You can also use the method with `NSMutableArray`, which is a subclass of `NSArray`:

```
NSMutableArray *aMutableArray = [NSMutableArray array];
```

In this case, the new object is an instance of `NSMutableArray`, not `NSArray`.

### Instance Variables

Class methods can’t refer directly to instance variables. For example, given the following class declaration:

```
@interface MyClass : NSObject {
    NSString *title;

}

+ (void)classMethod;

@end
```

you cannot refer to `title` within the implementation of `classMethod`.

## self

Within the body of a class method, `self` refers to the class object itself. You might implement a factory method like this:

```
+ (id)myClass {

    return [[[self alloc] init] autorelease];

}
```

In this method, `self` refers to the class to which the message was sent. If you created a subclass of `MyClass`:

```
@interface MySubClass : MyClass {

}

@end
```

and then sent a `myClass` message to the subclass:

```
id instance = [MySubClass myClass];
```

at runtime, within the body of the `myClass` method, `self` would refer to the `MySubClass` class (and so the method would return an instance of the subclass).

## Prerequisite Articles

(None)

### Related Articles

Accessor method

### Definitive Discussion

“Defining Classes”

## Cocoa (Touch)

Cocoa and Cocoa Touch are the application development environments for OS X and iOS, respectively. Both Cocoa and Cocoa Touch include the Objective-C runtime and two core frameworks:

- **Cocoa**, which includes the Foundation and AppKit frameworks, is used for developing applications that run on OS X.
- **Cocoa Touch**, which includes Foundation and UIKit frameworks, is used for developing applications that run on iOS.

**Note:** The term “Cocoa” has been used to refer generically to any class or object that is based on the Objective-C runtime and inherits from the root class, `NSObject`. The terms “Cocoa” or “Cocoa Touch” are also used when referring to application development using any programmatic interface of the respective platforms.

### The Frameworks

The Foundation framework implements the root class, `NSObject`, which defines basic object behavior. It implements classes that represent primitive types (for example, strings and numbers) and collections (for example, arrays and dictionaries). Foundation also provides facilities for internationalization, object persistence, file management, and XML processing. You can use its classes to access underlying system entities and services, such as ports, threads, locks, and processes. Foundation is based on the Core Foundation framework, which publishes a procedural (ANSI C) interface.

You use the AppKit and UIKit frameworks for developing an application’s user interface. These two frameworks are equivalent in purpose but are specific to a platform. They include classes for event handling, drawing, image-handling, text processing, typography, and interapplication data transfer. They also include user-interface elements such as table views, sliders, buttons, text fields, and alert dialogs.

### The Language

Objective-C is the native, primary language for developing Cocoa and Cocoa Touch applications. However, projects for Cocoa and Cocoa Touch applications may include C++ and ANSI C code. Additionally, you can develop Cocoa applications using scripting languages that are bridged to the Objective-C runtime, such as PyObjC and RubyCocoa.

### Prerequisite Articles

(None)

### Related Articles

Root class
Objective-C

### Definitive Discussion

(None)

## Coding conventions

Coding conventions are a set of guidelines that help to ensure efficiency and consistency in API usage and clarity and consistency in API naming. If you following the usage conventions in your code, you are less likely to experience problems such as runtime exceptions. If you adhere to the naming conventions, any methods, functions, constants, or other symbols that you declare will be better understood by developers who need to work with your code.

The methods defined in Cocoa frameworks—for example, Foundation, AppKit, and UIKit—behave in certain ways in different circumstances. For example:

- Methods that return objects typically return `nil` if they cannot create or find the object. They do not directly return a status code.
- Methods that perform an operation return a Boolean value to indicate success or failure.
- If a method takes a collection object—that is, an `NSArray`, `NSDictionary`, or `NSSet` object—as an argument, do not specify `nil` to indicate “default” or “no value”; instead, pass in an empty collection object.

- If you are explicitly managing program memory, follow the guidelines and practices for memory management.

Here is a sampling of API naming conventions:

- Clarity and brevity are both important, but clarity should never be sacrificed for brevity.
- Avoid names that are ambiguous.
- Use verbs in the names of methods or functions that represent actions.
- Use prefixes for class names and for symbols associated with the class, such as functions and data types.

### Prerequisite Articles

(None)

### Related Articles

Memory management

### Definitive Discussion

Coding Guidelines for Cocoa

# Collection

A collection is a Foundation framework object whose primary role is to store objects in the form of arrays, dictionaries, and sets.

### Collection Classes

The primary classes—`NSArray`, `NSSet`, and `NSDictionary`—share a number of features in common:

- They can hold only objects, but the objects can be of any type. An instance of `NSArray`, for example, could contain cats, dogs, or wombats, or any combination of these.
- They maintain strong references to their contents.
- They are immutable, but have a mutable subclass that allows you to change the contents of the collection.
- You can iterate over their contents using `NSEnumerator` or fast enumeration.

Cocoa also provides three classes—`NSPointerArray`, `NSHashTable`, and `NSMapTable`—that are modeled on these classes but that differ in the following ways:

- They may contain elements other than objects.
- They offer other memory management options.
- They are mutable.

Since a Cocoa collection object can hold any sort of object (unlike collections in some other environments), you typically don't create special collection classes to contain objects of a particular type.

## Ordering Schemes

Collections store and vend other objects in a particular ordering scheme:

- `NSArray` and its mutable subclass `NSMutableArray` use zero-based indexing.In other environments, an array may be called a vector, table, or list.
  `NSPointerArray` is modeled after `NSMutableArray`, but it can also hold `NULL` values (which contribute to the object's count). You can also set the count of the pointer array directly (something you can't do in a traditional array).
- `NSDictionary` and its mutable subclass `NSMutableDictionary` use key-value pairs.In other environments, a dictionary may be referred to as a hash table or hash map.
  `NSMapTable` is modeled after `NSMutableDictionary` but provides different options, in particular to support weak relationships in a garbage-collected environment.
- `NSSet` and its mutable subclass `NSMutableSet` provide unordered storage of objects.Cocoa also provides `NSCountedSet`, which is a subclass of `NSMutableSet` and which keeps a count of how many times each object has been added to the set.
  `NSHashTable` is modeled after `NSMutableSet` but provides different options, mostly to support weak relationships in a garbage-collected environment.

## Prerequisite Articles

(None)

## Related Articles

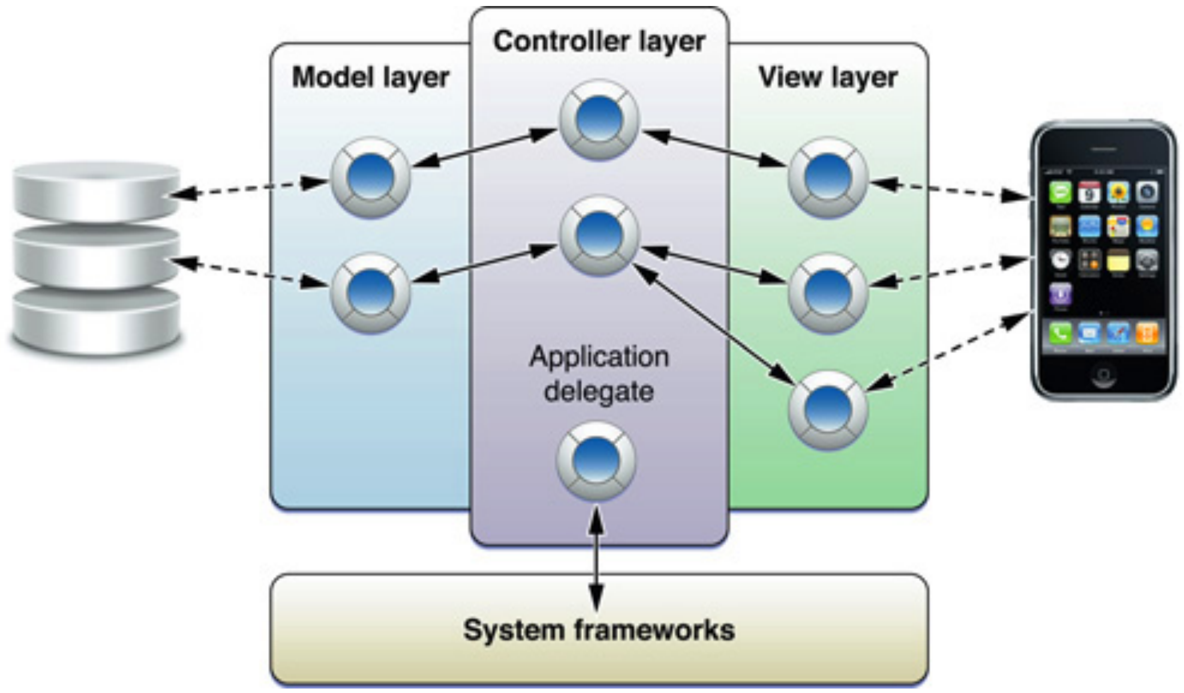
Enumeration

## Definitive Discussion

Collections Programming Topics

# Controller object

A controller object acts as a coordinator or as an intermediary between one or more view objects and one or more model objects. In the Model–View–Controller design pattern, a controller object (or, simply, a controller) interprets user actions and intentions made in view objects—such as when the user taps or clicks a button or enters text in a text field—and communicates new or changed data to the model objects. When model objects change—for example, the user opens a document stored in the file system—it communicates that new model data to the view objects so that they can display it. Controllers are thus the conduit through which view objects learn about changes in model objects and vice versa. Controller objects can also set up and coordinate tasks for an application and manage the life cycles of other objects. The Cocoa frameworks offer three main controller types: coordinating controllers, view controllers (on iOS), and mediating controllers (on OS X).



## Coordinating Controllers

Coordinating controllers oversee and manage the functioning of an entire application, or part of one. They are often the places where application-specific logic is injected into the application. A coordinating controller fulfills a variety of functions, including:

- Responding to delegation messages and observing notifications

- Responding to action messages (which are are sent by controls such as buttons when users tap or click them)
- Establishing connections between objects and performing other setup tasks, such as when the application launches
- Managing the life cycle of “owned” objects

A coordinating controller is often an instance of a custom subclass of `NSObject`. In OS X, if a Cocoa application takes advantage of the document architecture, the coordinating controller is often an `NSWindowController` or `NSDocumentController` object. In iOS applications, view controllers often subsume the role of coordinating controller.

### View Controllers

The UIKit and AppKit frameworks both provide view controller classes (for iOS and OS X, respectively) but these classes have different characteristics. In AppKit, a view controller is an instance of a custom subclass of the `NSViewController` class. The view controller owns a view archived in a nib file, and that view represents a data object. The view controller manages connections and updates to subviews of its view.

In UIKit, a view controller manages a view displaying a screenful of content; it keeps a reference to this view and may create or load it from a nib file. The controller manages the presentation of this view and the transition to any subsequent view in the app. (In most cases, the next view slides in from the right.) The navigation bar and the tab bar, and all their associated presentation behavior, are managed and implemented by view controller objects. View controllers can also display modal views, respond to low-memory warnings, and rotate views when the orientation changes.

A view controller in iOS is an instance of a subclass of `UIViewController`. The UIKit provides several special-purpose subclasses of `UIViewController`, such as `UITableViewController`. You must extend the framework view-controller classes to have the controller mediate the data between models and views. View controllers are typically the delegate or data source objects for many types of framework objects.

### Mediating Controllers (OS X)

A mediating controller facilitates the flow of data between view objects and model objects. When users change a value displayed in a view object, the mediating controller automatically communicates the new value to a model object for storage; and when a property of a model changes its value, the mediating controller ensures that the appropriate view object displays the changed value. Unlike other types of controller objects, they are highly reusable. For these and other reasons, mediating controllers are a central component of the Cocoa bindings technology. You drag a mediating controller from the Interface Builder library and then configure these objects to establish bindings between the controller and its view objects and its model objects. A mediating controller is typically an instance of a concrete subclass of the abstract `NSController` class.

### Prerequisite Articles

Model-View-Controller
Message

### Related Articles

Model object
Delegation
Notification

### Definitive Discussion

"Model-View-Controller" in Cocoa Fundamentals Guide

## Declared property

A declared property provides a syntactical shorthand for declaring a class's accessor methods and, optionally, implementing them. You can declare a property anywhere in the method declaration list, which is in the interface of a class, or in the declaration of a protocol or category. You use the following syntax:

```
@property (<#attributes#>) <#type#> <#name#>;
```

You begin a property declaration with the keyword `@property`. You can then optionally provide a parenthesized set of property attributes that define the storage semantics and other behaviors of the property. (Refer to the document that definitively describes property lists for descriptions of these attributes.)

Each property declaration ends with a type specification and a name. For example:

```
@property(copy) NSString *title;
```

This syntax is equivalent to declaring the following accessor methods:

```
- (NSString *)title;

- (void)setTitle:(NSString *)newTitle;
```

In addition to declaring the accessor methods, you can instruct the compiler to synthesize implementations of them (or inform the compiler that your class will synthesize them at runtime).

You use the `@synthesize` statement in a class's implementation block to tell the compiler to create implementations that match the specification you gave in the property declaration.

```
@interface MyClass : NSObject

{

    NSString *title;

}

@property(copy) NSString *title;

@end



@implementation MyClass

@synthesize title;

@end
```

You use the `@dynamic` statement to tell the compiler to suppress a warning if it can't find an implementation of accessor methods specified by an `@property` declaration.

```
@implementation MyClass

@dynamic title;

@end
```

### Prerequisite Articles

Accessor method

### Related Articles

(None)

### Definitive Discussion

"Properties Encapsulate an Object's Values" in Programming with Objective-C

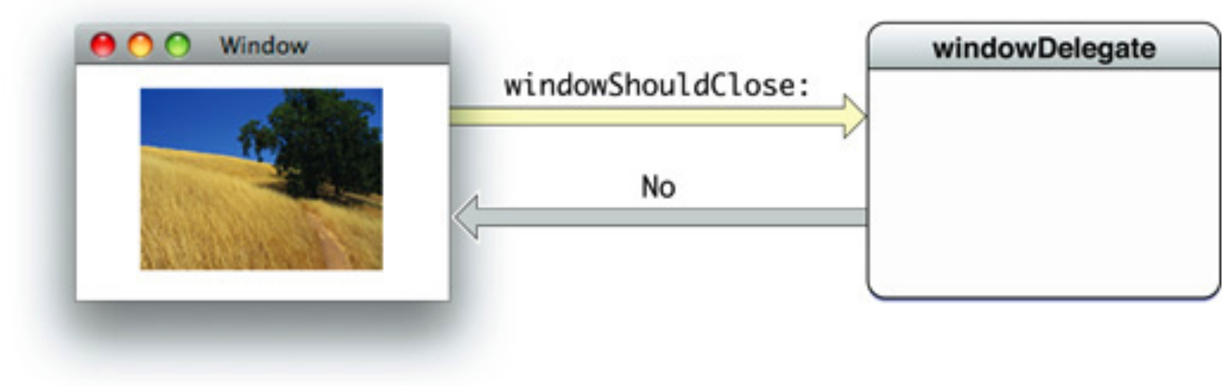
# Delegation

Delegation is a simple and powerful pattern in which one object in a program acts on behalf of, or in coordination with, another object. The delegating object keeps a reference to the other object—the delegate—and at the appropriate time sends a message to it. The message informs the delegate of an event that the delegating object is about to handle or has just handled. The delegate may respond to the message by updating the appearance or state of itself or other objects in the application, and in some cases it can return a value that affects how an impending event is handled. The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object.

## Delegation and the Cocoa Frameworks

The delegating object is typically a framework object, and the delegate is typically a custom controller object. In a managed memory environment, the delegating object maintains a weak reference to its delegate; in a garbage-collected environment, the receiver maintains a strong reference to its delegate. Examples of delegation abound in the Foundation, UIKit, AppKit, and other Cocoa and Cocoa Touch frameworks.

An example of a delegating object is an instance of the `NSWindow` class of the AppKit framework. `NSWindow` declares a protocol, among whose methods is `windowShouldClose:`. When a user clicks the close box in a window, the window object sends `windowShouldClose:` to its delegate to ask it to confirm the closure of the window. The delegate returns a Boolean value, thereby controlling the behavior of the window object.



## Delegation and Notifications

The delegate of most Cocoa framework classes is automatically registered as an observer of notifications posted by the delegating object. The delegate need only implement a notification method declared by the framework class to receive a particular notification message. Following the example above, a window object posts an `NSWindowWillCloseNotification` to observers but sends a `windowShouldClose:` message to its delegate.

## Data Source

A data source is almost identical to a delegate. The difference is in the relationship with the delegating object. Instead of being delegated control of the user interface, a data source is delegated control of data. The delegating object, typically a view object such as a table view, holds a reference to its data source and occasionally asks it for the data it should display. A data source, like a delegate, must adopt a protocol and implement at minimum the required methods of that protocol. Data sources are responsible for managing the memory of the model objects they give to the delegating view.

## Prerequisite Articles

Class definition

### Related Articles

Notification
Protocol
Controller object

### Definitive Discussion

"Delegates and Data Sources" in Cocoa Fundamentals Guide

### Sample Code Projects

TableView Fundamentals for iOS

## Dynamic binding

Dynamic binding is determining the method to invoke at runtime instead of at compile time. Dynamic binding is also referred to as late binding. In Objective-C, all methods are resolved dynamically at runtime. The exact code executed is determined by both the method name (the selector) and the receiving object.

Dynamic binding enables polymorphism. For example, consider a collection of objects including `Dog`, `Athlete`, and `ComputerSimulation`. Each object has its own implementation of a `run` method. In the following code fragment, the actual code that should be executed by the expression `[anObject run]` is determined at runtime. The runtime system uses the selector for the method `run` to identify the appropriate method in whatever the class of `anObject` turns out to be.

```
NSArray *anArray = [NSArray arrayWithObjects:aDog, anAthlete,
aComputerSimulation, nil];


id anObject = [anArray objectAtIndex:(random()/pow(2, 31)*3)];


[anObject run];
```

The example illustrates how dynamic Objective-C is—this feature is used pervasively in Cocoa.

### Prerequisite Articles

Key-value observing
Selector

### Related Articles

(None)

### Definitive Discussion

“Working with Objects”

## Dynamic typing

A variable is dynamically typed when the type of the object it points to is not checked at compile time. Objective-C uses the `id` data type to represent a variable that is an object without specifying what sort of object it is. This is referred to as dynamic typing.

Dynamic typing contrasts with static typing, in which the system explicitly identifies the class to which an object belongs at compile time. Static type checking at compile time may ensure stricter data integrity, but in exchange for that integrity, dynamic typing gives your program much greater flexibility. And through object introspection (for example, asking a dynamically typed, anonymous object what its class is), you can still verify the type of an object at runtime and thus validate its suitability for a particular operation.

The following example illustrates dynamic typing using a heterogeneous collection of objects:

```
NSArray *anArray = [NSArray arrayWithObjects:@"A string",
[NSDecimalNumber zero], [NSDate date], nil];


NSInteger index;


for (index = 0; index < 3; index++) {


    id anObject = [anArray objectAtIndex:index];


    NSLog(@"Object at index %d is %@", index, [anObject
description]);
}
```

The object pointed to by the variable at runtime must be able to respond to whatever messages you send to it; otherwise, your program throws an exception. The actual implementation of the method invoked is determined using dynamic binding.

### The isa Pointer

Every object has an `isa` instance variable that identifies the object's class. The runtime uses this pointer to determine the actual class of the object when it needs to.

### Prerequisite Articles

(None)

### Related Articles

Dynamic binding
Exception handling

### Definitive Discussion

"Working with Objects"

## Enumeration

Enumeration is the process of sequentially operating on elements of an object—typically a collection—each at most once, one at a time in turn. Enumeration is also referred to as iteration. When you enumerate an object, you typically select one element within the object at a time and perform an operation on it or with it. The object is usually a collection such as an array or set. In the conceptually simplest case, you can use a standard C `for` loop to enumerate the contents of an array, as shown in the following example:

```
NSArray *array = // get an array;

NSInteger i, count = [array count];

for (i = 0; i < count; i++) {
    id element = [array objectAtIndex:i];

    /* code that acts on the element */

}
```

Using a for loop, however, can be inefficient, and requires an ordered collection of elements. Enumeration is more general. Cocoa therefore provides two additional means to enumerate objects—the `NSEnumerator` class and fast enumeration.

### NSEnumerator

Several Cocoa classes, including the collection classes, can be asked to provide an enumerator so that you can retrieve elements held by an instance in turn. For example:

```
NSSet *aSet = // get a set;

NSEnumerator *enumerator = [aSet objectEnumerator];

id element;



while ((element = [enumerator nextObject])) {

    /* code that acts on the element */

}
```

In general, however, use of the `NSEnumerator` class is superseded by fast enumeration.

### Fast Enumeration

Several Cocoa classes, including the collection classes, adopt the `NSFastEnumeration` protocol. You use it to retrieve elements held by an instance using a syntax similar to that of a standard C `for` loop, as illustrated in the following example:

```
NSArray *anArray = // get an array;

for (id element in anArray) {

    /* code that acts on the element */

}
```

As the name suggests, fast enumeration is more efficient than other forms of enumeration.

### Prerequisite Articles

Collection

### Related Articles

Block object

### Definitive Discussion

"Use the Most Efficient Collection Enumeration Techniques" in Programming with Objective-C

## Exception handling

Exception handling is the process of managing atypical events (such as unrecognized messages) that interrupt the normal flow of program execution. Without adequate error handling, a program that encounters an atypical event will most likely terminate by immediately throwing (or raising) what's known as an exception.

### Types of Exception

There are a variety of reasons why an exception may be thrown, by hardware as well as software. Examples include arithmetical errors such as division by zero, underflow or overflow, calling undefined instructions (such as attempting to invoke an unimplemented method), and attempting to access a collection element out of bounds.

### Handling Exceptions Using Compiler Directives

There are four compiler directives to support exception handling:

- A `@try` block encloses code that can potentially throw an exception.
- A `@catch()` block contains exception-handling logic for exceptions thrown in a `@try` block. You can have multiple `@catch()` blocks to catch different types of exception.
- A `@finally` block contains code that must be executed whether an exception is thrown or not.
- A `@throw` directive raises an exception, which is essentially an Objective-C object. You typically use an `NSException` object, but are not required to.

This example shows how you use the directives when executing code that might raise an exception:

```
Cup *cup = [[Cup alloc] init];




@try {


    [cup fill];


}


@catch (NSException *exception) {


    NSLog(@"main: Caught %@: %@", [exception name], [exception
reason]);


}


@finally {


    [cup release];


}
```

### Signaling Errors

Although exceptions are commonly used in many programming environments to control programming flow or to signify errors, do not use exceptions in this way in Cocoa and Cocoa Touch applications. Instead, you should use the return value of a method or function to indicate that an error has occurred, and provide information about the problem in an error object. For more information, see Error Handling Programming Guide.

### Prerequisite Articles

(None)

### Related Articles

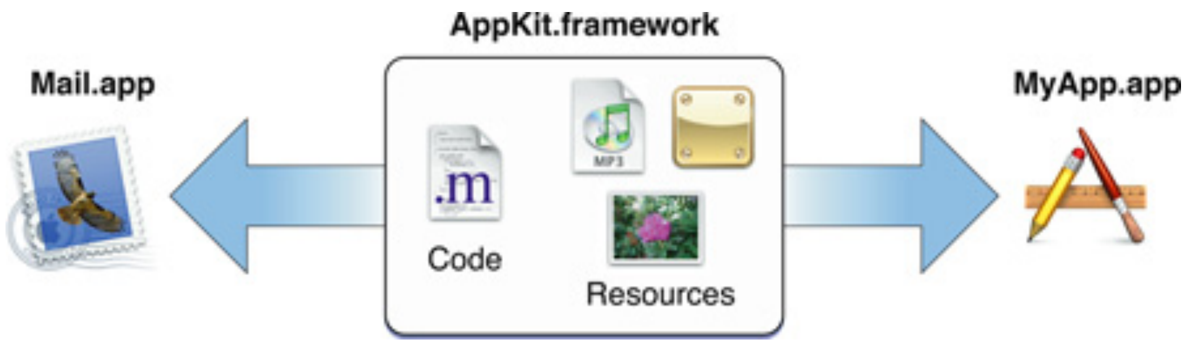
(None)

### Definitive Discussion

Exception Programming Topics

## Framework

A framework is a bundle (a structured directory) that contains a dynamic shared library along with associated resources, such as nib files, image files, and header files. When you develop an application, your project links to one or more frameworks. For example, iPhone application projects link by default to the Foundation, UIKit, and Core Graphics frameworks. Your code accesses the capabilities of a framework through the application programming interface (API), which is published by the framework through its header files. Because the library is dynamically shared, multiple applications can access the framework code and resources simultaneously. The system loads the code and resources of a framework into memory, as needed, and shares the one copy of a resource among all applications.



Because a framework is a bundle, you may access its contents using the NSBundle class or, for procedural code, CFBundle of Core Foundation. You may create your own frameworks for OS X, but third-party frameworks are not allowed on iOS. On OS X, you may browse the contents of a framework in the Finder. When developing for either platform, you may also view the header files of a framework from within the Xcode application.

**Note:** The chapter referenced as the Definitive Discussion is part of a document that is specific to OS X only. If you are using the iOS Developer Reference Library, this link does not work.

### Prerequisite Articles

Bundle

### Related Articles

(None)

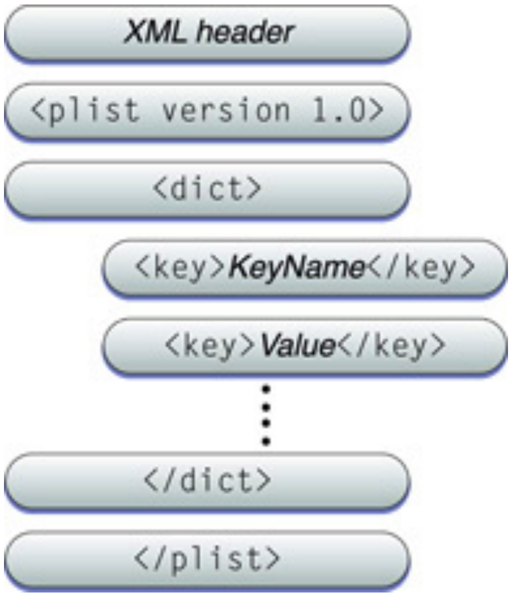
### Definitive Discussion

Framework Programming Guide

## Information property list

An information property list is structured text specifying configuration details for an application or other bundled executable. The operating system extracts data from an information property list at runtime and processes it in suitable ways. For example, when they list applications in the Home screen or in the Finder, iOS and OS X (respectively) get these names from the information property lists of installed applications.

The contents of an information property list are structured in a special form of XML where the root node is always a dictionary. The dictionary contains a series of key-value pairs, or properties, where the key is a `key` element and the value is an element that indicates the data type of the value.



Here is a sample key-value pair:

```
<key>CFBundleDisplayName</key>


<string>Mail</string>
```

Properties specified in an information property list include display name, bundle identifier, bundle icon, bundle version, supported platforms, and document types.

The name of an information property list file must be `Info.plist`, with letters in lowercase or uppercase as shown. The text in the file is encoded in Unicode UTF-8. When you build an application or other bundle, the file is put in a certain location in the bundle.

When you use Xcode (the primary development application) to create a project for an application or other bundle, Xcode creates a file named ProjectName-`Info.plist` in your project's Resources folder. (When you build your project, the tools copy this file to the bundle as `Info.plist`.) Xcode configures some properties in ProjectName-`Info.plist` for you, but you often have to specify additional ones. You may edit the information property list in Xcode's editor by selecting the file directly. You can also edit some properties in the Properties pane of the target inspector.

### Prerequisite Articles

Property list
Bundle

### Related Articles

(None)

### Definitive Discussion

Information Property List Key Reference

## Initialization

Initialization is the stage of object creation that makes a newly allocated object usable by setting its state to reasonable initial values. Initialization should always occur right after allocation. It is performed by an initializer method (or simply, an initializer), which you always invoke on a newly allocated object. Initializers can also perform other setup tasks that bring the object into a useful state, such as loading resources and allocating heap memory.

### The Form of an Initializer Declaration

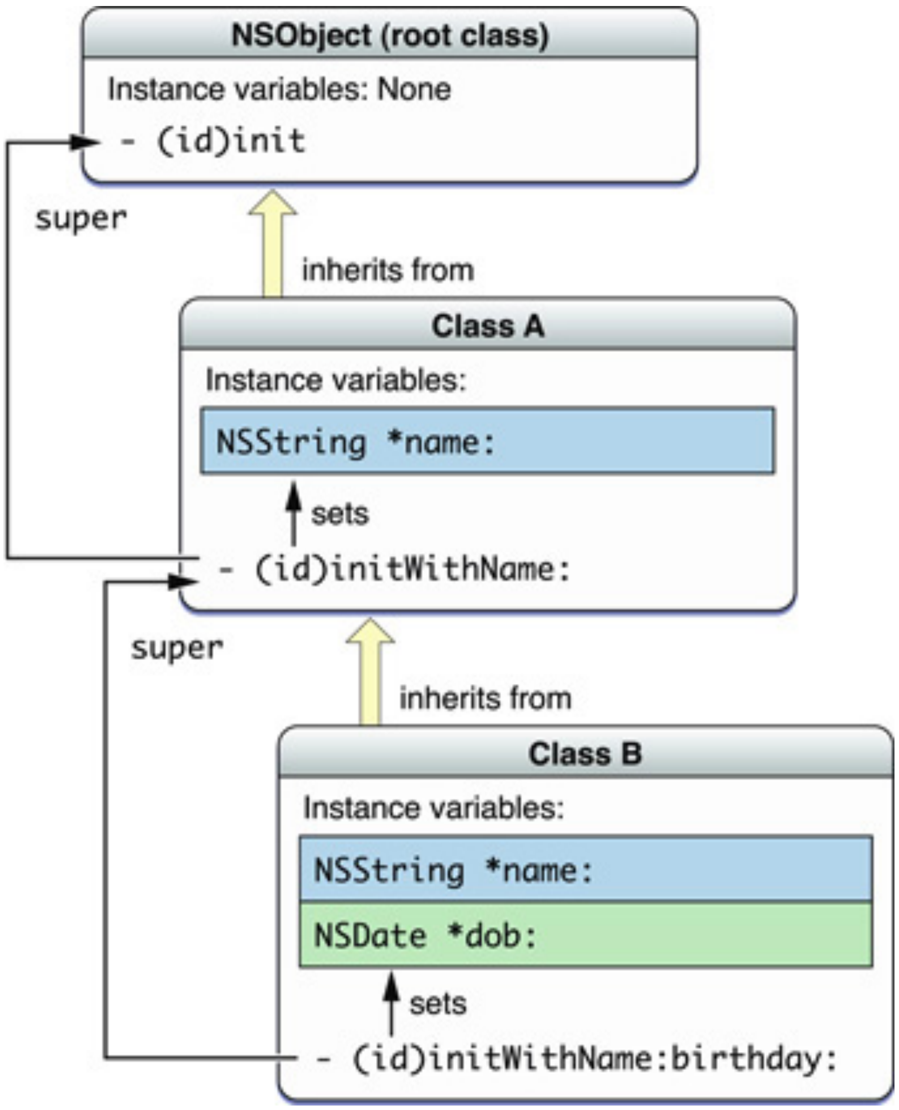
By convention, the name of an initializer always begins with `init`. It returns a dynamically typed object (`id`) or, if initialization does not succeed, `nil`. An initializer can include one or more parameters that specify initial values.

Here is a sample declaration of an initializer from the `NSString` class:

```
- (id)initWithData:(NSData *)data encoding:
(NSStringEncoding)encoding
```

### Implementing an Initializer

A class generally implements an initializer for its objects, but is not required to. If a class does not implement an initializer, Cocoa calls the initializer of the nearest ancestor of the class. However, subclasses often define their own initializer or override an initializer of their superclass to add class-specific initializations. If a class does implement an initializer, it should invoke an initializer of its superclass as the first step. This requirement ensures a series of initializations for an object down the inheritance chain, starting with the root object. The `NSObject` class declares the `init` method as the default object initializer, so it is always invoked last but returns first.



The basic steps for implementing an initializer method are the following:

1. **Invoke the superclass initializer and check the value it returns.** (Use the reserved word `super` to designate the superclass.) If the value is not `nil`, the superclass initializer has returned a valid object, so you may proceed with initialization.
2. **Assign values to the object's instance variables.** In memory-managed code, if those values are objects themselves, copy or retain them, as appropriate.
3. **Return the initialized object** or, if initialization did not succeed,

return `nil`.

Here is a simple initializer that follows these steps, initializing its `date` instance variable to the current date:

```
- (id)init {

    if (self = [super init]) { // equivalent to "self does not equal nil"

        date = [[NSDate date] retain];

    }

    return self;

}
```

In this code, if the superclass returns `nil`, the method skips initialization and returns that value to its caller.

A class may have multiple initializers. This occurs when the initialization data can take varied forms or where certain initializers, as a matter of convenience, supply default values. In this case, one of the initialization methods is called the designated initializer, which takes the full complement of initialization parameters.

## Prerequisite Articles

- Object creation
- Message

## Related Articles

- Multiple initializers
- Object copying
- Memory management

## Definitive Discussion

- "Objects Are Created Dynamically" in Programming with Objective-C

# Internationalization

Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization, in turn, is the cultural and linguistic adaptation of an internationalized application to two or more culturally distinct markets. Internationalization and localization can be seen, then, as complementary activities.

Cocoa provides a rich architecture for internationalizing applications. The primary feature is that you can refer to resources by name. In your program, you ask for a resource by name rather than loading it from—for example—a hard-coded location on the file system. At runtime, Cocoa looks for the representation of a named resource that best matches the user's current language preferences.

This principle of naming resources applies to almost everything that you want to localize in your application. It applies, for example, to:

- Nib files, for storing the layout of user interfaces (such as windows and controls), text that is embedded in user interface elements, and some data formatter elements
- Strings that the user sees but that are not contained in nib files
- Icons and graphics, especially those containing culture-specific images
- Sound files that contain spoken language

Just as internationalization can be seen as referring to resources by name in Cocoa, localization consists of putting the localized representation of those names into locale-specific directories. You store localized resources for your application in special directories whose name is the name of a locale with a `.lproj` extension (for example, `fr.lproj` for French). You add to the appropriate `.lproj` directory representations of these resources that are customized for people of the specified locale.

## Prerequisite Articles

Cocoa (Touch)

## Related Articles

Nib file
Value object

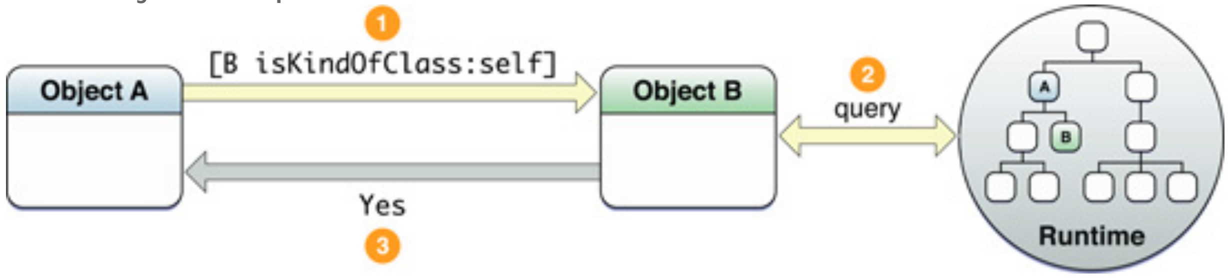
## Definitive Discussion

"Internationalization and Localization"

## Introspection

Introspection refers to the inherent ability of an object to divulge, upon request, its essential characteristics at runtime. By sending objects certain messages, you can ask them questions about themselves as objects and the Objective-C runtime provides you with answers. Introspection is an important coding tool because it makes your programs more efficient and robust. Here are a couple of examples of how introspection might be useful:

- You can call introspection methods as runtime checks to help you avoid problems such as exceptions, which, for example, would occur if you send a message to an object that cannot respond to it.
- You can also use introspection to help locate an object in the inheritance hierarchy, which would give you information about the object's capabilities.



### Types of Introspection Information

The `NSObject` protocol, which is adopted by the `NSObject` class, defines introspection methods that yield the following kinds of information about an object:

- **Class membership**. To determine if an object inherits, directly or indirectly, from a particular class, send it an `isKindOfClass:` message and evaluate the result. This method tells you if the object is a direct instance of the given class. You can also use the `class` and `superclass` methods to obtain the class or superclass of an object and then use that result in comparison operations.
- **Messages responded to.** To find out if an object's class or superclass implements a method, send the object a `respondsToSelector:` message. The parameter is a `SEL`-typed value constructed from the signature of the method using the `@selector` directive. For example:

```
•   BOOL doesRespond = [anObject
    respondsToSelector:@selector(writeToFile:atomically:)];
```

- 
- **Protocol conformance.** If a class conforms to a formal protocol, you can expect it to implement the required methods of that protocol and send messages to it accordingly. Use the `conformsToProtocol:` method to obtain this information. You specify the argument of this method using the `@protocol` directive.

### Prerequisite Articles

Message

### Related Articles

Object comparison
Protocol
Selector

### Definitive Discussion

NSObject Protocol Reference

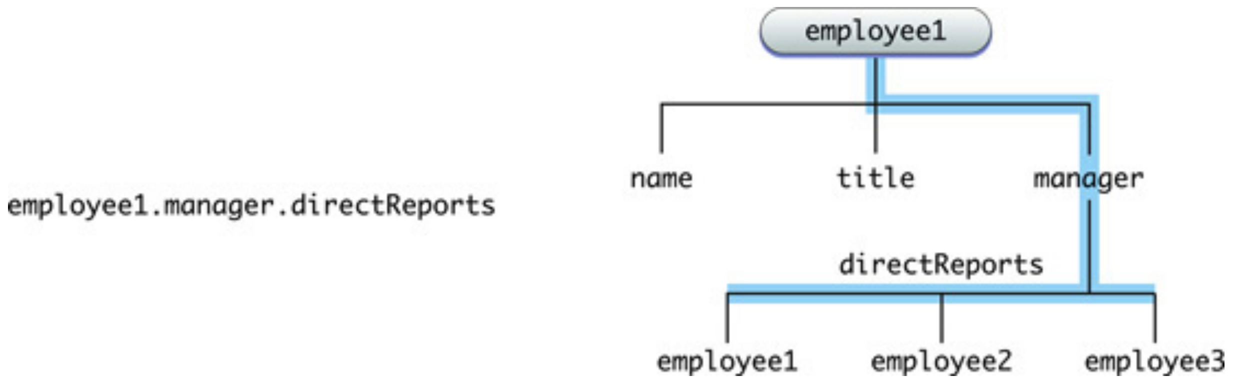
# Key-value coding

Key-value coding is a mechanism for indirectly accessing an object's attributes and relationships using string identifiers. It underpins or is related to several mechanisms and technologies special to Cocoa programming, among them Core Data, application scriptability, the bindings technology, and the language feature of declared properties. (Scriptability and bindings are specific to Cocoa on OS X.) You can also use key-value coding to simplify your program code.

## Object Properties and KVC

Central to key-value coding (or KVC) is the general notion of properties. A property refers to a unit of state that an object encapsulates. A property can be one of two general types: an attribute (for example, `name`, `title`, `subtotal`, or `textColor`) or a relationship to other objects. Relationships can be either to-one or to-many. The value for a to-many relationship is typically an array or set, depending on whether the relationship is ordered or unordered.

KVC locates an object's property through a key, which is a string identifier. A key usually corresponds to the name of an accessor method or instance variable defined by the object. The key must conform to certain conventions: It must be ASCII encoded, begin with a lowercase letter, and have no whitespace. A key path is a string of dot-separated keys that is used to specify a sequence of object properties to traverse. The property of the first key in the sequence is relative to a specific object (`employee1` in the following diagram), and each subsequent key is evaluated relative to the value of the previous property.



## Making a Class KVC Compliant

The `NSKeyValueCoding` informal protocol makes KVC possible. Two of its methods—`valueForKey:` and `setValue:forKey:`—are particularly important because they fetch and set a property's value when given its key. `NSObject` provides a default implementation of these methods, and if a class is compliant with key-value coding, it can rely on this implementation.

How you make a property KVC compliant depends on whether that property is an attribute, a to-one relationship, or a to-many relationship. For attributes and to-one relationships, a class must implement at least one of the following in the given order of preference (key refers to the property key):

1. The class has a declared property with the name key.
2. It implements accessor methods named key and, if the property is mutable, `setKey:`. (If the property is a Boolean attribute, the getter accessor method has the form `isKey`.)
3. It declares an instance variable of the form key or _key.

Implementing KVC compliance for a to-many relationship is a more complicated procedure. Refer to the document that definitively describes key-value coding to learn what this procedure is.

## Prerequisite Articles

Object modeling

Accessor method

**Related Articles**

Declared property

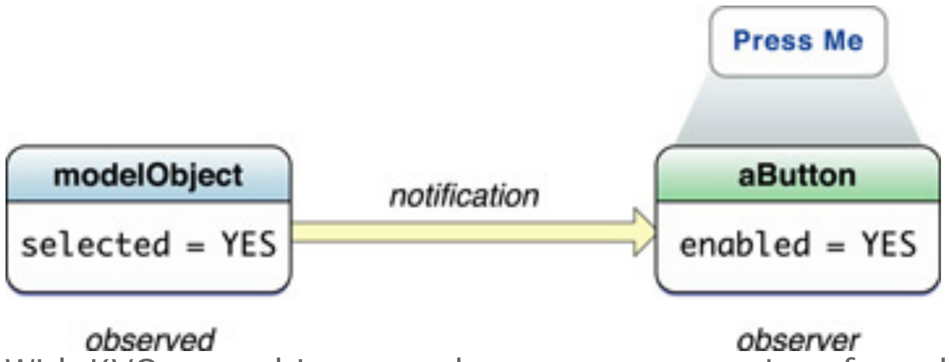
**Definitive Discussion**

Key-Value Coding Programming Guide

## Key-value observing

Key-value observing is a mechanism that enables an object to be notified directly when a property of another object changes. Key-value observing (or KVO) can be an important factor in the cohesiveness of an application. It is a mode of communication between objects in applications designed in conformance with the Model-View-Controller design pattern. For example, you can use it to synchronize the state of model objects with objects in the view and controller layers. Typically, controller objects observe model objects, and views observe controller objects or model objects.

**Note:** Although the classes of the UIKit framework generally do not support KVO, you can still implement it in the custom objects of your application, including custom views.



With KVO, one object can observe any properties of another object, including simple attributes, to-one relationships, and to-many relationships. An object can find out what the current and prior values of a property are. Observers of to-many relationships are informed not only about the type of change made, but are told which objects are involved in the change.

As a notification mechanism, key-value observing is similar to the mechanism provided by the `NSNotification` and

NSNotificationCenter classes, but there are significant differences, too. Instead of a central object that broadcasts notifications to all objects that have registered as observers, KVO notifications go directly to observing objects when changes in property values occur.

## Implementing KVO

The root class, NSObject, provides a base implementation of key-value observing that you should rarely need to override. Thus all Cocoa objects are inherently capable of key-value observing. To receive KVO notifications for a property, you must do the following things:

- You must ensure that the observed class is key-value observing compliant for the property that you want to observe.KVO compliance requires the class of the observed object to also be KVC compliant and to either allow automatic observer notifications for the property or implement manual key-value observing for the property.
- Add an observer of the object whose value can change. You do this by calling `addObserver:forKeyPath:options:context:`. The observer is just another object in your application.
- In the observer object, implement the method `observeValueForKeyPath:ofObject:change:context:`. This method is called when the value of the observed object's property changes.

## KVO Is an Integral Part of Bindings (OS X)

Cocoa bindings is an OS X technology that allows you to keep values in the model and view layers of your application synchronized without having to write a lot of "glue code." Through an Interface Builder inspector, you can establish a mediated connection between a view's property and a piece of data, "binding" them such that a change in one is reflected in the other. KVO, along with key-value coding and key-value binding, are technologies that are instrumental to Cocoa bindings.

## Prerequisite Articles

Key-value coding

## Related Articles

Model-View-Controller
Dynamic binding

## Definitive Discussion

Key-Value Observing Programming Guide

## Sample Code Projects

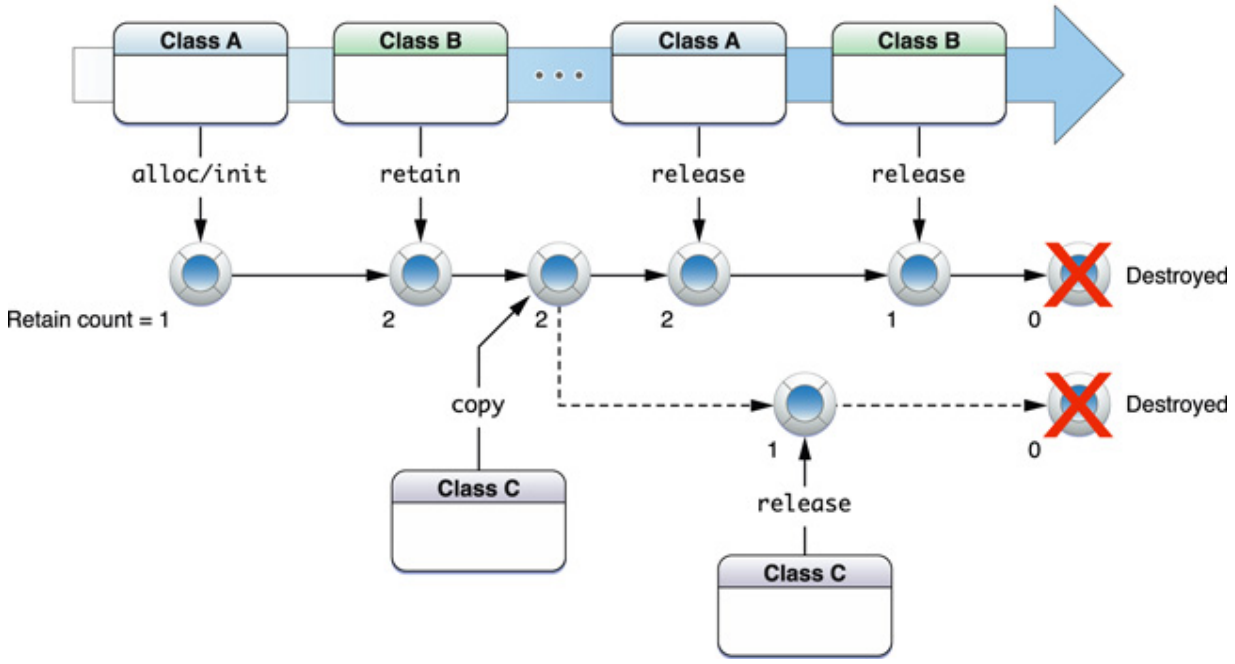
SourceView
BindingsJoystick

## Memory management

Memory management is the programming discipline of managing the life cycles of objects and freeing them when they are no longer needed. Managing object memory is a matter of performance; if an application doesn't free unneeded objects, its memory footprint grows and performance suffers. Memory management in a Cocoa application that doesn't use garbage collection is based on a reference counting model. When you create or copy an object, its retain count is 1. Thereafter other objects may express an ownership interest in your object, which increments its retain count. The owners of an object may also relinquish their ownership interest in it, which decrements the retain count. When the retain count becomes zero, the object is deallocated (destroyed).

To assist you in memory management, Objective-C gives you methods and mechanisms that you must use in conformance with a set of rules.

**Note:** In OS X, you can either explicitly manage memory or use the garbage collection feature of Objective-C. Garbage collection is not available in iOS.



### Memory-Management Rules

Memory-management rules, sometimes referred to as the ownership policy, help you to explicitly manage memory in Objective-C code.

- You own any object you create by allocating memory for it or copying it.Related methods: `alloc`, `allocWithZone:`, `copy`, `copyWithZone:`, `mutableCopy`, `mutableCopyWithZone:`
- If you are not the creator of an object, but want to ensure it stays in memory for you to use, you can express an ownership interest in it. Related method: `retain`
- If you own an object, either by creating it or expressing an ownership interest, you are responsible for releasing it when you no longer need it.Related methods: `release`, `autorelease`
- Conversely, if you are not the creator of an object and have not expressed an ownership interest, you must not release it.

If you receive an object from elsewhere in your program, it is normally guaranteed to remain valid within the method or function it was received in. If you want it to remain valid beyond that scope, you should retain or copy it. If you try to release an object that has already been deallocated, your program crashes.

## Aspects of Memory Management

The following concepts are essential to understanding and properly managing object memory:

- **Autorelease pools.** Sending `autorelease` to an object marks the object for later release, which is useful when you want the released object to persist beyond the current scope. Autoreleasing an object puts it in an autorelease pool (an instance of `NSAutoreleasePool`), which is created for an arbitrary program scope. When program execution exits that scope, the objects in the pool are released.
- **Deallocation.** When an object's retain count drops to zero, the runtime calls the `dealloc` method of the object's class just before it destroys the object. A class implements this method to free any resources the object holds, including objects pointed to by its instance variables.
- **Factory methods.** Many framework classes define class methods that, as a convenience, create objects of the class for you. These returned objects are not guaranteed to be valid beyond the receiving method's scope.

## Prerequisite Articles

Object creation

Object copying

### Related Articles

Object life cycle

### Definitive Discussion

Advanced Memory Management Programming Guide

## Message

A message is the name of a method, and any parameters associated with it, that are sent to, and executed by, an object. To get an object to do something, you send it a message telling it to apply a method. In Objective-C, you specify the object (known as the receiver of the method) and the message being sent to that object by enclosing the message expression in brackets. For example, this message expression tells the `myRectangle` object to perform its `display` method:

```
[myRectangle display];
```

(The expression is followed by a semicolon (`;`) as is normal for any line of code in C.)

The method name in a message serves to select a method implementation—when a message is sent, the runtime system selects the appropriate method from the receiver's repertoire and invokes it. For this reason, method names in messages are often referred to as selectors.

Methods can also take parameters, also called arguments. A message with a single argument affixes a colon (`:`) to the selector name and puts the parameter right after the colon. This construct is called a keyword; a keyword ends with a colon, and a parameter follows the colon. A method that takes multiple parameters has multiple keywords, each followed by a colon.

```
[myRectangle setLineWidth:0.25];


[myRectangle setWidth:20.0 height:50.0];
```

### Prerequisite Articles

(None)

**Related Articles**

Objective-C
Class definition
Selector

**Definitive Discussion**

"Objects Send and Receive Messages" in Programming with Objective-C

## Method overriding

Method overriding is a language feature in which a class can provide an implementation of a method that is already provided by one of its parent classes. The implementation in this class replaces (that is, overrides) the implementation in the parent class.

When you define a method with the same name as that of a parent class, that new method replaces the inherited definition. The new method must have the same return type and take the same number and type of parameters as the method you are overriding. Here's an example:

```
@interface MyClass : NSObject {

}

- (int)myNumber;

@end



@implementation MyClass : NSObject {

}
```

```
- (int)myNumber {

    return 1;

}

@end



@interface MySubclass : MyClass {

}

- (int)myNumber;

@end



@implementation MySubclass

- (int)myNumber {

    return 2;

}

@end
```

If you create an instance of `MyClass` and send it a `myNumber` message, it returns 1. If you create an instance of `MySubclass` and send it a `myNumber` message, it returns 2.

Notice that the subclass's method must have the same name and parameter list as the superclass's overridden method.

In addition to completely replacing an existing implementation, you might want to extend a superclass's implementation. To do this, you can invoke the superclass's implementation using the `super` keyword.

Within a method definition, `super` refers to the parent class (the superclass) of the current object. You send a message to `super` to execute the superclass's implementation of a method. You often do this within a new implementation of the same method to extend its functionality. In the following example, the implementation of `myNumber` by `MySubclass` simply adds 1 to whatever value is returned by the implementation of `MyClass`.

```
@implementation MySubclass

- (int)myNumber {

    int subclassNumber = [super myNumber] + 1;

    return subclassNumber;

}

@end
```

### Prerequisite Articles

(None)

### Related Articles

(None)

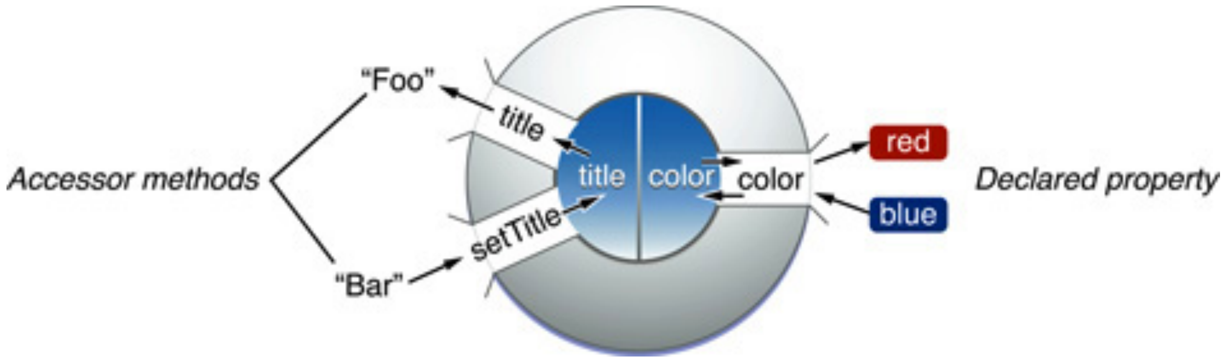
### Definitive Discussion

"Objects Can Call Methods Implemented by Their Superclasses" in Programming with Objective-C

## Model object

A model object is a type of object that contains the data of an application, provides access to that data, and implements logic to manipulate the data. Model objects play one of the three roles defined by the Model-View-Controller design pattern. (The other two roles are played by view and controller 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 after the data is loaded into the application.

Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused when that problem domain is in effect. Ideally, a model object should have no explicit connection to the view objects that present its data and allow users to edit that data—in other words, it should not be concerned with user-interface and presentation issues.



## A Well-Designed Model Class

A model class—that is, a class that produces model objects—is typically a subclass of `NSObject` or, if you are taking advantage of the Core Data technology, a subclass of `NSManagedObject`. To create a model object, define it as you would a basic class and observe the Cocoa naming conventions. For example, begin the names of instance variables, declared properties, and declared methods with a lowercase letter and capitalize the first letter of embedded words.

When you implement your model subclass, you should consider the following aspects of class design:

- **Instance variables.** You declare instance variables to hold the encapsulated data of an application. Instance variables can be objects, scalar values, or structures such as `NSRange`. There are tradeoffs to using object versus nonobject types, and object mutability is a consideration.
- **Accessor methods and declared properties.** Accessor methods

and declared properties provide ways to preserve encapsulation because they mediate access to an object’s instance data. Accessor methods typically get and set the values of instance variables (and are colloquially known as getter and setter methods). Declared properties are a language-level convenience that allow the runtime to synthesize accessor methods for a class. An important role of accessor methods and declared properties is to manage object memory. For this reason, there are recommended forms for implementing getter methods and setter methods.

- **Key-value coding.** Key-value coding is a mechanism that lets clients access an object's property using the property name as a key. It is used by Core Data and elsewhere in Cocoa. The naming of accessor methods (and, implicitly, declared properties) is a factor in this mechanism.
- **Initialization and deallocation.** In most cases, a model class implements an initializer method to set its instance variables to reasonable initial values. It should follow the standard form for initializer methods. It should also ensure that it releases any instance variables holding object values in its `dealloc` method.
- **Object encoding.** If you expect objects of your model class to be archived, make sure that the class encodes and decodes the instance variables of its objects.
- **Object copying.** If you expect clients to copy your model objects, your class should implement object copying.

### Prerequisite Articles

Model-View-Controller
Class definition

### Related Articles

Coding conventions
Declared property
Object copying

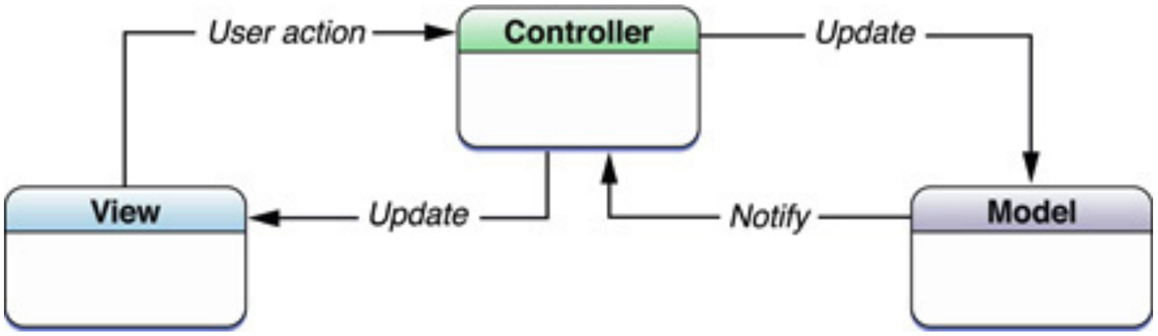
### Definitive Discussion

Programming with Objective-C

## Model-View-Controller

The Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, it defines the way objects communicate with each other. 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. The collection of objects of a certain MVC type in an application is sometimes referred to as a layer—for example, model layer.

MVC is central to a good design for a Cocoa application. The benefits of adopting this pattern are numerous. Many objects in these applications tend to be more reusable, and their interfaces tend to be better defined. Applications having an MVC design are also more easily extensible than other applications. Moreover, many Cocoa technologies and architectures are based on MVC and require that your custom objects play one of the MVC roles.



## Model Objects

Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data. For example, a model object might represent a character in a game or a contact in an address book. A model object can have to-one and to-many relationships with other model objects, and so sometimes the model layer of an application effectively is one or more object graphs. Much of the 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 after the data is loaded into the application. Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused in similar problem domains. Ideally, a model object should have no explicit connection to the view objects that present its data and allow users to edit that data—it should not be concerned with user-interface and presentation issues.

**Communication**: User actions in the view layer that create or modify data are communicated through a controller object and result in the creation or updating of a model object. When a model object changes (for example, new data is received over a network connection), it notifies a controller object, which updates the appropriate view objects.

## View Objects

A view object is an object in an application that users can see. A view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data. Despite this, view objects are typically decoupled from model objects in an MVC application.

Because you typically reuse and reconfigure them, view objects provide consistency between applications. Both the UIKit and AppKit frameworks provide collections of view classes, and Interface Builder offers dozens of view objects in its Library.

**Communication**: View objects learn about changes in model data through the application's controller objects and communicate user-initiated changes—for example, text entered in a text field—through controller objects to an application's model objects.

## Controller Objects

A controller object acts as an intermediary between one or more of an application's view objects and one or more of its model objects. Controller objects are thus a conduit through which view objects learn about changes in model objects and vice versa. Controller objects can also perform setup and coordinating tasks for an application and manage the life cycles of other objects.

**Communication**: A controller object interprets user actions made in view objects and communicates new or changed data to the model layer. When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

## Prerequisite Articles

Message

## Related Articles

Model object
Controller object

## Definitive Discussion

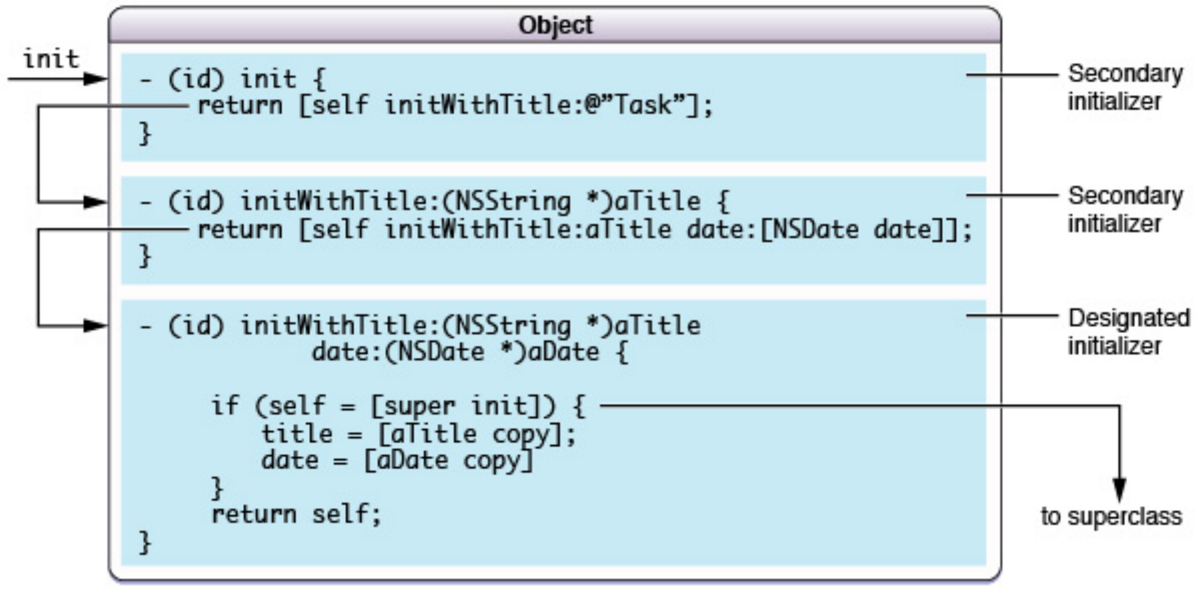
"Model-View-Controller" in Cocoa Fundamentals Guide

## Multiple initializers

A class may define multiple initializer methods, either when it can take different forms of input or to provide default initialization values as a convenience to clients. If the class can take initialization data in different forms or from different sources, it can declare an initializer for each form or source. For example, the `NSString` class has initializers for obtaining string data as arrays of Unicode characters or from URL resources (among others).

Multiple initializers can also be a programming convenience. A class can have a series of initializers that, at one end, allow the client to specify all initial values and, at the other end, supply most or all of these values as defaults. Clients of the class may later be able to substitute new values for the default values through accessor methods or properties.

Framework classes sometimes have multiple factory methods as well as multiple initializers. They are similar in that they may be a convenience or take initialization data in different forms. However, they allocate the returned object as well as initialize it.



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

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

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

     if (self = [super init]) {
         title = [aTitle copy];
         date = [aDate copy]
     }
     return self;
}
```

### The Designated Initializer

The initializer of a class that takes the full complement of initialization parameters is usually the designated initializer. The designated initializer of a subclass must invoke the designated initializer of its superclass by sending a message to `super`. The convenience (or secondary) initializers

—which can include `init`—do not call `super`. Instead they call (through a message to `self`) the initializer in the series with the next most parameters, supplying a default value for the parameter not passed into it. The final initializer in this series is the designated initializer.

### Prerequisite Articles

Object creation
Initialization

### Related Articles

Declared property

### Definitive Discussion

"Objects Are Created Dynamically" in Programming with Objective-C

## Nib file

A nib file is a special type of resource file that you use to store the user interfaces of iOS and Mac apps. A nib file is an Interface Builder document. You use Interface Builder to design the visual parts of your app—such as windows and views—and sometimes to configure nonvisual objects, such as the controller objects that your app uses to manage its windows and views. In effect, as you edit an Interface Builder document, you create an object graph that is then archived when you save the file. When you load the file, the object graph is unarchived.

The nib file—and hence the object graph—may contain placeholder objects that are used to refer to objects that live outside of the document but that may have references to objects in the document, or to which objects in the document may have references. A special placeholder is the File's Owner.

At runtime, you load a nib file using the method `loadNibNamed:owner:` or a variant thereof. The File's Owner is a placeholder in the nib file for the object that you pass as the owner parameter of that method. Whatever connections you establish to and from the File's Owner in the nib file in Interface Builder are reestablished when you load the file at runtime.

iOS uses nibs as an implementation detail that supports storyboards, the iOS user interface design layout format. Storyboards allow you to design and visualize the entire user interface of your app on one canvas. For iOS developers, using storyboards is the recommended way to design user interfaces.

### Prerequisite Articles

- Archiving
- Object graph

### Related Articles

- Model-View-Controller
- Storyboard

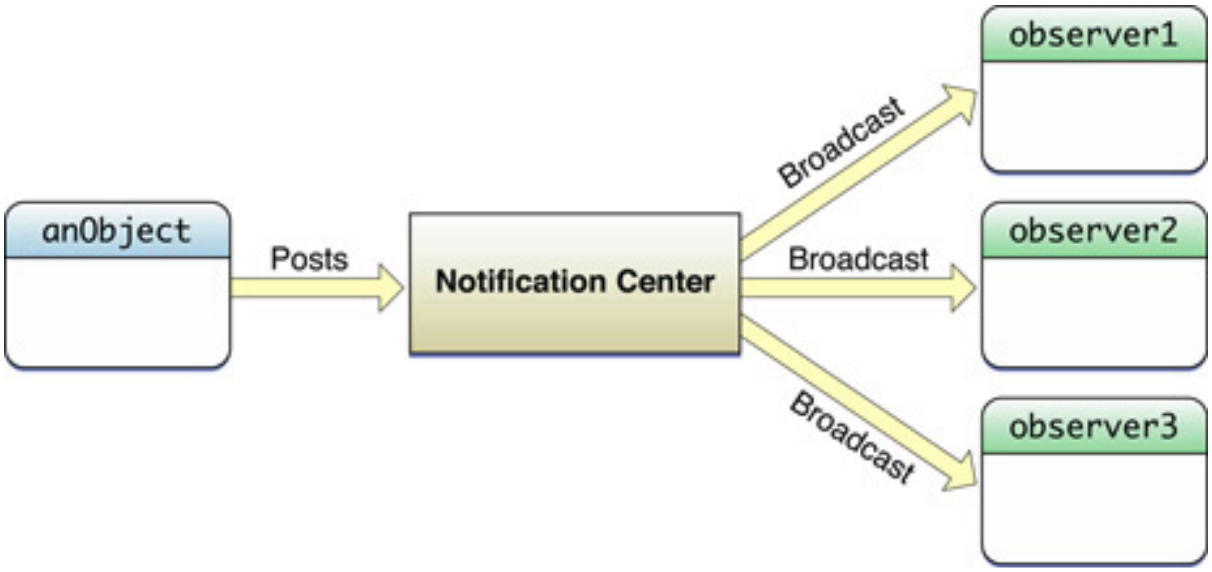
### Definitive Discussion

- "Nib Files"

## Notification

A notification is a message sent to one or more observing objects to inform them of an event in a program. The notification mechanism of Cocoa follows a broadcast model. It is a way for an object that initiates or handles a program event to communicate with any number of objects that want to know about that event. These recipients of the notification, known as observers, can adjust their own appearance, behavior, and state in response to the event. The object sending (or posting) the notification doesn't have to know what those observers are. Notification is thus a powerful mechanism for attaining coordination and cohesion in a program. It reduces the need for strong dependencies between objects in a program (such dependencies would reduce the reusability of those objects). Many classes of the Foundation, AppKit, and other Objective-C frameworks define notifications that your program can register to observe.

The centerpiece of the notification mechanism is a per-process singleton object known as the notification center (`NSNotificationCenter`). When an object posts a notification, it goes to the notification center, which acts as a kind of clearing house and broadcast center for notifications. Objects that need to know about an event elsewhere in the application register with the notification center to let it know they want to be notified when that event happens. Although the notification center delivers a notification to its observers synchronously, you can post notifications asynchronously using a notification queue (NSNotificationQueue).



## The Notification Object

A notification is represented by an instance of the NSNotification class. A notification object contains several bits of state: a unique name, the posting object, and (optionally) a dictionary of supplementary information, called the userInfo dictionary. When a notification is delivered to an interested observer, the notification object is passed in as an argument of the method handling the notification.

## Observing a Notification

To observe a notification, obtain the singleton NSNotificationCenter instance and send it an addObserver:selector:name:object: message. Typically, this registration step is done shortly after your application launches. The second parameter of the addObserver:selector:name:object: method is a selector that identifies the method that you implement to handle the notification. The method must have the following signature:

```
- (void)myNotificationHandler:(NSNotification *)notif;
```

In this handling method, you can extract information from the notification to help you in your response, especially data in the userInfo dictionary (if one exists).

## Posting a Notification

Before posting a notification, you should define a unique global string constant as the name of your notification. A convention is to use a two-

or three-letter application-specific prefix for the name, for example:

```
NSString *AMMyNotification = @"AMMyNotication";
```

To post the notification, send a `postNotificationName:object:userInfo:` (or similar) message to the singleton `NSNotificationCenter` object. This method creates the notification object before it sends the notification to the notification center.

### Prerequisite Articles

Selector
Message

### Related Articles

Delegation
Model-View-Controller
Singleton

### Definitive Discussion

Notification Programming Topics

### Sample Code Projects

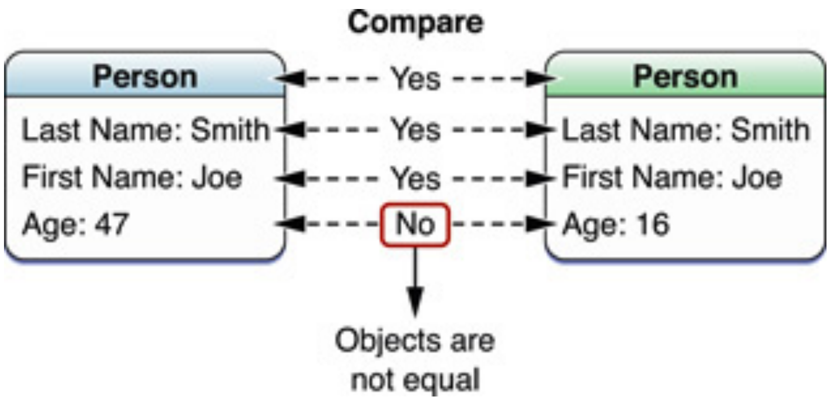
HeadsUpUI

## Object comparison

Object comparison refers to the ability of an object to determine whether it is essentially the same as another object. You evaluate whether one object is equal to another by sending one of the objects an `isEqual:` message and passing in the other object. If the objects are equal, you receive back `YES`; if they are not equal, you receive `NO`. Each class determines equality for its instances by implementing class-specific comparison logic. The root class, `NSObject`, measures equality by simple pointer comparison; at the other extreme, the determinants of equality for a custom class might be class membership plus all encapsulated values.

Some classes of the Foundation framework implement comparison methods of the form `isEqualTo`Type`:`—for example, `isEqualToString:` and `isEqualToArray:`. These methods perform comparisons specific to the given class type.

The comparison methods are indispensable coding tools that can help you decide at runtime what to do with an object. The collection classes such as NSArray and NSDictionary use them extensively.



## Implementing Comparison Logic

If you expect instances of your custom subclass to be compared, override the isEqual: method and add comparison logic that is specific to your subclass. Your class, for example, might accept the superclass's determination of equality but then add further tests. Your class may have one or more instance variables whose values should be equal before two instances of your class can be considered equal. The following implementation of isEqual: performs a series of checks, ending with one that is class-specific (the name property).

```
- (BOOL)isEqual:(id)other {

    if (other == self)

        return YES;

    if (![super isEqual:other])

        return NO;

    return [[self name] isEqualToString:[other name]]; //
class-specific
}
```

If you override `isEqual:`, you should also implement the `hash` method to generate and return an integer that can be used as a table address in a hash table structure. If `isEqual:` determines that two objects are equal, they must have the same hash value.

### Prerequisite Articles

Message

### Related Articles

Introspection

### Definitive Discussion

(None)

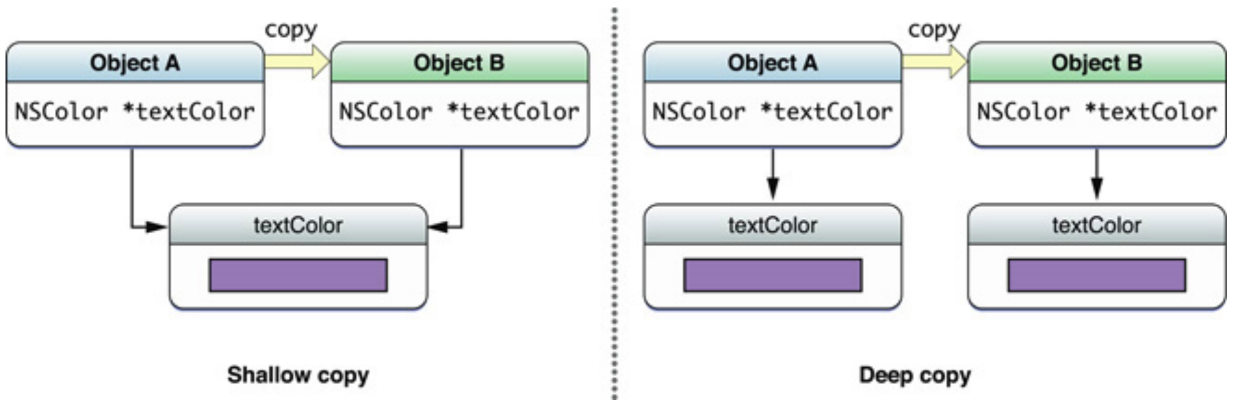
## Object copying

Copying an object creates a new object with the same class and properties as the original object. You copy an object when you want your own version of the data that the object contains. If you receive an object from elsewhere in an application but do not copy it, you share the object with its owner (and perhaps others), who might change the encapsulated contents. If you are creating a subclass, you might consider making it possible for others to copy instances of your class. Generally, an object should be “copyable” when it is a value object—an object whose main purpose is to encapsulate some data.

### Requirements for Object Copying

An object can be copied if its class adopts the `NSCopying` protocol and implements its single method, `copyWithZone:`. If a class has mutable and immutable variants, the mutable class should adopt the `NSMutableCopying` protocol (instead of `NSCopying`) and implement the `mutableCopyWithZone:` method to ensure that copied objects remain mutable. You make a duplicate of an object by sending it a `copy` or `mutableCopy` message. These messages result in the invocation of the appropriate `NSCopying` or `NSMutableCopying` method.

Copies of objects can be shallow or deep. Both shallow- and deep-copy approaches directly duplicate scalar properties but differ on how they handle pointer references, particularly references to objects (for example, `NSString *str`). A deep copy duplicates the objects referenced while a shallow copy duplicates only the references to those objects. So if object A is shallow-copied to object B, object B refers to the same instance variable (or property) that object A refers to. Deep-copying objects is preferred to shallow-copying, especially with value objects.



### Memory-Management Implications

Like object creation, object copying returns an object with a retain count of 1. In memory-managed code, the client copying the object is responsible for releasing the copied object. Copying an object is similar in purpose to retaining an object in that both express an ownership in the object. However, a copied object belongs exclusively to the new owner, who can mutate it freely, while a retained object is shared between the owners of the original object and clients who have retained the object.

### Prerequisite Articles

Object creation
Protocol

### Related Articles

Memory management
Object life cycle

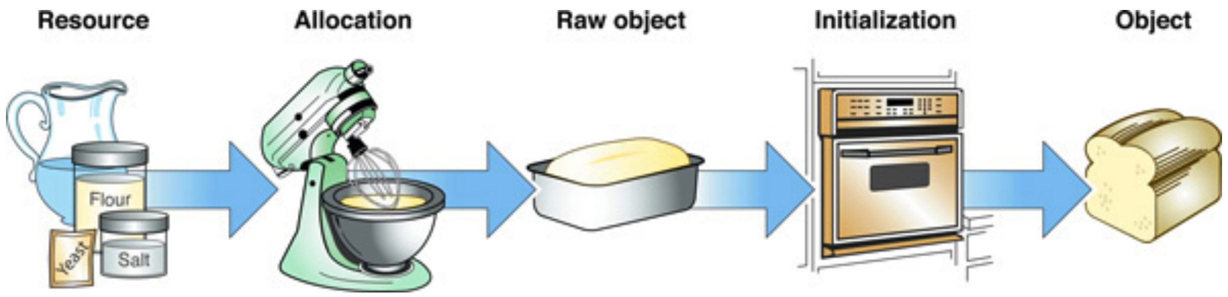
### Definitive Discussion

NSCopying Protocol Reference

## Object creation

An object comes into runtime existence through a two-step process that allocates memory for the object and sets its state to reasonable initial values. To allocate an Objective-C object, send an `alloc` or `allocWithZone:` message to the object's class. The runtime allocates memory for the object and returns a “raw” (uninitialized) instance of the class. It also sets a pointer (known as the `isa` pointer) to the object’s class, zeros out all instance variables to appropriately typed values, and sets the object’s retain count to 1.

After you allocate an object, you must initialize it. Initialization sets the instance variables of an object to reasonable initial values. It can also allocate and prepare other global resources needed by the object. You initialize an object by invoking an `init` method or some other method whose name begins with `init`. These initializer methods often have one or more parameters that enable you to specify beginning values of an object’s instance variables. If these methods succeed in initializing an object, they return it; otherwise, they return `nil`. If an object’s class does not implement an initializer, the Objective-C runtime invokes the initializer of the nearest ancestor instead.



## The Form of an Object-Creation Expression

A convention in Cocoa programming is to nest the allocation call inside the initialization call.

```
MyCustomClass *myObject = [[MyCustomClass alloc] init];
```

When you create an object using this form, you should verify that the returned value is not `nil` before proceeding. In memory-managed code, an object’s instance variables and other allocated memory should be deallocated before that object itself is released.

## Memory-Management Implications

In code that explicitly manages memory, the allocation and initialization procedure returns an object with a retain count of 1. This means that the client receiving the object now “owns” the object and is responsible for releasing it. You release it by sending it a `release` or `autorelease` message; the latter message causes a delayed release. If you do not release objects that you own, your program will leak memory.

### Factory Methods

A factory method is a class method that, as a convenience to clients, creates an instance of a class. A factory method combines allocation and initialization in one step and returns an autoreleased instance of the class. Because the received object is autoreleased, the client must copy or retain the instance if it wants it to persist in memory. The names of factory methods have the following initial form:

`+ (id)`typeRemainderOfMethodName

where type is the class name minus the prefix and RemainderOfMethodName often begins with `With` or `From`. For example,

```
+ (id)dataWithContentsOfURL:(NSURL *)url;
```

### Prerequisite Articles

Message

### Related Articles

Initialization
Memory management
Object life cycle

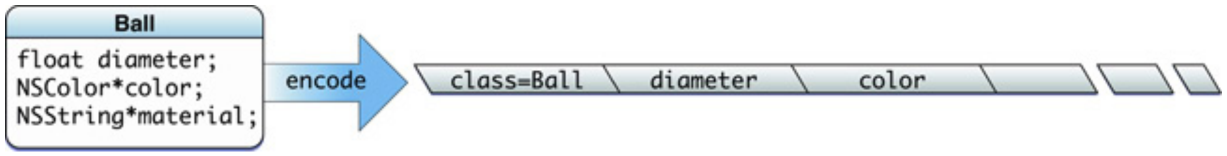
### Definitive Discussion

"Working with Objects" in Object Oriented Programming and the Objective-C Programming Language 1.0

## Object encoding

Object encoding converts an object's class identity and state to a format that can be stored or transferred between processes. The class type and instance data are written to a byte stream that can persist after a program terminates. When the program is launched again, a newly allocated object can decode the stored representation of itself and restore itself to its previous runtime state. Encoding usually occurs in concert with archiving, which puts a graph of objects into a format (an archive) that can be written to the file system; unarchiving operates on an archive, asking each object in the stored graph to decode itself.

Object encoding is also used in the OS X distributed objects API for transferring objects from one process to another. However, its most common use is for archiving, which like a property list, is a mechanism for object persistence.



## How to Encode and Decode Objects

For your subclass to encode and decode its instances, it must conform to the `NSCoding` protocol and implement two methods: `initWithCoder:` and `encodeWithCoder:`. When a program unarchives or archives an object graph, these methods are invoked. In the `encodeWithCoder:` method, you encode the values of an object's important instance variables; in the `initWithCoder:` method, you decode those values and reassign them to their instance variables. If an object receives an `initWithCoder:` message, none of its initializer methods are invoked.

The sole argument of `initWithCoder:` and `encodeWithCoder:` is an `NSCoder` object whose purpose is to perform the actual decoding or encoding. Because `NSCoder` is an abstract class, the coder object is in most cases an instance of one of the following concrete subclasses: `NSKeyedArchiver`, `NSKeyedUnarchiver`, `NSArchiver`, `NSUnarchiver`. The archiver classes declare methods for encoding an object's instance variables; the unarchiver classes declare methods for decoding instance variables.

The `NSCoder` methods work on objects, scalars, C arrays, structures, and strings, and on pointers to these types. Before you encode or decode an instance variable of your own class, be sure to first invoke the superclass implementation of `initWithCoder:` or `encodeWithCoder:`. When you decode objects from the byte stream, be sure to retain or copy them when you assign them to their instance variables.

## Keyed Versus Sequential Archiving

Two of the concrete `NSCoder` subclasses are different from each other in a fundamental way. The "keyed" archiver class (`NSKeyedArchiver` and `NSKeyedUnarchiver`) associate an encoded value with a string key and use that same key when decoding that value; thus instance variables can be encoded and decoded in any sequence. With the other type of coder (`NSArchiver` and `NSUnarchiver`) you encode instance variables in a certain sequence, and you must decode them in the same sequence. The sequential coders should be used only with legacy code; new subclasses should use keyed archive coders.

### Prerequisite Articles

Class definition
Protocol

### Related Articles

Archiving
Property list
Object life cycle

### Definitive Discussion

"Encoding and Decoding Objects"

### Sample Code Projects

Reducer

## Object graph

In an object-oriented program, groups of objects form a network through their relationships with each other—either through a direct reference to another object or through a chain of intermediate references. These groups of objects are referred to as object graphs. Object graphs may be small or large, simple or complex. An array object that contains a single string object represents a small, simple object graph. A group of objects containing an application object, with references to the windows, menus and their views, and other supporting objects, may represent a large, complex object graph.

Sometimes you may want to convert an object graph—usually just a section of the full object graph in the application—into a form that can be saved to a file or transmitted to another process or machine and then reconstructed. This process is known as archiving.

Some object graphs may be incomplete—these are often referred to as partial object graphs. Partial object graphs have placeholder objects that represent the boundaries of the graph and that may be filled in at a later stage. An example is a nib file that includes a placeholder for the File's Owner.

### Related Articles

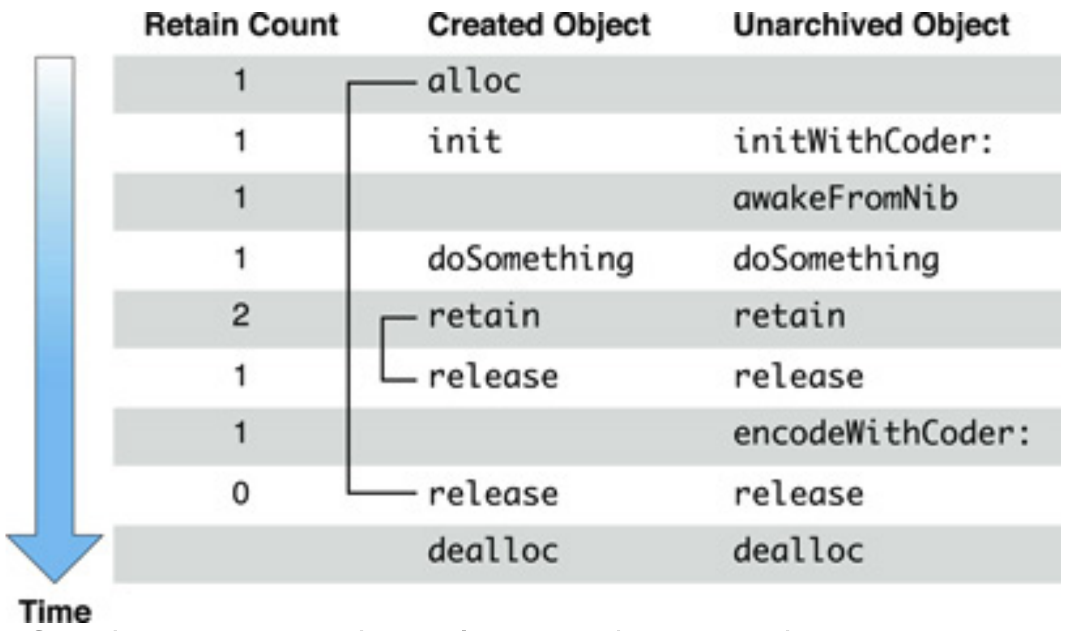
Archiving

Nib file

## Object life cycle

An object’s life cycle—that is, its runtime life from its creation to its destruction—is marked or determined by various messages it receives. An object comes into being when a program explicitly allocates and initializes it or when it makes a copy of another object. An object can also begin its runtime life during unarchiving, when it is asked to decode itself from the archive byte stream. If an object was unarchived from a nib file, it receives an `awakeFromNib` message after all objects in the nib file have been loaded into memory.

**Note:** This article describes a concept related to explicit memory management, which has been supplanted since iOS 5.0 by Automatic Reference Counting (ARC). ARC is a compiler feature that provides automatic memory management of Objective-C objects.



| Retain Count | Created Object | Unarchived Object |
|---|---|---|
| 1 | alloc | |
| 1 | init | initWithCoder: |
| 1 | | awakeFromNib |
| 1 | doSomething | doSomething |
| 2 | retain | retain |
| 1 | release | release |
| 1 | | encodeWithCoder: |
| 0 | release | release |
| | dealloc | dealloc |

After the creation and initialization phase, an object remains in memory as long as its retain count is greater than zero. Other objects in the program may express an ownership interest in an object by sending it `retain` or by copying it, and then later relinquish that ownership interest by sending `release` to the object. While the object is viable, a program may begin the archiving process, in which the object encodes its state in the archive byte stream. When the object receives its final `release` message, its retain count drops to zero. Consequently, the object's `dealloc` method is called, which frees any objects or other memory it has allocated, and the object is destroyed.

### Prerequisite Articles

Object creation
Archiving

### Related Articles

Memory management
Nib file
Object encoding

### Definitive Discussion

"Working with Objects" in Cocoa Fundamentals Guide

## Object modeling

Object modeling is the process of designing the objects or classes through which an object-oriented application examines and manipulates some service. Numerous modeling techniques are possible; the Cocoa development environment does not recommend one over another.

Typically, the characteristics of a class should make sense in context. Aspects such as the names of the class itself, its variables, and its methods should be recognizable to a nonprogrammer who is familiar with the service being modeled. Exactly what classes you use in your application, what attributes each class has, and what the relationships are between the classes may depend on the way you present information to the user and how you expect the user to interact with your application.

### Prerequisite Articles

(None)

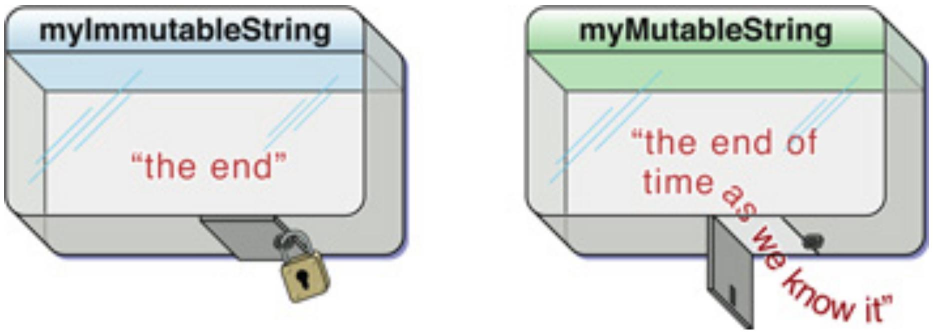
### Related Articles

Model-View-Controller

## Object mutability

Most Cocoa objects are mutable—meaning you can change their encapsulated values—but some are immutable, and you cannot change their encapsulated values after they are created. The main benefit of immutable objects is the assurance that their values won't change while you're using them. Once you create an immutable object, the value it represents remains the same throughout its runtime life. But you can change the encapsulated value of a mutable object at any time. You can do this through methods that replace the values ("setter" accessor methods) or methods that incrementally modify them.

Object mutability is particularly important with the Foundation framework classes that represent collections and primitive data types. These classes have immutable and mutable variants, for example `NSArray` and `NSMutableArray`, or `NSData` and `NSMutableData`. The mutable class is a subclass of the immutable class and produces an object whose represented value or, in the case of collection classes, contained values can be changed.



## Receiving Mutable Objects

When you call a method and receive an object in return, the object could be mutable even if the method's return type characterizes it as immutable. There is nothing to prevent a class from declaring a method to return an immutable object but returning a mutable object in its implementation. Although you could use introspection to determine whether a received object is actually mutable or immutable, you shouldn't. Always use the return type of an object to judge its mutability.

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.

## Storing Mutable Objects

When you design a subclass, a common decision is whether instance variables should be typed as the mutable or immutable variant of a given class. 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.

### Prerequisite Articles

Class definition
Message

### Related Articles

Introspection
Declared property

### Definitive Discussion

"Object Mutability" in Cocoa Fundamentals Guide

## Objective-C

Objective-C defines a small but powerful set of extensions to the ANSI C programming language that enables sophisticated object-oriented programming. Objective-C is the native language for Cocoa programming—it's the language that the frameworks are written in, and the language that most applications are written in. You can also use some other languages—such as Python and Ruby—to develop programs using the Cocoa frameworks. It's useful, though, to have at least a basic understanding of Objective-C because Apple's documentation and code samples are typically written in terms of this language.

Because Objective-C rests on a foundation of ANSI C, you can freely intermix straight C code with Objective-C code. Moreover, your code can call functions defined in non-Cocoa programmatic interfaces, such as the BSD library interfaces in `/usr/include`. You can even mix C++ code with your Cocoa code and link them into the same executable.

### Prerequisite Articles

(None)

## Related Articles

(None)

## Definitive Discussion

Programming with Objective-C

# Property list

A property list is a representation of a hierarchy of objects that can be stored in the file system and reconstituted later. Property lists give applications a lightweight and portable way to store small amounts of data. They are hierarchies of data made from specific types of objects—they are, in effect, an object graph. Property lists are easy to create programmatically and are even easier to serialize into a representation that is persistent. Applications can later read the static representation back into memory and recreate the original hierarchy of objects. Both Cocoa Foundation and Core Foundation have APIs related to property list serialization and deserialization.

## Property List Types and Objects

Property lists consist only of certain types of data: dictionaries, arrays, strings, numbers (integer and float), dates, binary data, and Boolean values. Dictionaries and arrays are special types because they are collections; they can contain one or multiple data types, including other dictionaries and arrays. This hierarchical nesting of objects creates a graph of objects. The abstract data types have corresponding Foundation classes, Core Foundation types, and XML elements for collection objects and value objects, as shown in the following list:

| Abstract type | Foundation framework class | Core Foundation type | XML element |
|---|---|---|---|
| Array | `NSArray` | `CFArrayRef` | `<array>` |
| Dictionary | `NSDictionary` | `CFDictionaryRef` | `<dict>` |
| String | `NSString` | `CFStringRef` | `<string>` |
| Data | `NSData` | `CFDataRef` | `<data>` |

| | | | |
|---|---|---|---|
| Date | NSDate | CFDateRef | `<date>` |
| Integer | NSNumber (intValue 32-bit) NSNumber (integerValue 64-bit) | CFNumberRef (kCFNumberSInt32Type) CFNumberRef (kCFNumberSInt64Type) | `<integer>` |
| Floating-point value | NSNumber (floatValue 32-bit) NSNumber (doubleValue 64-bit) | CFNumberRef (kCFNumberFloat32Type) CFNumberRef (kCFNumberFloat64Type) | `<real>` |
| Boolean | NSNumber (boolValue) | CFBooleanRef | `<true/>` or `<false/>` |

Mutable versions of the Foundation classes are also supported.

Collectively, instances of these classes are known as property list objects. For example, an `NSMutableDictionary` object is a property list object, as are `NSNumber` object, an `NSString` object, and so on. For a property list to be valid, all objects in the object graph must be property list objects.

## Best Practices for Property Lists

You can write property lists out in both XML and binary formats. The binary format is much more compact than the XML version and thus more efficient. It is recommended for most situations. However, you can manually edit an XML property list if you ever need to. Property list files have the filename extension of `plist`.

You should not use property lists to store large, complex graphs of objects, especially when the objects have variable mutability settings. And you cannot use property lists to store objects that are not supported by the architecture, such as model objects. For these cases, use archiving instead. Although property lists can include `NSData` objects, it's best to not use data objects in property lists to hold large amounts of binary data.

### Property List Serialization

To serialize and deserialize property lists, call the appropriate class methods of the `NSPropertyListSerialization` class or, if using Core Foundation, the facilities related to the `CFPropertyListRef` opaque type. In Cocoa, the serialized output is in the form of an `NSData` object. You can therefore use the methods of that class (for example, `writeToFile:atomically:`) to write that data to the file system and use the appropriate `NSData` class factory memory to read it back into memory. Then, when you deserialize it, you can specify mutability options for the property list.

### Prerequisite Articles

Object graph
Collection

### Related Articles

Archiving
Object mutability

### Definitive Discussion

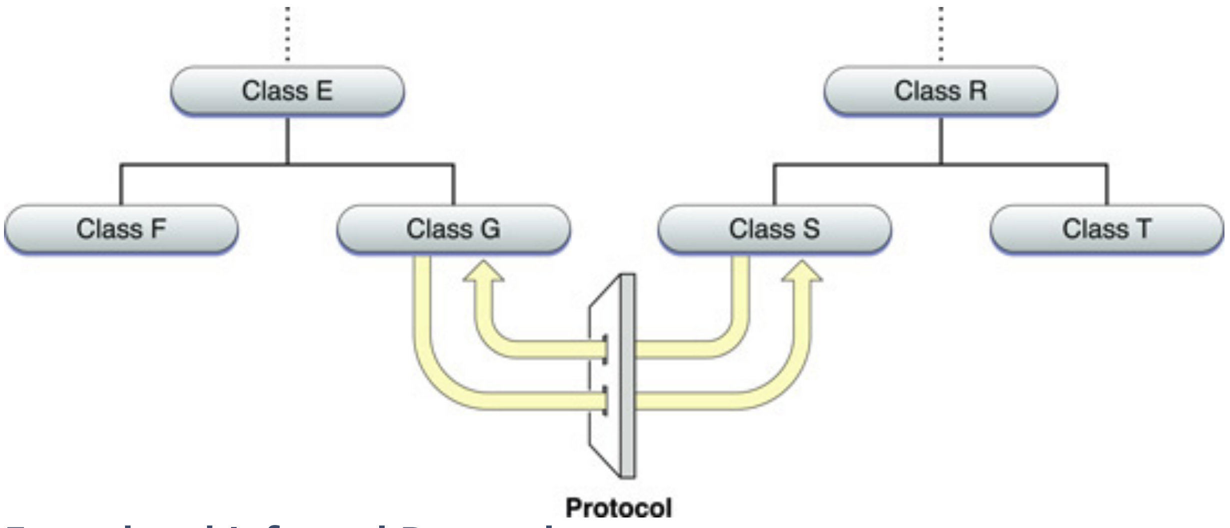
Property List Programming Guide

### Sample Code Projects

People

# Protocol

A protocol declares a programmatic interface that any class may choose to implement. Protocols make it possible for two classes distantly related by inheritance to communicate with each other to accomplish a certain goal. They thus offer an alternative to subclassing. Any class that can provide behavior useful to other classes may declare a programmatic interface for vending that behavior anonymously. Any other class may choose to adopt the protocol and implement one or more of its methods, thereby making use of the behavior. The class that declares a protocol is expected to call the methods in the protocol if they are implemented by the protocol adopter.



## Formal and Informal Protocols

There are two varieties of protocol, formal and informal:

- A formal protocol declares a list of methods that client classes are expected to implement. Formal protocols have their own declaration, adoption, and type-checking syntax. You can designate methods whose implementation is required or optional with the `@required` and `@optional` keywords. Subclasses inherit formal protocols adopted by their ancestors. A formal protocol can also adopt other protocols.Formal protocols are an extension to the Objective-C language.
- An informal protocol is a category on `NSObject`, which implicitly makes almost all objects adopters of the protocol. (A category is a language feature that enables you to add methods to a class without subclassing it.) Implementation of the methods in an informal protocol is optional. Before invoking a method, the calling object checks to see whether the target object implements it. Until optional protocol methods were introduced in Objective-C 2.0, informal protocols were essential to the way Foundation and AppKit classes implemented delegation.

## Adopting and Conforming to a Formal Protocol

A class can either declare adoption of a formal protocol or inherit adoption from a superclass. The adoption syntax uses angle brackets in the `@interface` declaration of the class. In the following example, the `CAAnimation` class declares its superclass to be `NSObject` and then formally adopts three protocols.

```
@interface CAAnimation : NSObject <NSCopying, CAMediaTiming,
CAAction>
```

A class—and any instance of that class—are said to conform to a formal protocol if the class adopts the protocol or inherits from another class that adopts it. Conformance to a protocol also means that a class implements all the required methods of the protocol. You can determine at runtime whether an object conforms to a protocol by sending it a `conformsToProtocol:` message.

### Creating Your Own Protocol

You may also declare your own protocol and implement the code that communicates with adopters of that protocol. For a description of how to do this, see the document that definitively describes protocols.

### Prerequisite Articles

Objective-C

### Related Articles

Category
Delegation
Introspection

### Definitive Discussion

“Working with Protocols”

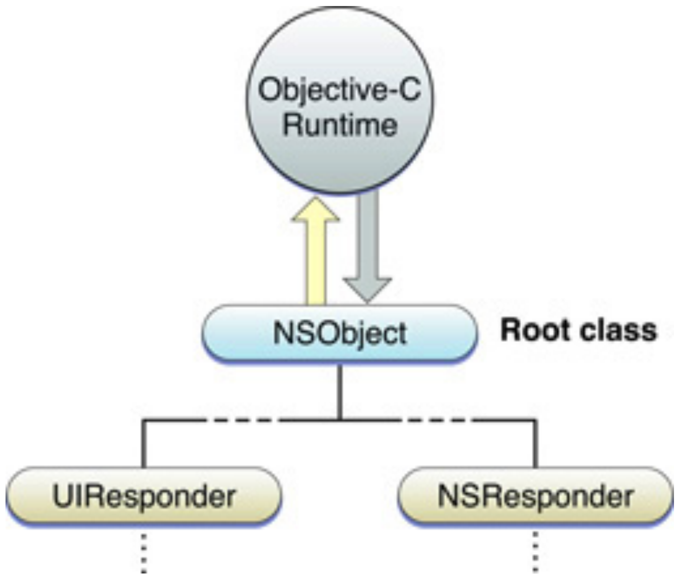
### Sample Code Projects

LocateMe

# Root class

A root class inherits from no other class and defines an interface and behavior common to all objects in the hierarchy below it. All objects in that hierarchy ultimately inherit from the root class. A root class is sometimes referred to as a base class.

The root class of all Objective-C classes is `NSObject`, which is part of the Foundation framework. All objects in a Cocoa or Cocoa Touch application ultimately inherit from `NSObject`. This class is the primary access point whereby other classes interact with the Objective-C runtime. It also declares the fundamental object interface and implements basic object behavior, including introspection, memory management, and method invocation. Cocoa and Cocoa Touch objects derive the ability to behave as objects in large part from the root class.



**Note:** The Foundation framework defines another root class, `NSProxy`, but this class is rarely used in Cocoa applications and never in Cocoa Touch applications.

The root class `NSObject` adopts a protocol, also named `NSObject`, which contributes to its programmatic interface. The protocol specifies the basic programmatic interface required of any root class.

### Prerequisite Articles

Cocoa (Touch)

### Related Articles

Introspection
Memory management

### Definitive Discussion

Programming with Objective-C

## Selector

A selector is the name used to select a method to execute for an object, or the unique identifier that replaces the name when the source code is compiled. A selector by itself doesn't do anything. It simply identifies a method. The only thing that makes the selector method name different from a plain string is that the compiler makes sure that selectors are unique. What makes a selector useful is that (in conjunction with the runtime) it acts like a dynamic function pointer that, for a given name, automatically points to the implementation of a method appropriate for whichever class it's used with. Suppose you had a selector for the method `run`, and classes `Dog`, `Athlete`, and `ComputerSimulation` (each of which implemented a method `run`). The selector could be used with an instance of each of the classes to invoke its `run` method—even though the implementation might be different for each.

### Getting a Selector

Compiled selectors are of type `SEL`. There are two common ways to get a selector:

- At compile time, you use the compiler directive `@selector`.

```
• SEL aSelector = @selector(methodName);
```

-
- At runtime, you use the `NSSelectorFromString` function, where the string is the name of the method:

```
• SEL aSelector = NSSelectorFromString(@"methodName");
```

-

  You use a selector created from a string when you want your code to send a message whose name you may not know until runtime.

### Using a Selector

You can invoke a method using a selector with `performSelector:` and other similar methods.

```
SEL aSelector = @selector(run);

[aDog performSelector:aSelector];

[anAthlete performSelector:aSelector];

[aComputerSimulation performSelector:aSelector];
```

(You use this technique in special situations, such as when you implement an object that uses the target-action design pattern. Normally, you simply invoke the method directly.)

### Prerequisite Articles

Message

### Related Articles

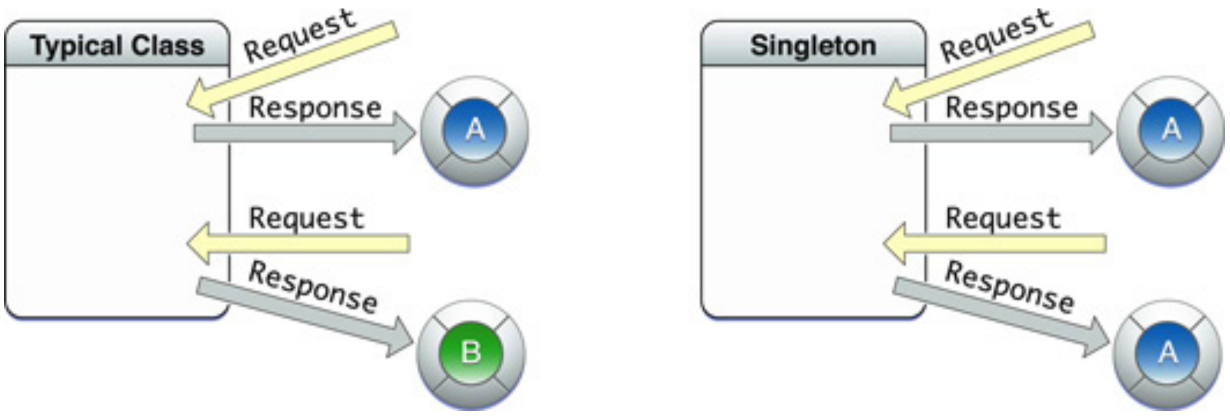
Dynamic binding

### Definitive Discussion

"Working with Objects"

## Singleton

A singleton class returns the same instance no matter how many times an application requests it. A typical class permits callers to create as many instances of the class as they want, whereas with a singleton class, there can be only one instance of the class per process. A singleton object provides a global point of access to the resources of its class. Singletons are used in situations where this single point of control is desirable, such as with classes that offer some general service or resource.



You obtain the global instance from a singleton class through a factory method. The class lazily creates its sole instance the first time it is requested and thereafter ensures that no other instance can be created. A singleton class also prevents callers from copying, retaining, or releasing the instance. You may create your own singleton classes if you find the need for them. For example, if you have a class that provides sounds to other objects in an application, you might make it a singleton.

Several Cocoa framework classes are singletons. They include `NSFileManager`, `NSWorkspace`, and, in UIKit, `UIApplication` and

`UIAccelerometer`. The name of the factory method returning the singleton instance has, by convention, the form `shared`ClassType. Examples from the Cocoa frameworks are `sharedFileManager`, `sharedColorPanel`, and `sharedWorkspace`.

### Prerequisite Articles

Object creation

### Related Articles

Object copying
Memory management

### Definitive Discussion

(None)

## Uniform Type Identifier

A uniform type identifier (UTI) is a string that identifies a class of entities with a type. UTIs are typically used to identify the format for files or in-memory data types and to identify the hierarchical layout of directories, volumes or packages. UTIs are used either to declare the format of existing data or to declare formats that your application accepts. For example, OS X and iPhone applications use UTIs to declare the format for data they place on a pasteboard. Mac apps use UTIs to declare the types of files that they are able to open.

UTIs have several advantages over other type identification schemes:

- The UTI naming convention is logical, and the syntax is well known.
- UTIs can be related in a hierarchical fashion, like a family tree.
- The list of UTIs can be extended by applications, meaning new types and subtypes can be created.
- A UTI declaration includes metadata that describes the type, including a human-readable description, related UTIs, and conversion information to other identification schemes, such as MIME types or filename extensions.

### UTIs Use the Reverse Domain Name System Convention

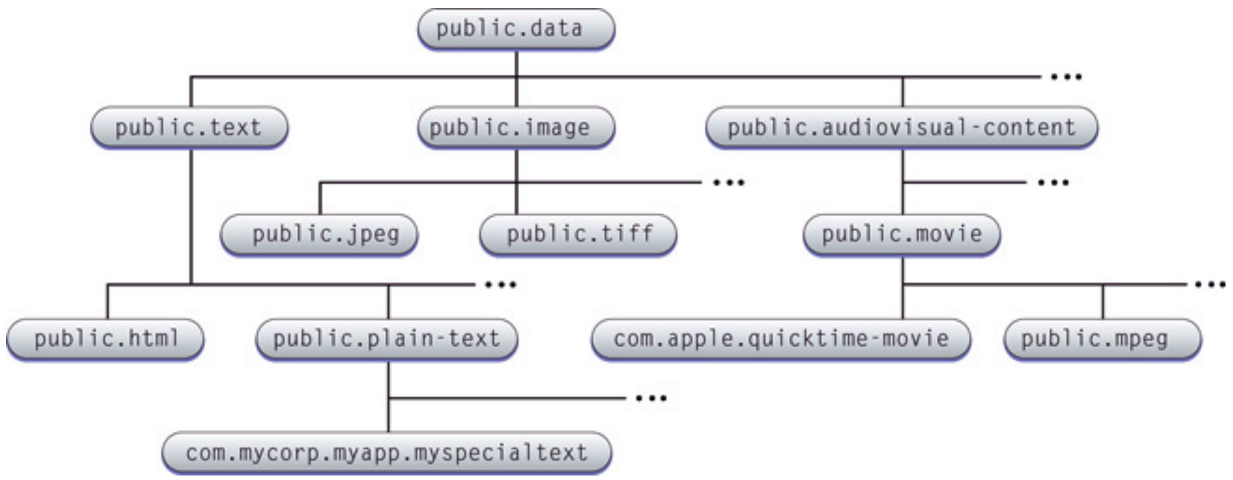
A UTI is defined as a string (`CFString`) that follows a reverse Domain Name System (DNS) convention. The top-level domain (for instance, `com`), comes first, followed by one or more subdomains, and ending in a token that represents the actual type. For example, `com.apple.application` is an abstract base type that identifies applications. Domains are used only to identify a UTI's position in the domain hierarchy; they do not imply any grouping of similar types.

UTIs in the `public` domain are defined by Apple and are used to represent common formats.

UTIs in the `dyn` domain are reserved. They are created automatically as a UTI-compatible wrapper around a type from another identification scheme when no defined conversion to a UTI exists.

## Uniform Type Identifiers Are Declared in a Conformance Hierarchy

A conformance hierarchy is similar to a class hierarchy in object-oriented programming. All instances of a type lower in the hierarchy are also instances of a type higher in the hierarchy.



Conformance gives your application flexibility in declaring the types it is compatible with. Your application specifies what types it can handle, and all subtypes underneath it are automatically included. For example, the UTI `public.html`, which defines HTML text, conforms to the `public.text` identifier. An application that opens text files automatically opens HTML files.

A UTI conformance hierarchy supports multiple inheritance. Most UTIs can trace their conformance information to a **physical** UTI that describes how its physical nature and a **functional** UTI that describes how the data is used.

UTI properties are inherited at runtime. When a value is needed, the hierarchy is searched, starting first with the current type and then through its parent types.

### OS X Applications Add New UTIs by Defining Them in an Application Bundle

Applications add a new UTI to the system by declaring the UTI in their information property list. Declarations include metadata to describe the UTI and its position in the conformance hierarchy.

Application-declared UTIs can be exported or imported. An exported UTI always represents the definitive declaration of the UTI. In contrast, an imported UTI is redeclared by another application. Imported declarations are useful when your application can read a file defined by another application but does not want to require that application to be installed on the target machine. If the operating system finds both an imported and an exported declaration, the exported declaration takes precedence.

### Prerequisite Articles

Bundle
Property list

### Related Articles

(None)

### Definitive Discussion

Uniform Type Identifiers Overview

### Sample Code Projects

(None)

# Value object

A value object is in essence an object-oriented wrapper for a simple data element such as a string, number, or date. The common value classes in Cocoa are `NSString`, `NSDate`, and `NSNumber`. Value objects are often attributes of other custom objects you create.

Value objects offer richer behavior than the corresponding simple scalar types (such as `char`, `NSTimeInterval`, `int`, `float`, or `double`):

- You can put any of the value objects in a collection, such as an instance of `NSArray` or `NSDictionary`.

- Using `NSString`, and its subclass `NSMutableString`, you can perform a wide range of string-related operations. For example, you can join strings together, split strings apart, work on file paths, transform the case of characters, and search for substrings. In all of these, string objects are treated as Unicode.
- Using `NSDate`, in conjunction with `NSCalendar` and other related classes, you can perform complicated calendrical calculations such as determining the number of months and days between two instants in time based on the user's preferred calendar, taking into account variables such as time zones and leap years.
- Using the `NSNumber` subclass `NSDecimalNumber`, you can accurately perform currency-based calculations.

### NSValue

`NSValue` provides a simple container for a single C or Objective-C data item. It can hold any of the scalar types such as `char`, `int`, `float`, or `double`, as well as pointers, structures, and object IDs. It lets you add items of such data types to collections such as instances of `NSArray` and `NSSet`, which require their elements to be objects. This is particularly useful if you need to put point, size, or rectangle structures (such as `NSPoint`, `CGSize`, or `NSRect`) into a collection.

## Prerequisite Articles

(None)