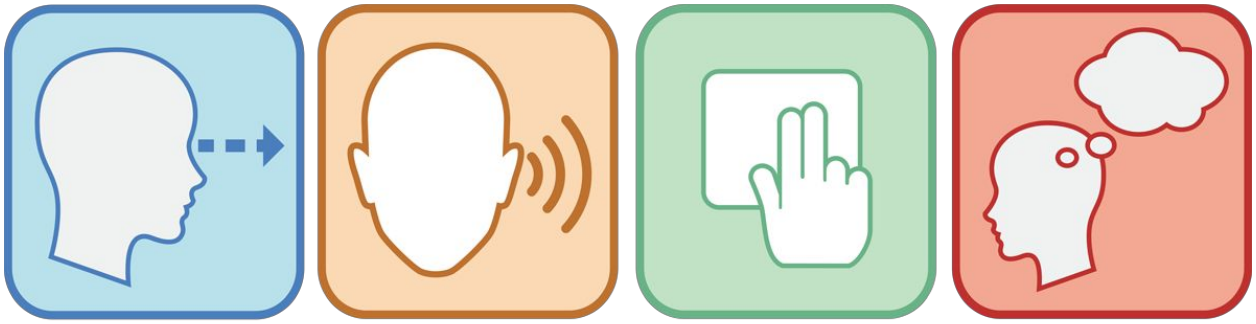
Cocoa Core Competencies

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.
Consider your app's user experience from the perspective of someone with a seeing, hearing, interacting, or learning impairment.



Working with VoiceOver

A screen-reading technology built into the operating system that speaks your app's user interface aloud.The standard UI elements provided by AppKit and UIKit are accessible to VoiceOver by default.
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.
Enable VoiceOver on iOS in Settings > General > Accessibility.

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."
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.

Example:

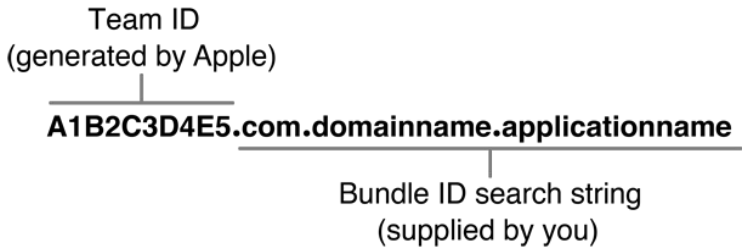
```
- (type)name;
```

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

Cocoa does not use `getName` because methods that start with “get” in Cocoa indicate that the method will return values by reference.

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.



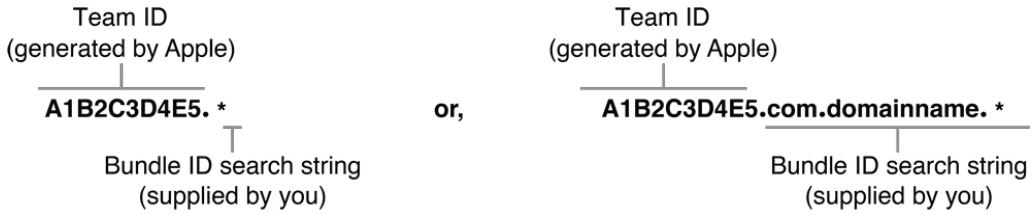
There are two types of App IDs: an *explicit App ID*, used for a single app, and *wildcard App IDs*, used for a set of apps.

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.



Block object

Blocks are a language feature added to C, C++ and ObjectiveC. Which allows us to create distinct segment of code that can be passed around to methods as they were values.

Blocks are objective C objects that means they can be added to collections like NSArray or NSDictionary.

Declaring a Block

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

Creating a Block

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

oneFrom = ^(int anInt) {
   return anInt - 1;
};
```

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.

Using Blocks

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

Example:

```
__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
```

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.
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.

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`).

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.
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.

Category

You use categories to define additional methods of an existing class—even one whose source code is unavailable to you—without subclassing.

- 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.

Declaration

```
#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.

```
#import "SystemClass+CategoryName.h"
@implementation SystemClass ( CategoryName )
// method definitions
@end
```

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`.

Benefits

The benefit of a class cluster is primarily efficiency. Moreover, the code you write will continue to work even if the underlying implementation changes.

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.

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.

```
#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
```

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,

```
[MyClass classMethod];
```

Subclasses

A subclass is a class that inherited by another class.

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

An instance variable is unique to a class. By default, only the class and subclasses can access it. Therefore, as a fundamental principal of object-oriented programming, instance variables (ivars) are private—they are encapsulated by the class.

Property Variables

A property is a public value that can be accessed from outside of class using class object.

Class Variables

Class variables, however, only have **one** copy of the variable(s) shared with all instances of the class. It's important to remember that **class variables are also known as static member variables.**

self

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

```
+ (id)myClass {
   return [[[self alloc] init] autorelease];
}
```

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.

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 use the UIKit frameworks for developing an application's user interface. 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.

Cocoa and Cocoa Touch applications may include C++ and ANSI C code. We can use scripting languages as PyObjc and RubyCocoa also.

Collection

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

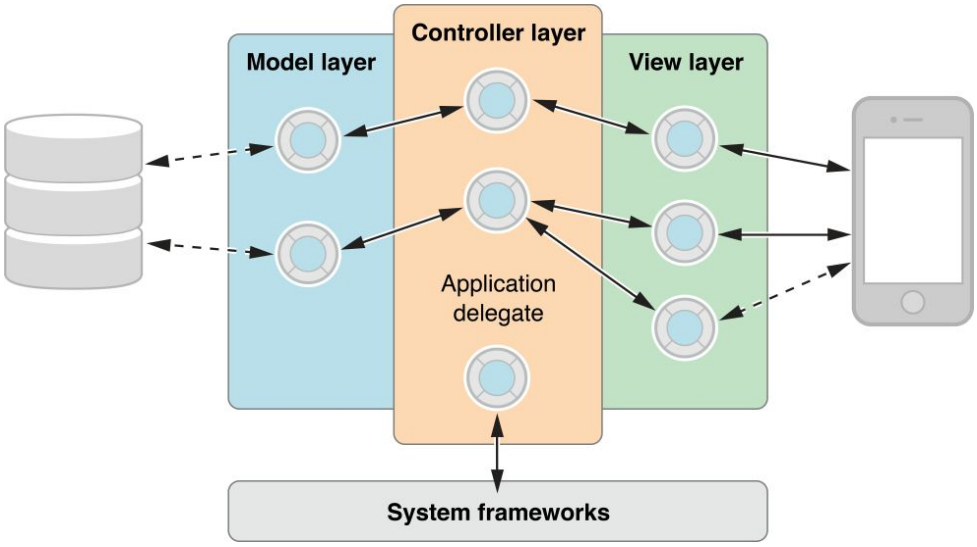
Collection Classes

The primary classes—`NSArray`, `NSSet`, and `NSDictionary`.

- They can hold only objects, but the objects can be of any type.
- 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.

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 and communicates new or changed data to the model objects.



Controller objects can also set up and coordinate tasks for an applicat.ion and manage the life cycles of other objects.

Coordinating Controllers

- 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

View Controllers

In UIKit, a view controller manages a view displaying a screenful of content. The controller manages the presentation of this view. A view controller in iOS is an instance of a subclass of `UIViewController`.

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 main value of delegation is that it allows you to easily customize the behavior of several objects in one central object.

Data Source

A data source is almost identical to a delegate. Instead of being delegated control of the user interface, a data source is delegated control of data.
A data source, like a delegate, must adopt a protocol and implement at minimum the required methods of that protocol.

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.

For example,

```
NSArray *anArray = [NSArray arrayWithObjects:aDog, anAthlete, aComputerSimulation, nil];
id anObject = [anArray objectAtIndex:(random()/pow(2, 31)*3)];
[anObject run];
```

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*.

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 dynamic typing gives your program much greater flexibility.

```
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 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.

Enumeration

Enumeration is the process of sequentially operating on elements of an object—typically a collection. Enumeration is also referred to as *iteration*.

```
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 */
 }
```

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 */
 }
```

Fast Enumeration

Fast enumeration is a language feature that helps us to enumerate through objects of collections like NSArray, NSDictionary etc.

```
 NSArray *anArray = // get an array;
 for (id element in anArray) {
```

```
    /* code that acts on the element */
}
```

Exception handling

Exception handling is the process of managing atypical events (such as unrecognized messages) that interrupt the normal flow of program execution.

Handling Exceptions Using Compiler Directives

- 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.

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

@try {

    [cup fill];

}

@catch (NSException *exception) {

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

```
}
@finally {
    [cup release];
}
```

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.

iPhone application projects link by default to the Foundation, UIKit, and Core Graphics frameworks.

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.

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, searching or listing application on Home screen. iOS use these information property lists of installed application.

The name of an information property list file of any application's scheme must be `Info.plist.`

Initialization

Initialization is the stage of object creation that makes a newly allocated object usable by setting its reasonable initial values. Initialization should always occur right after allocation.

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

Implementing an Initializer

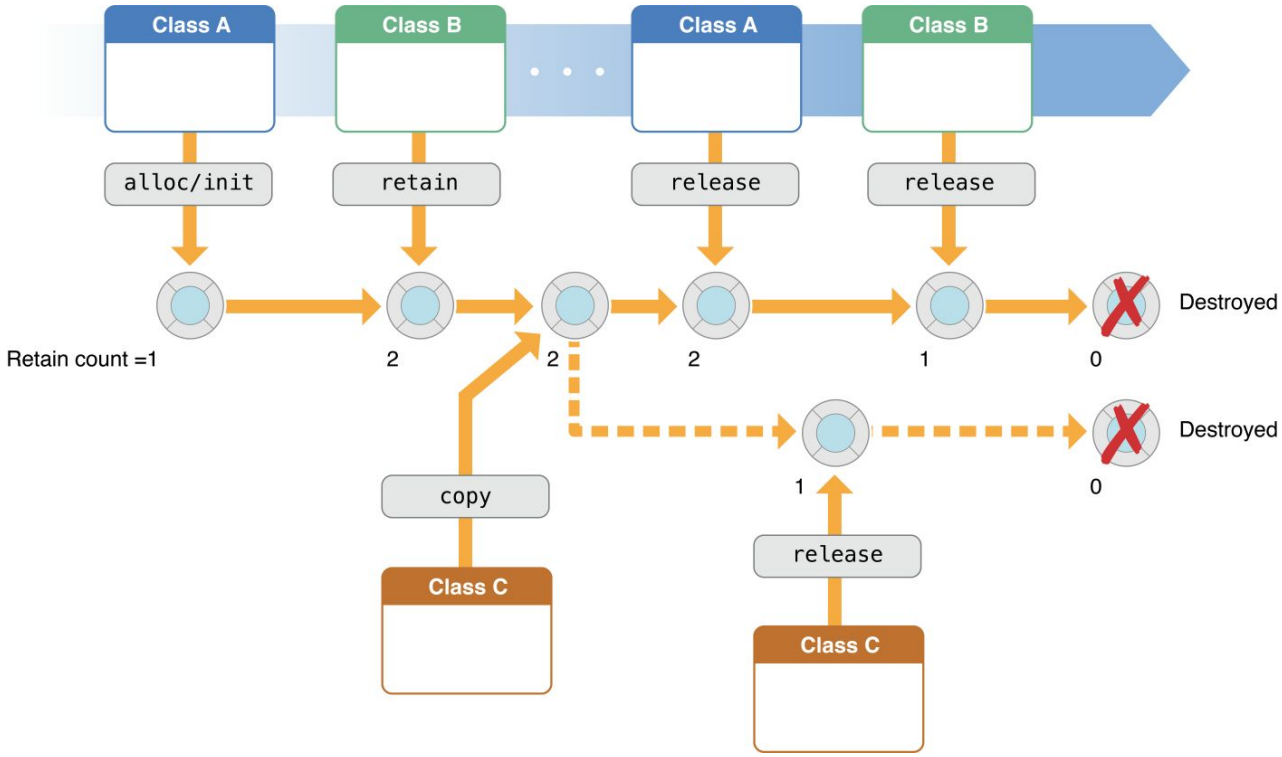
1. Invoke the superclass initializer and check the value it returns.
2. Assign values to the object's instance variables.
3. Return the initialized object or, if initialization did not succeed, return `nil`.

```
- (id)init {
   if (self = [super init]) { // equivalent to "self does not equal nil"
      date = [[NSDate date] retain];
   }
   return self;
}
```

Memory management

Memory management is the programming discipline of managing the life cycles of objects and freeing them when they are no longer needed.
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. The owners of an object may also give up their ownership interest in it, which decrements the retain count. When the retain count becomes zero, the object is deallocated (destroyed).

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, 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. Example: objects created by iOS default Factory methods. As NSString or NSArray factory methods.

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.

Message

A message is the name of a method with or without any parameters associated with it, that are sent to an object to do something.

```
[myRectangle display];
```

When a message is sent, the runtime system selects the appropriate method from the receiver invokes it. For this reason, method names in messages are often referred to as *selectors*.

Methods can also take parameters, also called *arguments*.

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.

```
@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.

If you might want to extend a superclass's implementation. To do this, you can invoke the superclass's implementation using the `super` keyword.

```
@implementation MySubclass
- (int)myNumber {
   int subclassNumber = [super myNumber] + 1;
   return subclassNumber;
}
@end
```

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.

Any data that is part of the persistent state of the application should reside in the model objects after the data is loaded into the application.

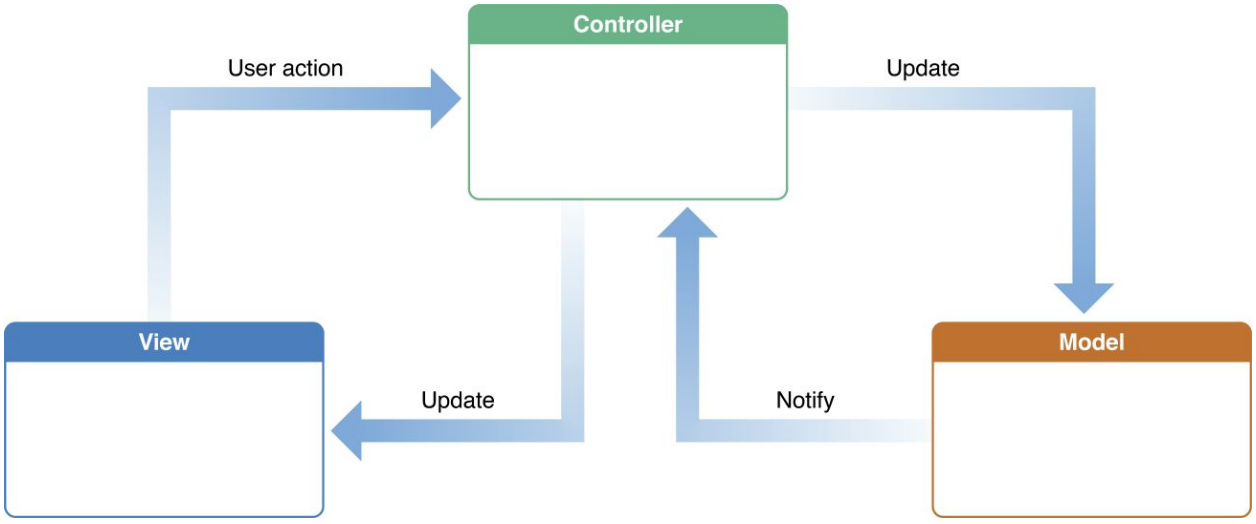
A Well-Designed Model Class

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`.

- **Instance variables.** You declare instance variables to hold the data of an application. Instance variables can be objects, scalar values, or structures such as `NSRange`.
- **Accessor methods and declared properties.** Accessor methods and declared properties provide ways to maintain encapsulation. Accessor methods typically get and set the values of instance variables (and are colloquially known as *getter* and *setter* methods).
- **Key-value coding.** Key-value coding is a mechanism that lets other objects access an object's property using the property name as a key. It is used by Core Data and elsewhere in Cocoa.
- **Initialization and deallocation.** In most cases, a model class implements an initializer method to set its instance variables to reasonable initial values. 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.

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.



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.

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.

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.

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.

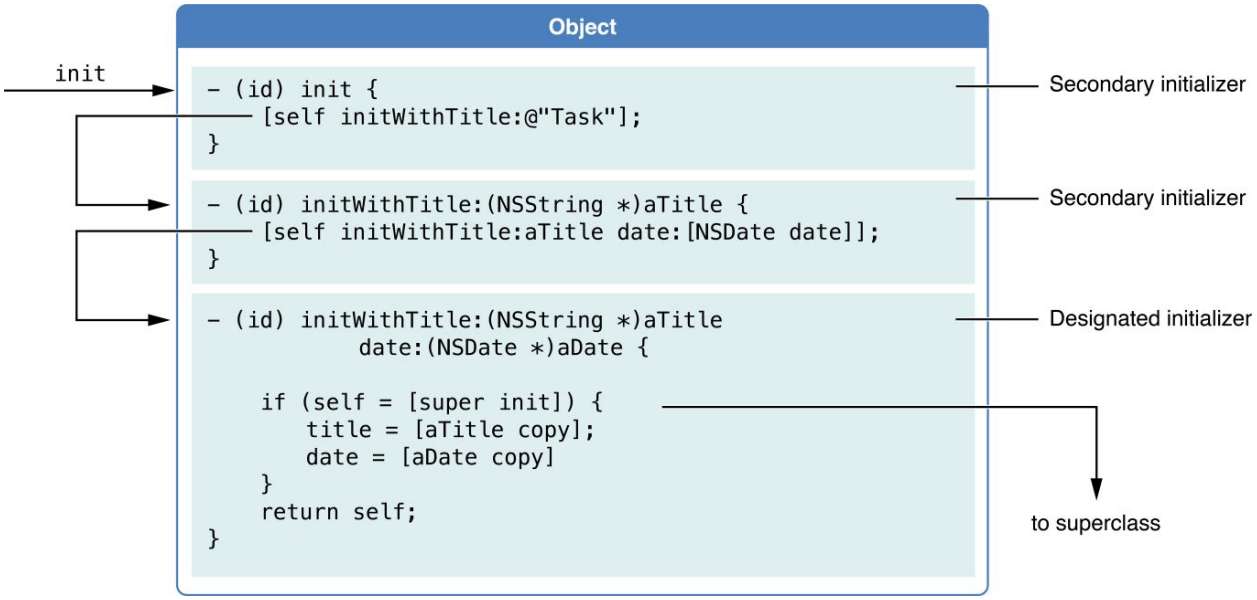
Controller Objects

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.

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.

Multiple initializers

A class may define multiple initializer methods according to requirement.



Nib Files

Nib files play an important role in the creation of applications in OS X and iOS. With nib files, you create and manipulate your user interfaces graphically, using Xcode, instead of programmatically.

For applications built using the AppKit or UIKit frameworks, both of these frameworks support the use of nib files both for laying out windows, views, and controls and for integrating those items with the application's event handling code. Xcode works in conjunction with these frameworks to help you connect the controls of your user interface to the objects in your project that respond to those controls

About the File's Owner

File's Owner object is a placeholder object that is not created when the nib file is loaded. Instead, you create this object in your code and pass it to the nib-loading code. The reason this object is so important is that it is the main link between your application code and the contents of the nib file.

```
[[NSBundle mainBundle] loadNibNamed:@"ViewController" owner:[UIViewController new] options:nil];
```

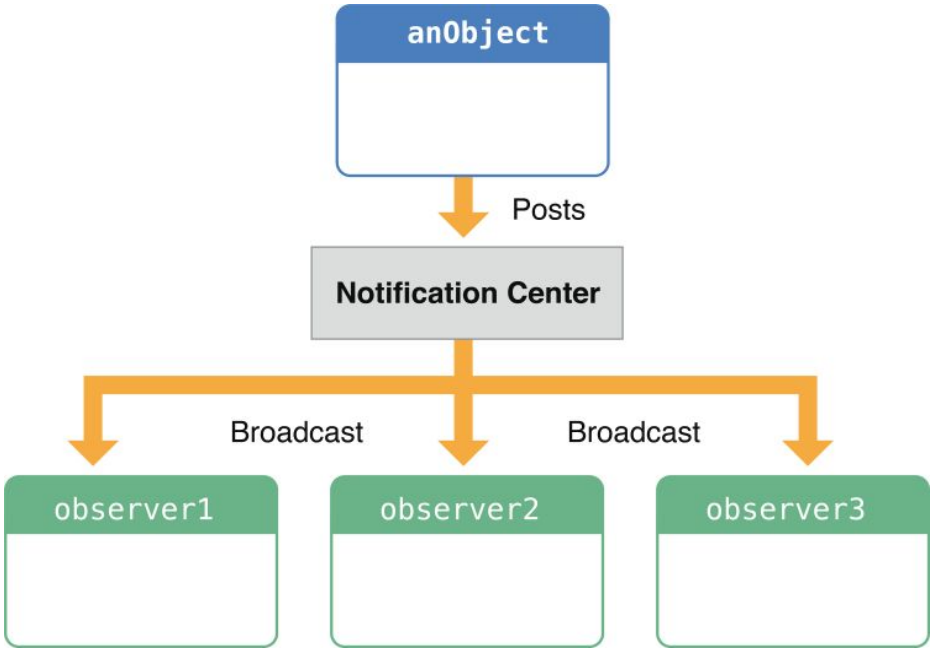
Notification

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.

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 in a program.

The centerpiece of the notification mechanism is a per-application singleton object known as the notification center (`NSNotificationCenter`). Although the notification center delivers a notification to its observers synchronously, you can post notifications asynchronously using a notification queue (`NSNotificationQueue`).




```
[[NSNotificationCenter defaultCenter] addObserver:nil selector:@selector(stringTest) name:nil object:nil];

[[NSNotificationCenter defaultCenter] postNotificationName:@"" object:nil userInfo:nil];
```

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`.

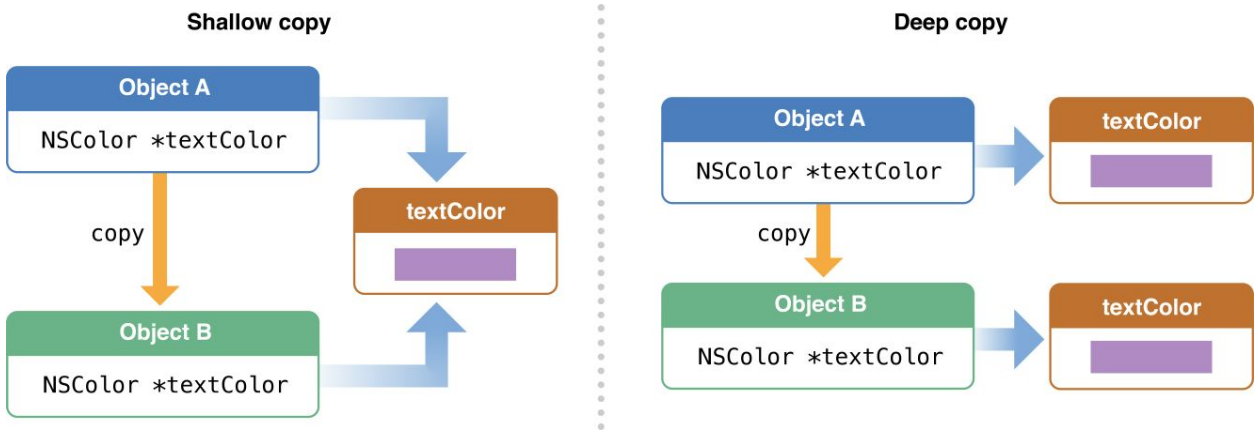
Some classes of the Foundation framework implement comparison methods of the form `isEqualTo`*Type*`:`—for example, `isEqualToString:` and`isEqualToArray:`. These methods perform comparisons specific to the given class type.

Object copying

Copying an object creates a new object with the same class and properties as the original object.

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.



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`.

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.

Factory Methods

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.

Object graph

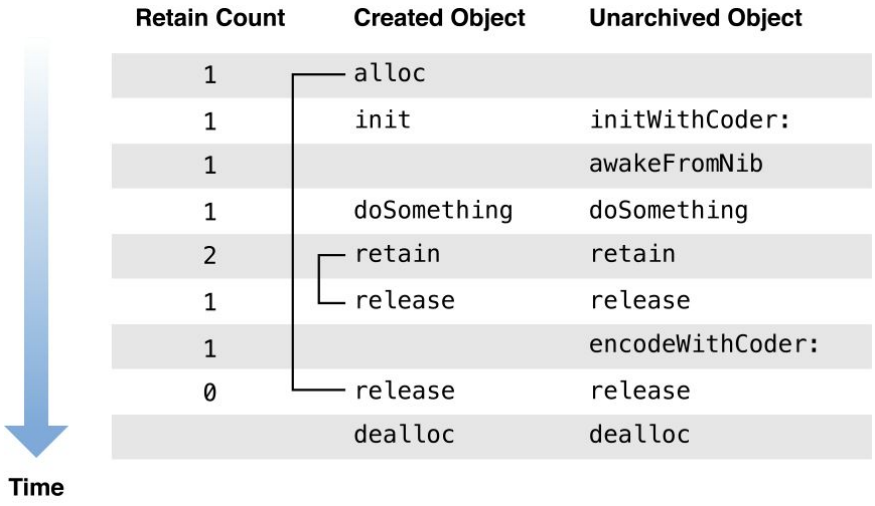
In an object-oriented program, groups of objects form a network through their relationships with each other—either through a direct reference to another object or through a chain of intermediate references. These groups of objects are referred to as object graphs. Object graphs may be **small or large, 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.

Object life cycle

An object's life cycle—that is, its runtime life from its creation to its destruction. An object comes into existence when a program explicitly allocates and initializes it or when it makes a copy of another object.



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 give up that ownership by sending `release` to the object.

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.

Object modeling

Object modeling is the process of designing the objects or classes through which an object-oriented application examines and changes some service.

Typically, the characteristics of a class should make sense. As the names of the class, its variables, and its methods should be understandable to a non-programmer. So that a non-programmer can get an idea about what is this object doing.

Object mutability

Most Cocoa objects are mutable—meaning you can change their values—but some are immutable, and you cannot change their values after they are created.

So immutable objects (As NSArray or NSDictionary) are thread-safe but mutable objects are not.

We will get a warning if we are trying to assign (shallow copy) a immutable object into mutable object.

Object ownership

Objective-C uses reference counting to manage the memory of its objects through the concept of object ownership. When compiling code with the ARC feature (enabled by default), the compiler takes the references you create and automatically inserts calls to the underlying memory management mechanism.

There are two types of object reference:

- Strong references, which keep an object “alive” in memory.
- Weak references, which have no effect on the lifetime of a referenced object.

A strong reference ensures that the referenced object remains in memory (that is, it does not get deallocated) for as long as the reference is valid. When there are no remaining strong references to an object, the object gets deallocated.

A weak reference has no effect on the lifecycle of the object it refers to: when no strong references to the object remain, even if there are still weak references, the object gets deallocated, and any weak references to the object are set to `nil`.

To create a weak reference, use the `__weak` qualifier as follows:

```
NSString * __weak someWeakReferenceToAString;
```

Strong references are not always the right choice. For example, if two objects need to refer to each other, making both references strong would create a **strong reference cycle**, preventing either object from ever being deallocated. In these cases, you use weak references.

For example: A view that presents data maintains weak references to its delegate and data source; the view is owned by some other object, usually a view controller, which commonly acts as the delegate and data source. This controller has a strong reference to view.

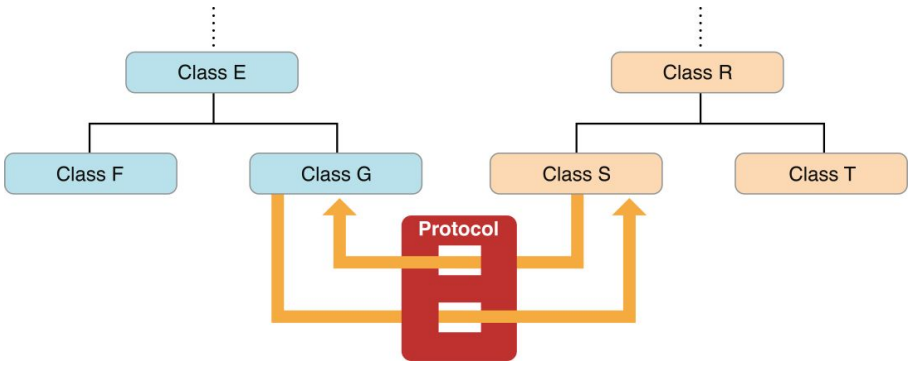
Objective-C

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.

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.



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.

It also declares the fundamental object interface and implements basic object behavior, including introspection, memory management, and method invocation.

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

Selector

A selector is the name used to select a method to execute for an object. A selector by itself doesn't do anything. It simply identifies a method. Selector acts like a dynamic function pointer that automatically points to the implementation of a method appropriate for whichever class it's used with.

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");
```

Using a Selector

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

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

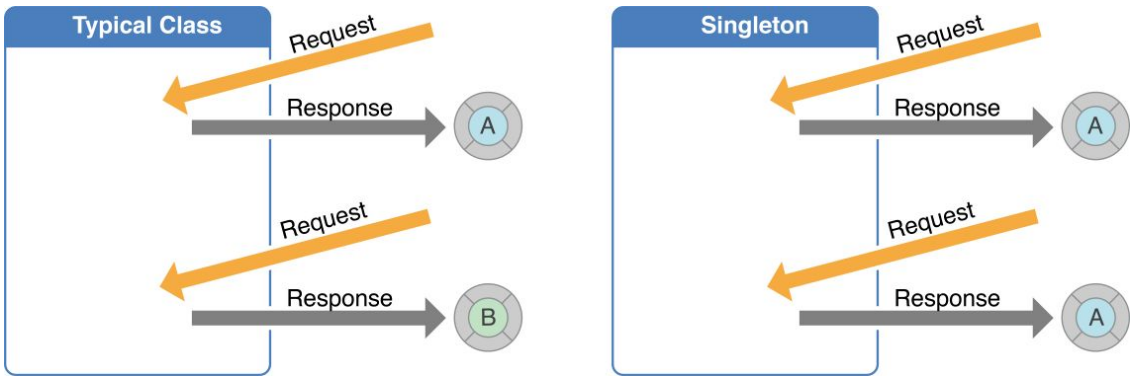
```
[aDog performSelector:aSelector];
```

```
[anAthlete performSelector:aSelector];
```

```
[aComputerSimulation performSelector:aSelector];
```

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.

Several Cocoa framework classes are singletons. They include `NSFileManager`, `UIApplication` and `UIAccelerometer etc.`

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.

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.