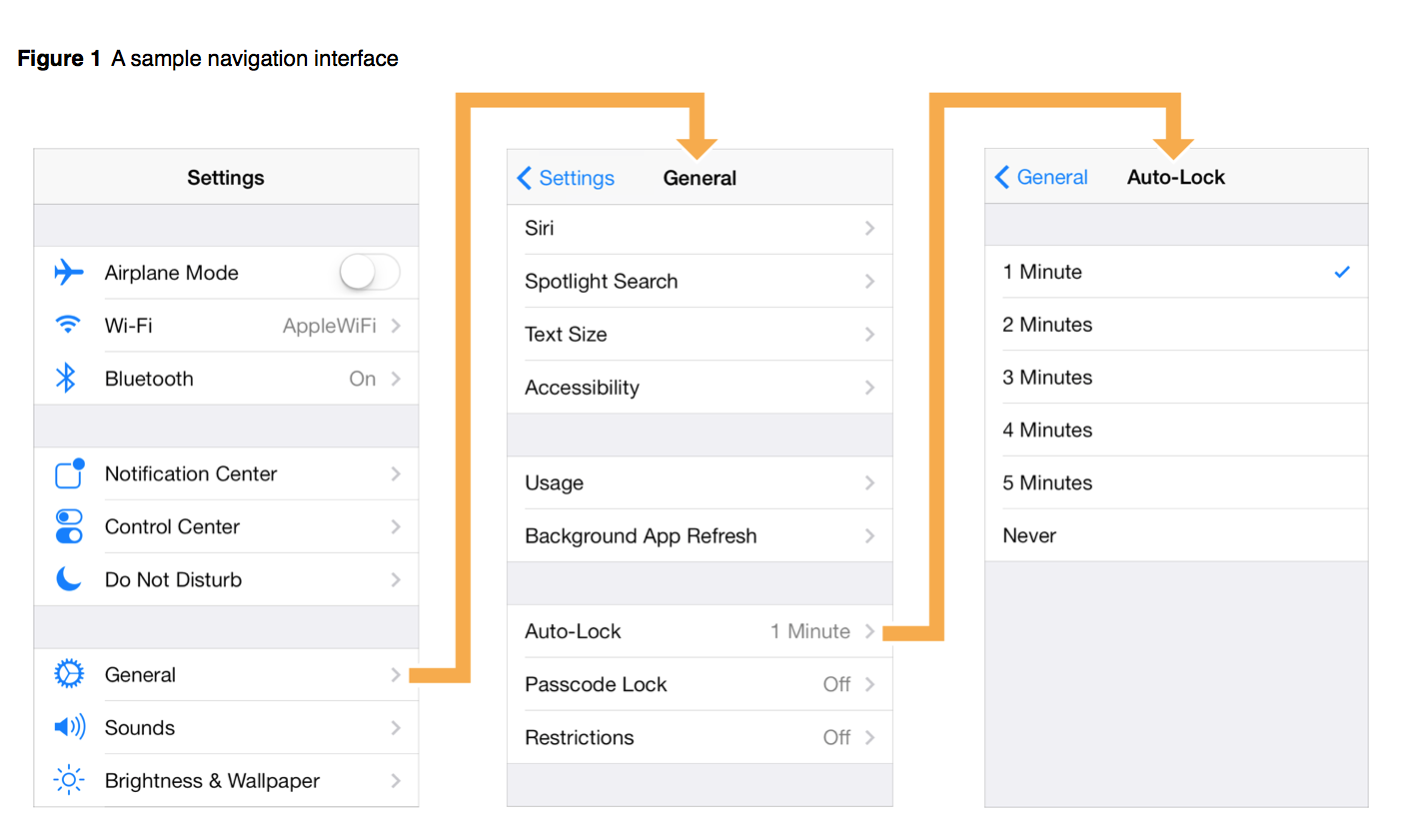
UINavigationController

The UINavigationController class implements a specialized view controller that manages the navigation of hierarchical content.

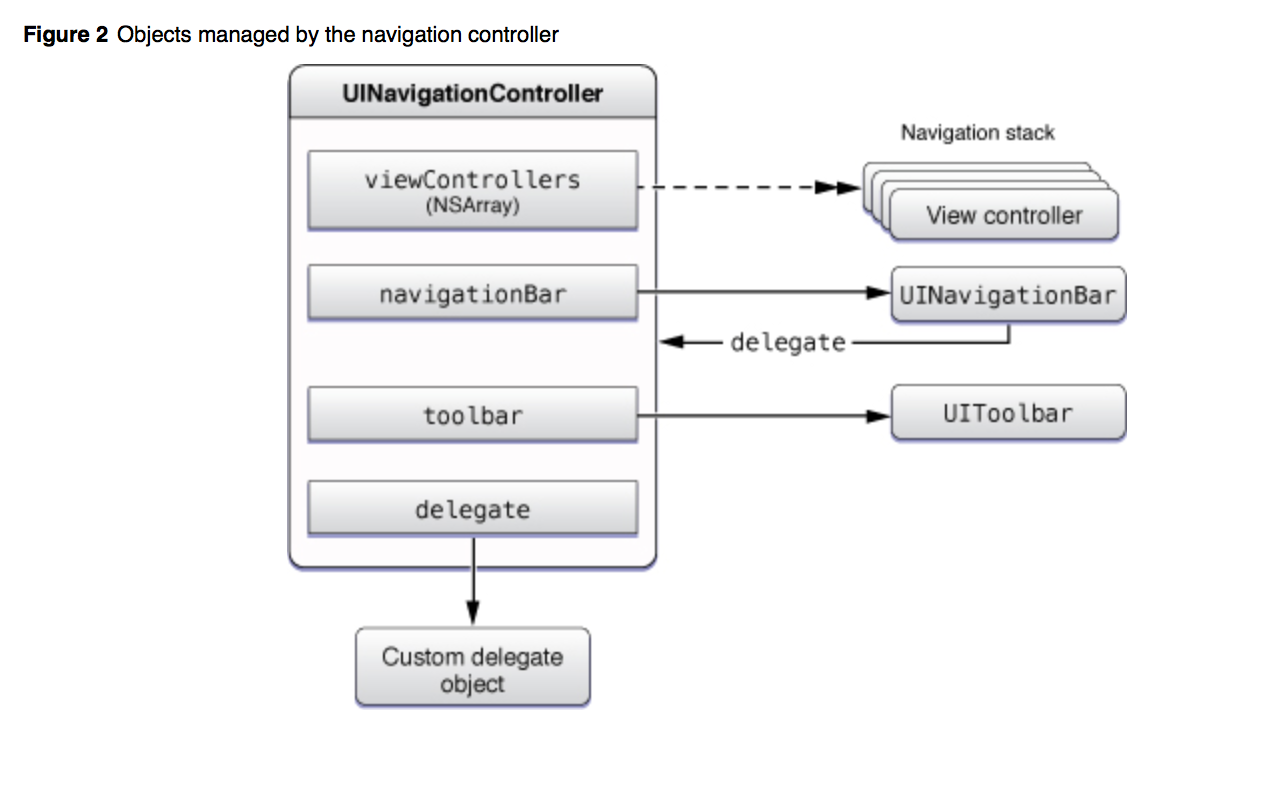


Use of Navigation Bar

Navigation bar contains the navigation buttons of a navigation controller, which is a stack of view controllers which can be pushed and popped. Title on the navigation bar is the title of the current view controller.

Navigation Controller Views

A navigation controller is a container view controller—that is, it embeds the content of other view controllers inside of itself. You access a navigation controller’s view from its [view](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/index.html#//apple_ref/occ/instp/UIViewController/view) property. This view incorporates the navigation bar, an optional toolbar, and the content view corresponding to the topmost view controller.



XIB stands for the XML Interface Builder.

Interface Builder is a software application which allows you to develop Graphical User Interface with the help of Cocoa and carbon. The generated files are either stored as NIB or XIB files. These files are copied into the app bundle and loaded at run time to provide the user interface for the application. XIB files were introduced in 2007 with Leopard (Xcode 3.0).

Apple introduced the concept of "storyboarding" in iOS5 SDK to simplify and better manage screens in your app. You can still use the .xib way of development.

Pre-storyboard, each UIViewController had an associated .xib with it. Storyboard achieves two things:

* .storyboard is essentially one single file for all your screens in the app and it shows the flow of the screens. You can add segues/transitions between screens, this way. So, this minimizes the boilerplate code required to manage multiple screens.

Minimizes the overall no. of files in an app.

**Storyboard**

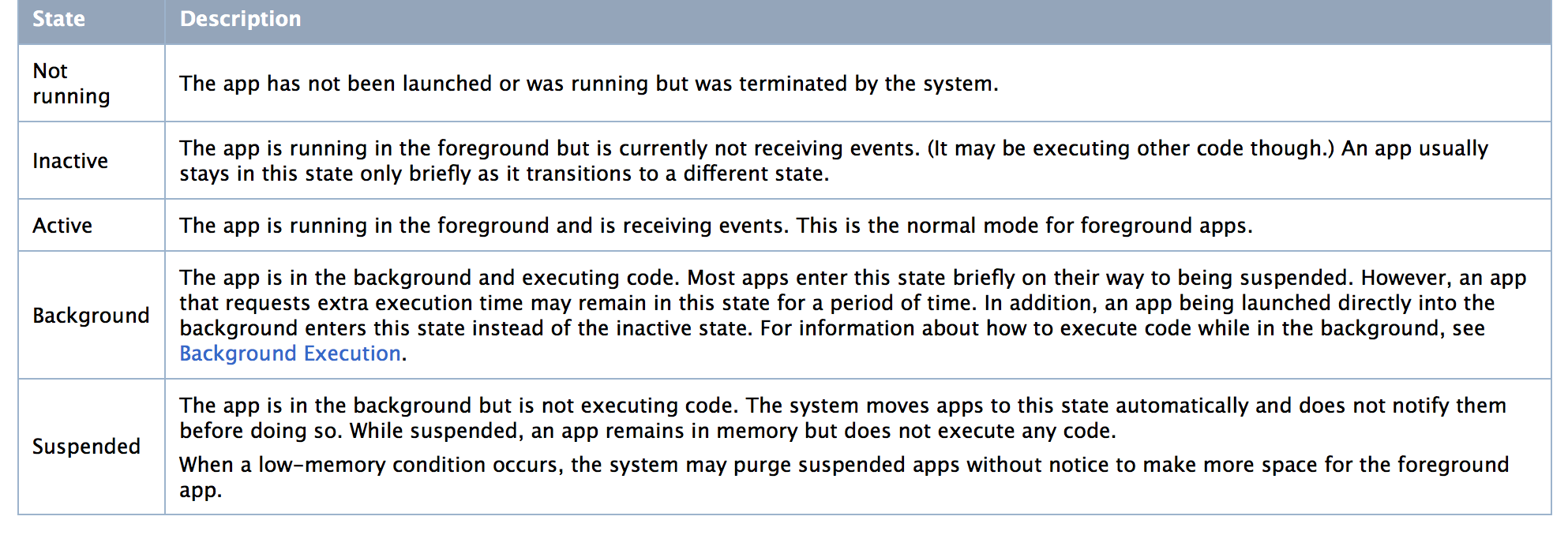
Storyboards are an exciting feature first introduced way back in iOS 5 that save you a lot of time building user interfaces for your apps.

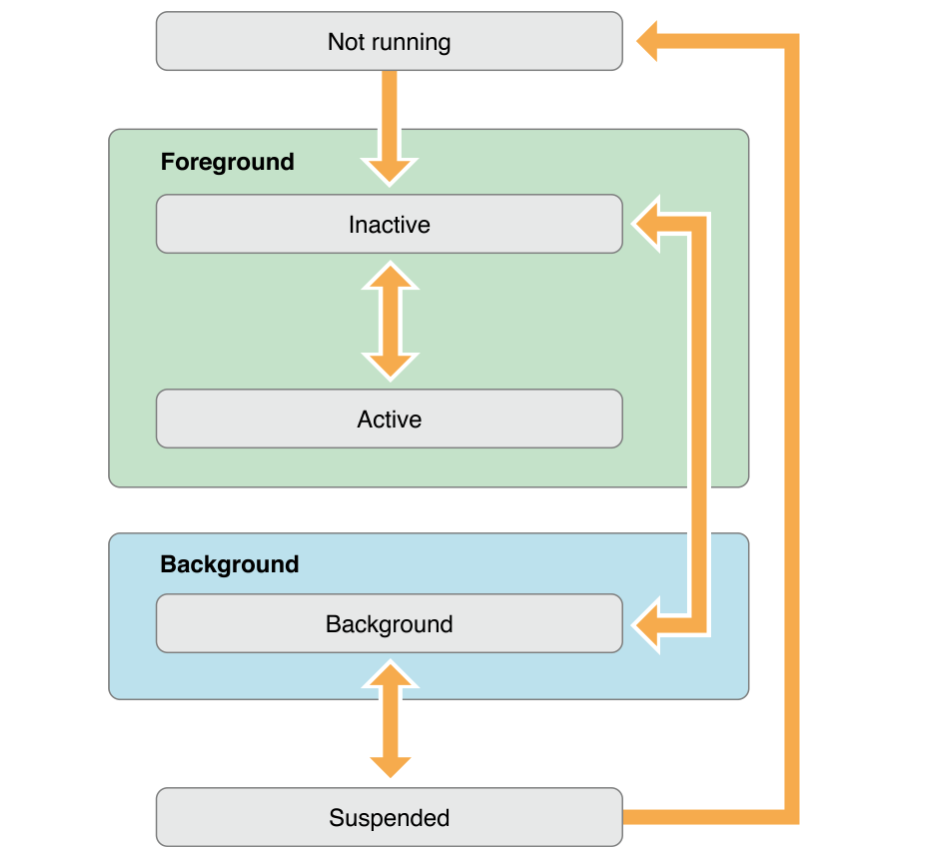
Storyboards have a number of advantages:

* You can visually lay out all your view controllers in “scenes” and describe the connections between them. With a storyboard you have a better conceptual overview of all the scenes in your app.
* Storyboards can describe the transitions between the various scenes. These transitions are called “segues” and you create them by connecting your view controllers right in the storyboard. Thanks to segues you need less code to take care of your UI.
* Storyboards make working with table views a lot easier with prototype cells and static cells features. You can design your table views almost completely in the storyboard editor, cutting down on the amount of code you have to

Execution States for Apps

App states





**What is the** **UserDefaults Class?**

With the UserDefaults class, you can save settings and properties related to application or user data. For example, you could save a profile image set by the user or a default color scheme for the application. The objects will be saved in what is known as the iOS “defaults system”. The iOS defaults system is available throughout all of the code in your app, and any data saved to the defaults system will persist through application sessions. This means that even if the user closes your application or reboots their phone, the saved data will still be available the next time they open the app!

With NSUserDefaults you can save objects from the following class types:

* NSData
* NSString
* NSNumber
* NSDate
* NSArray

NSDictionary

Understanding Auto Layout

Auto Layout dynamically calculates the size and position of all the views in your view hierarchy, based on constraints placed on those views.

It is a new way to define dynamin GUIs. With Autolayout you can do more complicated GUIs quite easy.

What is the basic difference between Auto Layout and Auto Resizing in iOS

Autoresizing is a matter of conceptually assigning a subview “springs and struts.” A spring can stretch; a strut can’t. Springs and struts can be assigned internally or externally. Thus you can specify (using internal springs and struts) whether and how the view can be resized, and (using external springs and struts) whether and how the view can be repositioned.

And

Autolayout, depends on the constraints of views. A constraint (an instance of NSLayoutConstraint) is much more sophisticated than the "autoresizingMask" it’s a full-fledged object with numeric values, and can describe a relationship between any two views (not just a subview and its superview).

<http://www.sm-cloud.com/interview-questions-3-commonly-asked-view-and-ui-ios-questions/>

**Frame vs Bounds**

The **bounds** of an [UIView](http://developer.apple.com/iPhone/library/documentation/UIKit/Reference/UIView_Class/UIView/UIView.html) is the [rectangle](http://developer.apple.com/iPhone/library/documentation/GraphicsImaging/Reference/CGGeometry/Reference/reference.html#//apple_ref/doc/c_ref/CGRect), expressed as a location (x,y) and size (width,height) relative to its own coordinate system (0,0).

The **frame** of an [UIView](http://developer.apple.com/iPhone/library/documentation/UIKit/Reference/UIView_Class/UIView/UIView.html) is the [rectangle](http://developer.apple.com/iPhone/library/documentation/GraphicsImaging/Reference/CGGeometry/Reference/reference.html#//apple_ref/doc/c_ref/CGRect), expressed as a location (x,y) and size (width,height) relative to the superview it is contained within.

So, imagine a view that has a size of 100x100 (width x height) positioned at 25,25 (x,y) of its superview. The following code prints out this view's bounds and frame:

NSLog(@"bounds.origin.x: %f", label.bounds.origin.x);

NSLog(@"bounds.origin.y: %f", label.bounds.origin.y);

NSLog(@"bounds.size.width: %f", label.bounds.size.width);

NSLog(@"bounds.size.height: %f", label.bounds.size.height);

NSLog(@"frame.origin.x: %f", label.frame.origin.x);

NSLog(@"frame.origin.y: %f", label.frame.origin.y);

NSLog(@"frame.size.width: %f", label.frame.size.width);

NSLog(@"frame.size.height: %f", label.frame.size.height);

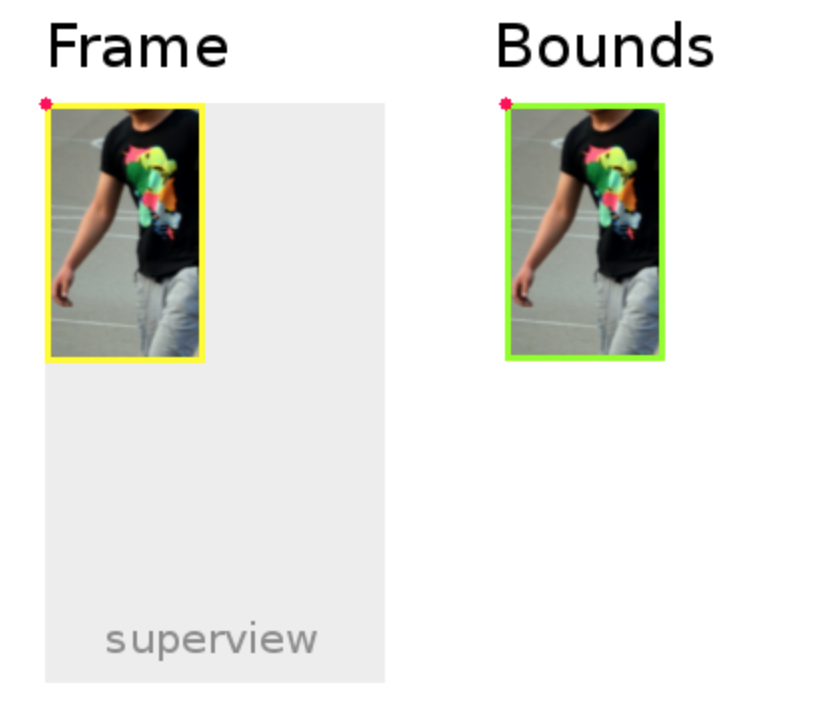
}

**frame** = a view's location and size using the **parent view's coordinate system**

* Important for: placing the view in the parent

**bounds** = a view's location and size using **its own coordinate system**

Important for: placing the view's content or subviews within itself



UITabBarController

The UITabBarController class implements a specialized view controller that manages a radio-style selection interface. This *tab bar interface* displays tabs at the bottom of the window for selecting between the different modes and for displaying the views for that mode. This class is generally used as-is but may be subclassed in iOS 6 and later.

Each tab of a tab bar controller interface is associated with a custom view controller. When the user selects a specific tab, the tab bar controller displays the root view of the corresponding view controller, replacing any previous views. (User taps always display the root view of the tab, regardless of which tab was previously selected. This is true even if the tab was already selected.) Because selecting a tab replaces the contents of the interface, the type of interface managed in each tab need not be similar in any way. In fact, tab bar interfaces are commonly used either to present different types of information or to present the same information using a completely different style of interface.

Use of Table View

It is used for displaying a vertically scrollable view which consists of a number of cells (generally reusable cells). It has special features like headers, footers, rows, and section.

Important Properties

* delegate
* dataSource
* rowHeight
* sectionFooterHeight
* sectionHeaderHeight
* separatorColor
* tableHeaderView
* tableFooterView

Important Methods

- (UITableViewCell \*)cellForRowAtIndexPath:(NSIndexPath \*)indexPath

- (void)deleteRowsAtIndexPaths:(NSArray \*)indexPaths

withRowAnimation:(UITableViewRowAnimation)animation

- (id)dequeueReusableCellWithIdentifier:(NSString \*)identifier

- (id)dequeueReusableCellWithIdentifier:(NSString \*)identifier

forIndexPath:(NSIndexPath \*)indexPath

- (void)reloadData

- (void)reloadRowsAtIndexPaths:(NSArray \*)indexPaths

withRowAnimation:(UITableViewRowAnimation)animation

- (NSArray \*)visibleCells

down vote

accepted

**UITableViewDelegate**

Serving as a table's delegate means you provide answers to requests about the **layout of the table** and about **actions the**

**user performs** on the tableview. **Layout** methods include the tableview asking about the height of rows, headers, and footers, what the buttons should look like, etc. **Action** methods include the user selecting a row and beginning and ending the editing of a row.

**UITableViewDatasource**

Serving as a table's datasource means you **provide data for the sections and rows of a table** and you **act on messages that change a table's data**. The datasource is asked for the data for a cell when the table is drawn, is told that the user has asked to delete a row, and is told the new value of a row that the user has edited.

**Plist**

An information property list file is a structured text file that contains essential configuration information for a bundled executable. The file itself is typically encoded using the Unicode UTF-8 encoding and the contents are structured using XML. The root XML node is a dictionary, whose contents are a set of keys and values describing different aspects of the bundle. The system uses these keys and values to obtain information about your app and how it is configured. As a result, all bundled executables (plug-ins, frameworks, and apps) are expected to have an information property list file.

By convention, the name of an information property list file is Info.plist. This name of this file is case sensitive and must have an initial capital letter I. In iOS apps, this file resides in the top-level of the bundle directory. In OS X bundles, this file resides in the bundle’s Contents directory. Xcode typically creates this file for you automatically when you create a project of an appropriate type.

Creating and Editing an Information Property List File

The simplest way to create an information property list file is to let Xcode create it for you. Each new bundle-based project that you create in Xcode comes with a file named <project>-Info.plist, where <project> is the name of the project. At build time, this file is used to generate the Info.plist file that is then included in the resulting bundle.

To edit the contents of your information property list file, select the <project>-Info.plist file in your Xcode project to display the property list editor. Figure 1 shows the editor for the information property list file of a new Cocoa app project. The file created by Xcode comes preconfigured with keys that every information property list should have.

In the context of iPhone development, Property Lists are a key-value store that your application can use to save and retrieve persistent data.

All iPhone applications have at least one of these by default, the Information Property List:

The information property list is a file named Info.plist that is included with every iPhone application project created by Xcode. It is a property list whose key-value pairs specify essential runtime-configuration information for the application. The elements of the information property list are organized in a hierarchy in which each node is an entity such as an array, dictionary, string, or other scalar type.

For big amount of data, sqlite or coreData are better because all data are not load in memory to access to one.

With pList, entire file is read in memory. After, you can retain only data what you want, but it is not optimized.

http://www.ongraph.com/blog/push-notification-in-ios

**Delegate &Notification**

In IOS application development, delegates and notification have its own importance.

A delegate uses protocols and creates a has-a relationship between the two classes. One of the other benefits of delegates is that you can return something back to the owning class.

Notifications, on the other hand, are more geared towards point to multipoint communication.

Delegates create relationships between classes. Notifications are used to send events to one or many classes.we have to use delegate to specified known object.notification for all object.

In iOS these are two ways to pass messages between objects.

1.Using Delegates

2. Using Notifications

The main difference between delegates and Notifications are :

* By using delegates, *one* designated object can receives a message.
* *By Using notifications any number* of objects can receive a message.

Delegates are usually implemented using protocols: a class will usually have something like

@property (weak) id<MyCustomDelegate> delegate;

which gives the delegate a certain set of methods to implement. You can use

myObject.delegate = /\* some object conforming to MyCustomDelegate \*/;

and then the object can send relevant messages to its delegate. For a good common example, see the [UITableViewDelegate protocol](http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDelegate_Protocol/Reference/Reference.html).

Formal and Informal Protocols

There are two varieties of protocol, formal and informal:

* 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.
* 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.

**Notification**

**Notification** Center is a feature in **iOS** and OS X that provides an overview of alerts from applications. It displays **notifications** until the user completes an associated action, rather than requiring instant resolution. Users may choose what applications appear in **Notification** Center, and how they are handled.

Notifications, on the other hand, are implemented using [NSNotificationCenter](http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSNotificationCenter_Class/Reference/Reference.html). An object (or more than one object) simply adds itself as an observer for specific notifications, and then can receive them when they are posted by another object.

[[NSNotificationCenter defaultCenter] addObserver:self

selector:@selector(notificationHappened:)

name:MyCustomNotificationName

object:nil];

Then just implement

- (void)notificationHappened:(NSNotification \*)notification {

// do work here

}

And you can post notifications from anywhere using

[[NSNotificationCenter defaultCenter] postNotificationName:MyCustomNotificationName

object:self

userInfo:nil];

And make sure to call removeObserver: when you're done!

Local and Remote Notifications in Depth

The essential purpose of both local and remote notifications is to enable an app to inform its users that it has something for them—for example, a message or an upcoming appointment—when the app isn’t running in the foreground. The essential difference between local notifications and remote notifications is simple:

* Local notifications are scheduled by an app and delivered on the same device.
* Remote notifications, also known as push notifications, are sent by your server to the Apple Push Notification service, which pushes the notification to devices.

**NSOperation vs. Grand Central Dispatch (GCD)**

You may have heard of [Grand Central Dispatch (GCD)](http://developer.apple.com/library/ios/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html). In a nutshell, GCD consists of language features, runtime libraries, and system enhancements to provide systemic and comprehensive improvements to support concurrency on multi-core hardware in iOS and OS X. If you’d like to learn more about GCD, you can read our [Multithreading and Grand Central Dispatch on iOS for Beginners Tutorial](http://www.raywenderlich.com/?p=4295).

NSOperation and NSOperationQueue are built on top of GCD. As a very general rule, Apple recommends using the highest-level abstraction, and then dropping down to lower levels when measurements show they are needed.

Here’s a quick comparison of the two that will help you decide when and where to use GCD or NSOperation:

* **GCD** is a lightweight way to represent units of work that are going to be executed concurrently. You don’t schedule these units of work; the system takes care of scheduling for you. Adding dependency among blocks can be a headache. Canceling or suspending a block creates extra work for you as a developer! :]
* **NSOperation** adds a little extra overhead compared to GCD, but you can add dependency among various operations and re-use, cancel or suspend them.

Part 1: GCD (Grand Central Dispatch)

GCD is the most commonly used API to manage concurrent code and execute operations asynchronously at the Unix level of the system. GCD provides and manages queues of tasks. First, let’s see what queues are.

What are queues?

Queues are data structures that manage objects in the order of First-in, First-out (FIFO). Queues are similar to the lines at the ticket window of the movie theatre. The tickets are sold as first-come, first-serve. The people in the front of the line get to buy their tickets before the others in the line who arrived later. Queues in computer science are similar because the first object added to the queue is the first object to be removed from the queue.

Dispatch Queues

Dispatch queues are an easy way to perform tasks asynchronously and concurrently in your application. They are queues where tasks are being submitted by your app in form of blocks (Blocks of codes). There are two varieties of dispatch queues: (1) serial queues, & (2) concurrent queues. Before talking about the differences, you need to know that tasks assigned to both queues are being executed in separate threads than the thread they were created on. In other words, you create blocks of code and submit it to dispatch queues in the main thread. But all these tasks (Blocks of codes) will run in separate threads instead of the main thread.

Serial Queues

When you choose to create a queue as serial queue, the queue can only execute one task at a time. All tasks in the same serial queue will respect each other and execute serially. However, they don’t care about tasks in separate queues which means that you can still execute tasks concurrently by using multiple serial queues. For example, you can create two serial queues, each queue executes only one task at a time but up to two tasks could still execute concurrently.

Serial queues are awesome for managing a shared resource. It provides guaranteed serialized access to the shared resource and prevents race conditions. Imagine that there is a single ticket booth but there are a bunch of people who want to buy cinema tickets, here the staff at the booth is a shared resource. It’ll be chaotic if the staff has to serve these people all at the same time. To handle this situation, people are required to queue up (serial queue), so that the staff can serve the customers one at a time.

Again, it doesn’t mean the cinema can only handle one customer at a time. If it sets up two more booths, it can serve three customers at one time. This is why I said you can still perform multiple tasks in parallel by using several serial queues.

The advantages of using serial queues are:

1. Guaranteed serialized access to a shared resource that avoids race condition.
2. Tasks are executed in a predictable order. When you submit tasks in a serial dispatch queue, they will be executed in the same order as they are inserted.
3. You can create any number of serial queues.

Concurrent Queues

As the name suggests, concurrent queues allows you to execute multiple tasks in parallel. The tasks (blocks of codes) starts in the order in which they are added in the queue. But their execution all occur concurrently and they don’t have to wait for each other to start. Concurrent queues guarantee that tasks start in same order but you will not know the order of execution, execution time or the number of tasks being executed at a given point.

For example, you submit three tasks (task #1, #2 and #3) to a concurrent queue. The tasks are executed concurrently and are started in the order in which they were added to the queue. However, the execution time and finish time vary. Even it may take some time for task #2 and task #3 to start, they both can complete before task #1. It’s up to the system to decide the execution of the tasks.

Using Queues

Now that we have explained both serial and concurrent queues, it’s time to see how we can use them. By default, the system provides each application with a single serial queue and four concurrent queues. The main dispatch queue is the globally available serial queue that executes tasks on the application’s main thread. It is used to update the app UI and perform all tasks related to the update of UIViews. There is only one task to be executed at a time and this is why the UI is blocked when you run a heavy task in the main queue.

Besides the main queue, the system provides four concurrent queues. We call them Global Dispatch queues. These queues are global to the application and are differentiated only by their priority level. To use one of the global concurrent queues, you have to get a reference of your preferred queue using the function dispatch\_get\_global\_queue which takes in the first parameter one of these values:

* DISPATCH\_QUEUE\_PRIORITY\_HIGH
* DISPATCH\_QUEUE\_PRIORITY\_DEFAULT
* DISPATCH\_QUEUE\_PRIORITY\_LOW
* DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND

These queue types represent the priority of execution. The queue with HIGH has the highest priority and BACKGROUND has the lowest priority. So you can decide the queue you use based on the priority of the task. Please also note that these queues are being used by Apple’s APIs so your tasks are not the only tasks in these queues.

GCD

* GCD is a low-level C-based API that enables very simple use of a task-based concurrency model. NSOperation and NSOperationQueue are Objective-C classes that do a similar thing. NSOperationwas introduced first, but as of 10.6 and iOS 4, NSOperationQueue and friends are internally implemented using GCD.
* In general, you should use the highest level of abstraction that suits your needs. This means that you should usually use NSOperationQueue instead of GCD, unless you need to do something that NSOperationQueue doesn't support.
* Note that NSOperationQueue isn't a "dumbed-down" version of GCD; in fact, there are many things that you can do very simply with NSOperationQueue that take a lot of work with pure GCD. (Examples: bandwidth-constrained queues that only run N operations at a time; establishing dependencies between operations. Both very simple with NSOperation, very difficult with GCD.) Apple's done the hard work of leveraging GCD to create a very nice object-friendly API with NSOperation. Take advantage of their work unless you have a reason not to.
* NSOperationQueue and GCD
* **NSOperationQueue and GCD**
* NSOperationQueue predates Grand Central Dispatch and on iOS it doesn't use GCD to execute operations (this is different on Mac OS X). It uses regular background threads which have a little more overhead than GCD dispatch queues.
* On the other hand, NSOperationQueue gives you a lot more control over how your operations are executed. You can define dependencies between individual operations for example, which isn't possible with plain GCD queues. It is also possible to cancel operations that have been enqueued in an NSOperationQueue (as far as the operations support it). When you enqueue a block in a GCD dispatch queue, it will definitely be executed at some point.
* To sum it up, NSOperationQueue can be more suitable for long-running operations that may need to be cancelled or have complex dependencies. GCD dispatch queues are better for short tasks that should have minimum performance and memory overhead.

**Dispatch queues** are an easy way to perform tasks asynchronously and concurrently in your application. A task is simply some work that your application needs to perform.

down vote

accepted

When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

That being, said, in the context of computers this translates into executing a process or task on another "thread." A thread is a series of commands--a block of code--that exists as a unit of work. The operating system can manage multiple threads and assign a thread a piece ("slice") of processor time before switching to another thread to give it a turn to do some work. At its core (pardon the pun), a processor can simply execute a command--it has no concept of doing two things at one time. The operating system simulates this by allocating slices of time to different threads.

Working with Blocks

An Objective-C class defines an object that combines data with related behavior. Sometimes, it makes sense just to represent a single task or unit of behavior, rather than a collection of methods.

Blocks are a language-level feature added to C, Objective-C and C++, which allow you to create distinct segments of code that can be passed around to methods or functions as if they were values. Blocks are Objective-C objects, which means they can be added to collections like NSArray or NSDictionary. They also have the ability to capture values from the enclosing scope, making them similar to *closures* or *lambdas* in other programming languages.

This chapter explains the syntax to declare and refer to blocks, and shows how to use blocks to simplify common tasks such as collection enumeration. For further information, see *[Blocks Programming Topics](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html" \l "//apple_ref/doc/uid/TP40007502" \t "_self)*.

Block Syntax

The syntax to define a block literal uses the caret symbol (^), like this:

|  |
| --- |
| ^{ |
| NSLog(@"This is a block"); |
| } |

As with function and method definitions, the braces indicate the start and end of the block. In this example, the block doesn’t return any value, and doesn’t take any arguments.

In the same way that you can use a function pointer to refer to a C function, you can declare a variable to keep track of a block, like this:

|  |
| --- |
| void (^simpleBlock)(void); |

If you’re not used to dealing with C function pointers, the syntax may seem a little unusual. This example declares a variable called simpleBlock to refer to a block that takes no arguments and doesn’t return a value, which means the variable can be assigned the block literal shown above, like this:

|  |
| --- |
| simpleBlock = ^{ |
| NSLog(@"This is a block"); |
| }; |

This is just like any other variable assignment, so the statement must be terminated by a semi-colon after the closing brace. You can also combine the variable declaration and assignment:

|  |
| --- |
| void (^simpleBlock)(void) = ^{ |
| NSLog(@"This is a block"); |
| }; |

Once you’ve declared and assigned a block variable, you can use it to invoke the block:

|  |
| --- |
| simpleBlock(); |

**Note:** If you attempt to invoke a block using an unassigned variable (a nil block variable), your app will crash.

**How Do I Declare A Block in Objective-C?**

As a **local variable**:

returnType (^blockName)(parameterTypes) = ^returnType(parameters) {...};

As a **property**:

@property (nonatomic, copy) returnType (^blockName)(parameterTypes);

As a **method parameter**:

- (void)someMethodThatTakesABlock:(returnType (^)(parameterTypes))blockName;

As an **argument to a method call**:

[someObject someMethodThatTakesABlock:^returnType (parameters) {...}];

As a **typedef**:

typedef returnType (^TypeName)(parameterTypes);

TypeName blockName = ^returnType(parameters) {...};

**Precompiled header.**

**What it is?**

Prefix.pch is a precompiled header. Precompiled headers were invented **to make compiling faster**. Rather than parsing the same header files over and over, these files get parsed once, ahead of time.

## Extensions

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as retroactive modeling). Extensions are similar to categories in Objective-C. (Unlike Objective-C categories, Swift extensions do not have names.)

Extensions in Swift can:

* Add computed instance properties and computed type properties
* Define instance methods and type methods
* Provide new initializers
* Define subscripts
* Define and use new nested types
* Make an existing type conform to a protocol

In Swift, you can even extend a protocol to provide implementations of its requirements or add additional functionality that conforming types can take advantage of. For more details, see [Protocol Extensions](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html#//apple_ref/doc/uid/TP40014097-CH25-ID521).

NOTE

Extensions can add new functionality to a type, but they cannot override existing functionality.

### Extension Syntax

Declare extensions with the extension keyword:

1. extension SomeType {
2. // new functionality to add to SomeType goes here
3. }

DOM& SAX both are the way to parse XML Data

DOM

• DOM loads the file into the memory and then parse- the file.

• Has memory constraints since it loads the whole XML file before parsing.

• DOM is read and write (can insert or delete nodes).

• Slower at run time.

SAX

• Event based parser (Sequence of events).

• SAX parses the file as it reads it, i.e. parses node by node.

• No memory constraints as it does not store the XML content in the memory.

• SAX is read only i.e. can’t insert or delete the node.

• Use SAX parser when memory content is large.

• Faster at run time.

DOM parser example: Kiss parser

SAX parser example: NSXML parser

Core data

<http://iosdose.com/wp/2018/03/26/swift-core-data/>

Struct / Class

<https://medium.com/@maddy.lucky4u/copying-in-swift-5b02e9bad00d>

<http://sqlitebrowser.org>

-com.apple.CoreData.SQLDebug 1