APP Life Cycle:

———————

**didFinishLaunchingWithOptions:**

When your app first runs, you’ll see that the first method fired is**application: didFinishLaunchingWithOptions**.  This method is called one time when your application starts.  Provided your application isn’t shut down by the user or the OS (and your app doesn’t die) this method won’t be called again.  This is a great place for doing global initialization.  Things like initializing singletons (for analytics) or registering for push notifications  are great candidates for this method.

**applicationDidBecomeActive**

This method may be called more than once (as we’ll see) so you don’t necessarily want to do the same global initialization that you do in**didFinishLaunchingWithOptions**.  Instead this is where you may want to start any background tasks you have (or unpause them if you previously paused them).  You can also make calls to your UI to update it if necessary.

**applicationWillResignActive**

When you hit the home key, the first method called is **applicationWillResignActive**.  When this method is called you should begin to pause any tasks your application is running, disable timers, pause processing, end audio playback, etc.  It’s important to note that this method is called whether the user hits the home key or if the app is interrupted by a phone call.

**applicationDidEnterBackground**

After**applicationWillResignActive, applicationDidEnterBackground** will be called.  In this method you should save any data important for restoring user state if the application is killed.  Essentially, if the user then  double taps home and kills your app, you won’t get a notification.  When you return from this method, your application should have no problem being shut down or gracefully being reopened.  It’s VERY important to note that you only get *about* [5 seconds of processing time when applicationDidEnterBackground is called](http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UIApplicationDelegate_Protocol/Reference/Reference.html).  You can request additional time by using **beginBackgroundTaskWithExpirationHandler**.  It’s critical to remember this because if you don’t return from the call to **applicationDidEnterBackground** fast enough, you’re app can be terminated.  Now, double tap home and tap your app to relaunch it.

**applicationWillEnterForeground:**

double tap home and tap your app to relaunch it.The first method called will be **applicationWillEnterForeground**.  This method indicates that your app was previously in the background.  If you stop anything when your app was backgrounded but didn’t fully kill it, you can restart it here.  However, the **applicationDidBecomeActive**is called next.  This means that you need to decide if you want to fully reinitialize things in **DidBecomeActive** or handle them in **WillEnterForeground**.

**applicationWillTerminate**

this method is supposed to be called when your app is terminated, however, people seem to differ on how and when it will actually be called.  In practice, other life cycle methods will be called before this and are the appropriate place to handle app shut down tasks.

**applicationDidReceiveMemoryWarning**

It is called on your app when it is using too much memory.  This is an indicator from the OS that you should release memory if possible.

**Simulator sizes:(in Points)**

iphone 4:320\*480

iphone 5:320\*568

iphone6:375 \*667

Ipad : 768\*1024

**MAC Versions**

10.8 Mountain Lion

10.9-Maverics

10.10-Yosemite

10.11-EI Captain (june 8 2015)

**My Tasks in ERT Project**

---------------------------------

1.removal of demographic

2.close button on interstitial ad screen

3.practice messages clickable behaviour

4.updating programs and savings

5.Screen Saver Animation

**View Life Cycle:**

——————

**loadView**

This event creates the view that the controller manages. It is only called when the view controller is created programmatically. This makes it a good place to create your views in code.

**View DidLoad**—>

Notice the first call is to **[super viewDidLoad]**.    This means we’re calling the original **ViewController** classes implementation of the **viewDidLoad** method.  We do this because the root class does a lot of setup that we want to use but don’t want to code ourselves.  After that is done, we can perform whatever setup we want.

Called after init(coder:) when the view is loaded into memory, this method is also called only once during the life of the view controller object. It’s a great place to do any view initialization or setup you didn’t do in the Storyboard. Perhaps you want to add subviews or auto layout constraints programmatically – if so, this is a great place to do either of those. Note that just because the view has been loaded into memory *doesn’t* necessarily mean that it’s going to be displayed soon – for that, you’ll want to look at viewWillAppear.

**ViewWillAppear—>**

Called right before your view appears, good for hiding/showing fields or any operations that you want to happen every time before the view is visible. Because you might be going back and forth between views, this will be called every time your view is about to appear on the screen

**ViewDidAppear—>**

Called after the view appears - great place to start an animations or the loading of external data from an API.

**ViewDidDisappear and ViewWillDisAppear—>**

If you tap the button on our view (which will cause the second view we created to show up) you’ll then see the**viewWillDisappear**and **viewDidDisappear** methods be called.  This happens because your view isn’t being completely deallocated when you push another view on top of the current one.  Instead it is still stored in memory so it can be quickly restored to the screen when the user goes back.  One example of something you might want to do on **WillDisappear** is to save to storage any text data entered into view.  That way, if the user leaves your app before returning to the view, you can load that text data back into the view to give the user a seamless experience and not require them to retype it.

**viewWillLayoutSubviews**

This is the first step in the lifecycle where the bounds are finalized. If you are not using constraints or Auto Layout you probably want to update the subviews here.

**viewDidLayoutSubviews**

This event notifies the view controller that the subviews have been setup. It is a good place to make any changes to the subviews after they have been set.

didReceiveMemoryWarning—>

this method also just calls the **super** class’ method of the same name.  This method will only be called when your app is using up too much memory and is an indicator to you that you should release memory if possible.  With Automatic Reference Counting (ARC) you typically aren’t handling deallocating memory so you may think you don’t need to worry about this method or what it’s being called means.  However, you still have control over how much memory you are allocating and where it’s being used.  I would advise you to keep an eye out for this method being called and, if you do see it being called, refactoring or redesigning your code.

**Lifecycle in short:**

(initWithCoder—>loadView—>viewDidLoad—>viewWillAppear—>viewWillLayoutSubviews—>viewDidLayoutSubViews—>viewDidAppear)

After going to next viewController(On tapping button)

(viewWillDisAppear—>viewWillLayoutSubviews—>viewDidLayoutSubviews—>viewDidDisappear)

After going back to previous screen(After tapping on back button)

(viewWillAppear—>viewWillLayoutSubviews—>viewDidLayoutSubViews—>viewDidAppear)

Properties

--------------

**Atomic** : By default any property variable is atomic, used for thread safe. Only single thread can access at a time.

**NonAtomic**: can be used on Multiple threads, not thread safe.

**strong** is a replacement for the retain attribute, as part of Objective-C Automated Reference Counting (ARC). In non-ARC code it's just a synonym for retain.

**Weak**

weak is similar to strong except that it won't increase the reference count by 1. It does not become an owner of that object but just holds a reference to it. If the object's reference count drops to 0, even though you may still be pointing to it here, it will be deallocated from memory.

A weak reference is useful for an unusual situation called a retain cycle. A retain cycle occurs when two or more objects have strong references to each other. This is bad news. When two objects own each other, they will never be destroyed by ARC. Even if every other object in the application releases ownership of these objects, these objects (and any objects that they own) will continue to exist by virtue of those two strong references.

"**copy"** is needed when the object is mutable. Use this if you need the value of the object as it is at this moment, and you don't want that value to reflect any changes made by other owners of the object. You will need to release the object when you are finished with it because you are retaining the copy.

**Assign** is usually used for primitive types, the compiler will create the setter such that all that is done is a simple assign operation.

Whereas setting a value on a property with the 'retain' (now called "strong" with ARC) qualifier causes your backing instance variable to take ownership of (in other words retain) the object that was set.

With objects, if you don't want to take ownership as described and you're using ARC, you would most likely want to use the 'weak' qualifier instead of 'assign'.

**Categories and extensions**

**Categories** allows you to add methods to an existing class with out subclassing.

This is useful because we can add methods to predefined classes.If you want to

add methods to all instances of NSString class in your Application just add category.

A category is an extension of a class, not of a specific instance of a class. And, any modification that a category makes to a class is available to all subclasses (as with other methods on classes in OOP).

So, if you add a category on NSObject, basically all classes in your entire app have access to the methods in that category - hence you need to be very careful what methods you add there.

Whether you add a category or a helper class is personal preference in a lot of cases, both will work.

**Extensions** (also called **class extensions**) are a special type of category that requires their methods to be defined in the *main* implementation block for the associated class, as opposed to an implementation defined in a category.

**Category ex**

Ex: ex1.h

//import stmts

@interface Abc:NSObject

{

}

-(void)xxx;

//Ex1.m

Ex1.category.h

@interface Ex1<Category>

{

}

-(void)m1;

Ex1.category.m

-(void)m1

{

}

//some other class.h

#import Abc.h

#import Abc+Category.h

//someotherclass.m

-(void)someMethod

{

Abc \*a=[Abc new];

[a m1];

}

**Extension ex**

Ex: ex1.h

//import stmts

@interface Abc:NSObject

{

}

-(void)xxx;

//Ex1.m

@interface Ex1

{

}

-(void)m1; //extension method-🡪it becomes private

@implements

-(void)m1

{

}

//some other class.h

#import Abc.h

//someotherclass.m

-(void)someMethod

{

Abc \*a=[Abc new];

//[a m1];🡪not accessible(private method is not accessible outside of the class)

}

Delegation

--------------

Let's assume an object A calls an object B to perform an action. Once the action is complete, object A should know that B has completed the task and take necessary action. This is achieved with the help of delegates.

#import <Foundation/Foundation.h>

// Protocol definition starts here

@protocol SampleProtocolDelegate <NSObject>

@required

- (void) processCompleted;

@end

// Protocol Definition ends here

@interface SampleProtocol : NSObject

{

// Delegate to respond back

id <SampleProtocolDelegate> \_delegate;

}

@property (nonatomic,strong) id delegate;

-(void)startSampleProcess; // Instance method

@end

#import "SampleProtocol.h"

@implementation SampleProtocol

-(void)startSampleProcess{

[NSTimer scheduledTimerWithTimeInterval:3.0 target:self.delegate

selector:@selector(processCompleted) userInfo:nil repeats:NO];

}

@end

#import <UIKit/UIKit.h>

#import "SampleProtocol.h"

@interface ViewController : UIViewController<SampleProtocolDelegate>

{

IBOutlet UILabel \*myLabel;

}

@end

#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad

{

[super viewDidLoad];

SampleProtocol \*sampleProtocol = [[SampleProtocol alloc]init];

sampleProtocol.delegate = self;

[myLabel setText:@"Processing..."];

[sampleProtocol startSampleProcess];

// Do any additional setup after loading the view, typically from a nib.

}

- (void)didReceiveMemoryWarning

{

[super didReceiveMemoryWarning];

// Dispose of any resources that can be recreated.

}

#pragma mark - Sample protocol delegate

-(void)processCompleted{

[myLabel setText:@"Process Completed"];

}

@end

PushNotifications

------------------------

🡪**App registers for push notification**

 We call registerForRemoteNotificationTypes: on the application object, passing in the notification types that we're interested in. The operating system now knows that the application is interested in receiving push notifications.

- (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions {

    // Register for Remote Notifications

    [application registerForRemoteNotificationTypes:(UIRemoteNotificationTypeAlert | UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeSound)];

    return YES;

}

🡪**OS asks APNS for device token to proceed**

The operating system contacts Apple's servers and obtains a **device token**to uniquely identify the device the application is running on. This device token is used by your server infrastructure to send push notifications. It does this by sending the device token along with the actual push notification to Apple's servers. Apple's servers are in charge of distributing the push notifications to the appropriate devices.

Note that the device token differs for each application and it can even change over time for the same application. Apple therefore recommends to ask for a device token every time the application is launched and send the device token to your backend to make sure the device token is up to date.

🡪**App receives the device token**

- (void)application:(UIApplication \*)application didRegisterForRemoteNotificationsWithDeviceToken:(NSData \*)deviceToken {

    NSLog(@"Did Register for Remote Notifications with Device Token (%@)", deviceToken);

}

- (void)application:(UIApplication \*)application didFailToRegisterForRemoteNotificationsWithError:(NSError \*)error {

    NSLog(@"Did Fail to Register for Remote Notifications");

    NSLog(@"%@, %@", error, error.localizedDescription);

}

🡪**App sends the token to our server**

**🡪If something interesting happens our server sends notification to APNS sever**

**🡪APNS sends push notification to our app**

**Size Classes and Auto layout**

--------------------------------------

**Auto Layout**" determines whether a storyboard uses the Auto Layout features introduced in iOS 6 to**automatically layout your interface using constraints**.

Tolls used for auto layout: Stack, Align, Pin, Resolve.

**Size Classes** let you**set different constraints for certain screen sizes**. This let’s you go beyond basic stretching layouts. In Xcode 6 Apple introduced Size Classes.

There are two size classes that can apply to the horizontal (width) or vertical (height) dimension of an application interface:

**Regular**

**Compact**

**Model Portrait Landscape**

IPhone 6 Plus Regular Ht,,Compact Wdth Compact Ht, Regular Wdth,

Other Iphones “ Compact Ht,Compact Wdth

IPad ------------------------------------------🡪 regular ht and wdth Regular ht and wdth

**IPad Slide Over and Split Screens**

**For both Portraits**

**Height-🡪Regular**

**Width🡪Compact(primary app),Compact(secondary app)**

**-🡪For slide over landscape**

**For primary app width is regular.**

**Parsing JSON and XML**

**--------------------------------**

With JSON, you are limited to only storing classical data like text and numbers. 🡪**NSJSONSerialization (**Convert to data format array or dict.**)**

When you want to share documents, XML is the right tool for the job. This is because it allows you to include data types like images, charts, and graphs.

🡪**NSXMLParser (**need to implement delegate methods**)**

**Dispatch Queues**

----------------------

Main Queue-🡪Will work on main thread. It’s a serial queue. Serial queues always wait for a task to finish before going to the next one. Thus tasks are completed in FIFO order. You can make as many serial queues as you need. Prefer to do UI updates under this.

Global Queue🡪Will work on separate thread. It’s a concurrent queue.  tasks submitted to the global queue are thread-safe and minimize side-effects. Tasks are submitted for execution in FIFO order, but order of completion is not guaranteed.

**APP Distribution Guide**

**Archieving:**

🡪Select simulator device as IOS device or device name.

🡪Organizer🡪product🡪Archieve

**Installing app on IOS device:**

🡪Connect the testing device to a Mac running iTunes.

🡪Double-click the iOS App file that you created earlier.

🡪In iTunes, click the device in the upper-left corner of the window.

🡪Click the Apps button.

🡪Install and Sync

**Certificates**

-----------------

Apple Accounts: Standard (distribution can be done only to app store—99$---Individual🡪100 device can be added in devices section for testing) and Enterprise(distribution can be done through your own channel---299$---Organization--200 device can be added in devices section for testing)

Keychain access-🡪certificate Assistant🡪Request a certificate(this is CSR[Certificate Signing Request])🡪use this for both development and distribution

Dev portal🡪Member center🡪Certificates, identifiers and profiles-🡪Browse request certificate from desktop🡪development or distribution

🡪Create App Id

🡪generate Provisioning profile

🡪double click the provisioning profile🡪it opens IPhone Configuration Utility(if it is not there, install it)

🡪go to your App🡪project🡪info🡪Change bundle identifier🡪Bundle display name(change if u want)

(bundle identifier should be the same as the one used in profile generation)

🡪Build settings🡪Code signing🡪Select profile

**Finding UDID of a device:**

Connect device🡪go iTunes🡪right top corner, select iPhone🡪click on serial number🡪go to edit option in file menu🡪copy it.

**NSOperation Queue**

-----------------------------

An NSOperationQueue object is a queue that handles objects of the **NSOperation**class type. the NSOperation class is an abstract one so it cannot be used directly in the program. Instead, there are two provided subclasses, the **NSInvocationOperation** class and the**NSBlockOperation** class.

**For more: http://code.tutsplus.com/tutorials/working-with-the-nsoperationqueue-class--mobile-14993**

  // Create a new NSOperationQueue instance.

    operationQueue = [NSOperationQueue new];

    // Create a new NSOperation object using the NSInvocationOperation subclass.

    // Tell it to run the counterTask method.

    NSInvocationOperation \*operation = [[NSInvocationOperation alloc] initWithTarget:self

                                                                selector:@selector(counterTask)

                                                                object:nil];

    // Add the operation to the queue and let it to be executed.

    [operationQueue addOperation:operation];

    [operation release];

    // The same story as above, just tell here to execute the colorRotatorTask method.

    operation = [[NSInvocationOperation alloc] initWithTarget:self

                                                    selector:@selector(colorRotatorTask)

                                                    object:nil];

    [operationQueue addOperation:operation];

    [operation release];

GCD

-----

Serial Queue

Concurrent Queue

**Serial Queue**

**dispatch\_async(serialQueue, ^{counterBlock(1000  ,’A’ ,,self);});**

**dispatch\_async(serialQueue, ^{counterBlock(20000 ,’0′ ,2 ,self);});**

**dispatch\_async(serialQueue, ^{counterBlock(2000  ,’Z’ ,3 ,self);});**

// If we write start at block starting and end at block ending.

==block 1 started ==

==block 1 ended ==

==block 2 started ==

==block 2 ended ==

==block 3 started ==

**Concurrent Queue**

dispatch\_queue\_t concurrentQueue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_LOW, 0);

dispatch\_async(concurrentQueue, ^{counterBlock(1000  ,’A’ ,1 ,self);});

dispatch\_async(concurrentQueue, ^{counterBlock(20000 ,’0′ ,2 ,self);});

dispatch\_async(concurrentQueue, ^{counterBlock(2000  ,’Z’ ,3 ,self);});

// If we write start at block starting and end at block ending.

==block 2 started ==

==block 3 started ==

==block 3 ended ==

==block 1 started ==

…

**Synchronous and Asynchronous**

The only difference is that dispatch\_sync only return after the block is finished whereas dispatch\_async return after it is added to the queue and may not finished.

sync means the function WILL BLOCK the current thread until it has completed, async means it will be handled in the background and the function WILL NOT BLOCK the current thread.