Interoperating between Swift and an existing Objective-C code base

So you want to adopt Swift as your language for developing iOS but you already have a great project written in Objective-C that works great and has been thoroughly tested. That is where interoperability comes in. You can start adding Swift classes to your code now. This tutorial will get you started in connecting your Swift and Objective-C code to each other.

There is a sample project, ColorMatcher that follows the points being made in this tutorial. The project is a bit contrived to make certain points of interoperability.

To begin adding Swift use XCode 6 to add "New File" and select a "Swift File". Give the class a name and choose "Create". The first time you choose to make a new Swift file, XCode will ask you if you want to configure an Objective-C bridging header. The bridging header will expose the existing Objective-C code to Swift. It is likely you will need this at some point so confirm that you want the bridging header.

Bridging from Objective C to Swift

In the project will be two new files, a .swift file and –Bridging-Header.h file. In the sample project these are the ESTimer.swift and the ColorMatcher-Bridging-Header.h. When the bridging header is first created it is blank with the exception of one comment. In the bridging header add the headers for any classes that should be exposed to Swift classes. The direction is important to note here, this is Objective-C being accessed in Swift. Later, we will discuss traversing the other direction. The bridging header is now part of the project and should be committed along with other code if you are using source control.

In the newly created Swift file the only non-commented code is the import of Foundation. Foundation contains NSObject, NSArray and so on. The sample project also imports UIKit so we can have that available as well. In the class signature we need to decide if this class will be derived from an Objective-C class or if it will be pure Swift. In the sample project, ESTimer extends from NSObject so it is derived from an Objective-C class. As a derived class, it easy to instantiate a new ESTimer with standard Objective C as follows:

```
ESTimer timer = [[ESTimer alloc] init];
```

The @objc Attribute

The @objc attribute can be added to a class, property or method to expose your code to Objective C. A pure Swift class with the @objc attribute before the keyword class will then automatically add @objc to all properties and methods in that class. In a pure Swift class, a class level initializer is used, for example:

```
class func newInstance() -> ESTimer {
        return ESTimer
    }
```

The Objective C class could then instantiate a new ESTimer by calling:

```
ESTimer *timer=[ESTimer newInstance];
```

Swift Properties

In Swift all values, including objects, are guaranteed to be non-nil. However, values coming from Objective-C may be nil. Here Swift allows us to assign a value as an implicitly unwrapped optional using an exclamation mark. Best practice suggests checking optionals before using them. In the sample project you will see that the internalTimer property was assigned as an optional and is initialized later.

```
var internalTimer:NSTimer!
```

In the sample project the myCounterView property will be a CountdownView. CountdownView is an Objective C class so it's header needs to be added to the ColorMatcher-Bridging-Header.h. Notice in the sample project that mCounterView is created as an optional AnyObject. AnyObject is the Swift equivalent of Objective C id. The reason why it was assigned as an optional AnyObject instead of a CountdownView is that when the properties of the Swift class are created the Bridging-Header isn't in scope yet.

To instantiate an Objective C class in Swift use the Swift version of its constructor instead of the Objective-C version:

```
myCounterview = [[CountdownView alloc]
initWithFrame:CGRectMake(0, 0,20, 20)];
```

Swift uses:

```
myCounterView = CountdownView(frame:
CGRectMake(0,0,20,20))
```

Alloc has been handled for automatically and like other inits imported from Objective C, the init or initWith has been truncated from the name. In the sample project, Command click on the word frame on that line to be directed to the Swift version of the UIView class.

Implicitly Unwrapped Optionals

Before using myCounterView in the sample project check and unwrap the implicitly unwrapped optional.

```
if let counterView = view as? CountdownView{
}
```

In the code above the as? first checks to see if myCounterView is non-nil and can be downcast to CountdownView. If true, it will assign it to counterView as a CountdownView. Inside that if block counterView can be safely used. If you are sure that the downcast will succeed a forced cast can be used. However, a runtime error will occur if it does not. For example:

```
let counterView = myCounterView as? CountdownView
return counterView!
```

Converting Methods from Objective C

Method names converted from Objective C will use dot syntax style. The first part of the method name appears after the dot. For methods with parameters, the first parameter goes inside the parenthesis and all subsequent parameters must contain their argument names as well. If there

are no parameters you use (). For example, the following code in Objective C:

```
internalTimer = [NSTimer
scheduledTimerWithTimeInterval:1.0f target:self
selector:@selector(timerFunction:) userInfo:nil
repeats:YES];
```

Gets converted in Swift to:

```
self.internalTimer =
NSTimer.scheduledTimerWithTimeInterval(1.0, target:
self, selector: "timerFunction", userInfo: nil,
repeats: true);
```

Remember to use the override keyword if you implement any superclass methods.

Selector Stucture

SEL in Objective-C has been replaced by a Selector structure. Notice in the code above the @selector(timerFunction:) has been changed to a Swift string “timerFunction”.

Closures

Blocks in Objective C are identical to closures in Swift and can be passes back and forth. The one exception is that variables are passed by reference and are therefore mutable. In the sample code you will see that the Objective C version of:

```
dispatch_async(dispatch_get_main_queue(), ^{
});
```

In Swift has the following syntax:

```
dispatch_async(dispatch_get_main_queue(), {
})
```

Swift Delegate of an Objective C Protocol

To use an Objective C protocol the Objective C header file needs to include the Bridging-Header file and the protocol name needs to be added to the class signature. The protocols in the class signature go in a comma-separated list after the superclass if there is one. In the sample project the ControllerDelegate from the ViewController is implemented in the Swift ESTimer class.

Properties are by default strong in Swift so the delegate property is assigned as weak. It is also an optional because we might not have a delegate. When calling the delegate method, first check the delegate is non-nil and that it responds to the selector. For example:

```
self.delegate?.timeIsUp()
```

Bridging from Swift to Objective C

To import your Swift code into Objective C, the generated header needs to be added. Looking in the project and you won't see the file as the compiler generates it as ProjectName-Swift.h. For the sample project, the import in ViewController.m looks like:

```
#import "ColorMatcher-Swift.h"
```

Control clicking on the header will navigate to the entirety of the swift interface and shows how the class can be implemented from the Objective C class. This generated header does not need be committed with your SVN commit.

If using a Swift class in a framework, instead of importing the generated header use a forward declaration to the class, i.e. @class ESTimer; In the .m implementation file import using framework style <NameOfFrameWork/NameOfFrameWork-Swift.h>

Once the imports are in place, the Swift code can be accessed as though it were Objective C. In the sample project, there are examples of the Swift ESTimer and ESTimerDelegate used throughout the Objective C ViewController class. Notice the @obc attribute was added to the ESTimerDelegate protocol so that it could be accessed in the Objective C code.

There are limitations to what can be used from Swift. Swift specific features such and Generics and Tuples cannot be imported into Objective C. Items such as enums and structs defined in Swift are inaccessible. In the sample project the CurrentTimerState enum is unavailable. Notice the enum does not show in the generated header.

Storyboard and Xib Files using Swift

Connecting to storyboard or xib files is the same as with Objective C. Control-drag to connect IBOutlets and IBActions. Swift will automatically assign IBOutlets as weak and nil. Remember that properties can't be nil, however, when the associated storyboard or xib file is initialized it will be converted to an implicitly unwrapped optional.

Bridging Data Types

Swift will import types such as NSString to the Swift String class. However, there are some methods that need a bridge by using a cast, although no optional is needed here. For example to get the doubleValue from a UITextField:

```
(totalTextField.text as String).doubleValue()
```

For numbers, Swift bridges numeric data types: Int, UInt, Float, Double and Bool all to NSNumber. NSArray is bridged as an array of AnyObject[] and should be down cast with the checked implicitly unwrapped optional as seen earlier.

Closing

This post presents interoperability between Swift and an existing Objective C project. Although not exhaustive, you should be well on your way to confidently adding Swift to your project. If you have questions you may contact me at strohtennis @ gmail.