Using Swift with Cocoa and Objective-C

Developer

Contents

Getting Started

> **Important:** This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

- Basic Setup (page 6)

Basic Setup

Swift is designed to provide seamless compatibility with Cocoa and Objective-C. You can use Objective-C APIs (ranging from system frameworks to your own custom code) in Swift, and you can use Swift APIs in Objective-C. This compatibility makes Swift an easy, convenient, and powerful tool to integrate into your Cocoa app development workflow.

This guide covers three important aspects of this compatibility that you can use to your advantage when developing Cocoa apps:

- *Interoperability* lets you interface between Swift and Objective-C code, allowing you to use Swift classes in Objective-C and to take advantage of familiar Cocoa classes, patterns, and practices when writing Swift code.

- *Mix and match* allows you to create mixed-language apps containing both Swift and Objective-C files that can communicate with each other.

- *Migration* from existing Objective-C code to Swift is made easy with interoperability and mix and match, making it possible to replace parts of your Objective-C apps with the latest Swift features.

Before you get started learning about these features, you need a basic understanding of how to set up a Swift environment in which you can access Cocoa system frameworks.
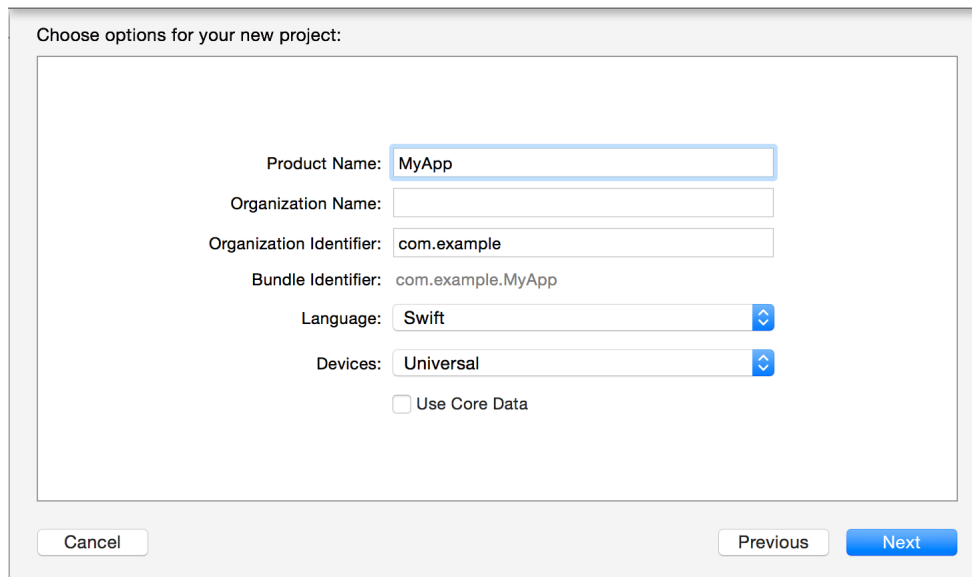
Setting Up Your Swift Environment

To start experimenting with accessing Cocoa frameworks in Swift, create a Swift-based app from one of the Xcode templates.

To create a Swift project in Xcode

1. Choose File > New > Project > (iOS *or* OS X) > Application > *your template of choice* .

2.   Click the Language pop-up menu and choose Swift.

Choose options for your new project:

Product Name:	MyApp
Organization Name:	
Organization Identifier:	com.example
Bundle Identifier:	com.example.MyApp
Language:	Swift
Devices:	Universal
	☐ Use Core Data

Cancel   Previous   Next

A Swift project's structure is nearly identical to an Objective-C project, with one important distinction: Swift has no header files. There is no explicit delineation between the implementation and the interface, so all the information about a particular class resides in a single `.swift` file.

From here, you can start experimenting by writing Swift code in the app delegate, or you can create a new Swift class file by choosing File > New > File > (iOS *or* OS X) > Other > Swift.

Understanding the Swift Import Process

After you have your Xcode project set up, you can import any framework from the Cocoa platform to start working with Objective-C from Swift.

Any Objective-C framework (or C library) that's accessible as a *module* can be imported directly into Swift. This includes all of the Objective-C system frameworks—such as Foundation, UIKit, and SpriteKit—as well as common C libraries supplied with the system. For example, to import Foundation, simply add this import statement to the top of the Swift file you're working in:

```
import Foundation
```

This import makes all of the Foundation APIs—including `NSDate`, `NSURL`, `NSMutableData`, and all of their methods, properties, and categories—directly available in Swift.

The import process is straightforward. Objective-C frameworks vend APIs in header files. In Swift, those header files are compiled down to Objective-C modules, which are then imported into Swift as Swift APIs. The importing determines how functions, classes, methods, and types declared in Objective-C code appear in Swift. For functions and methods, this process affects the types of their arguments and return values. For types, the process of importing can do the following things:

- Remap certain Objective-C types to their equivalents in Swift, like `id` to `AnyObject`

- Remap certain Objective-C core types to their alternatives in Swift, like `NSString` to `String`

- Remap certain Objective-C concepts to matching concepts in Swift, like pointers to optionals

In Interoperability (page 9), you'll learn more about these mappings and about how to leverage them in your Swift code.

The model for importing Swift into Objective-C is similar to the one used for importing Objective-C into Swift. Swift vends its APIs—such as from a framework—as Swift modules. Alongside these Swift modules are generated Objective-C headers. These headers vend the APIs that can be mapped back to Objective-C. Some Swift APIs do not map back to Objective-C because they leverage language features that are not available in Objective-C. For more information on using Swift in Objective-C, see Swift and Objective-C in the Same Project (page 44).

> **Note:** You cannot import C++ code directly into Swift. Instead, create an Objective-C or C wrapper for C++ code.

Interoperability

Interacting with Objective-C APIs

Interoperability is the ability to interface between Swift and Objective-C in either direction, letting you access and use pieces of code written in one language in a file of the other language. As you begin to integrate Swift into your app development workflow, it's a good idea to understand how you can leverage interoperability to redefine, improve, and enhance the way you write Cocoa apps.

One important aspect of interoperability is that it lets you work with Objective-C APIs when writing Swift code. After you import an Objective-C framework, you can instantiate classes from it and interact with them using native Swift syntax.

Initialization

To instantiate an Objective-C class in Swift, you call one of its initializers with Swift syntax. When Objective-C `init` methods come over to Swift, they take on native Swift initializer syntax. The "init" prefix gets sliced off and becomes a keyword to indicate that the method is an initializer. For `init` methods that begin with "initWith," the "With" also gets sliced off. The first letter of the selector piece that had "init" or "initWith" split off from it becomes lowercase, and that selector piece is treated as the name of the first argument. The rest of the selector pieces also correspond to argument names. Each selector piece goes inside the parentheses and is required at the call site.

For example, where in Objective-C you would do this:

```
UITableView *myTableView = [[UITableView alloc] initWithFrame:CGRectZero
style:UITableViewStyleGrouped];
```

In Swift, you do this:

```
let myTableView: UITableView = UITableView(frame: CGRectZero, style: .Grouped)
```

You don't need to call `alloc`; Swift correctly handles this for you. Notice that "init" doesn't appear anywhere when calling the Swift-style initializer.

You can be explicit in typing the object during initialization, or you can omit the type. Swift's type inference correctly determines the type of the object.

```
let myTextField = UITextField(frame: CGRect(0.0, 0.0, 200.0, 40.0))
```

These `UITableView` and `UITextField` objects have the same familiar functionality that they have in Objective-C. You can use them in the same way you would in Objective-C, accessing any properties and calling any methods defined on the respective classes.

For consistency and simplicity, Objective-C factory methods get mapped as convenience initializers in Swift. This mapping allows them to be used with the same concise, clear syntax as initializers. For example, whereas in Objective-C you would call this factory method like this:

```
UIColor *color = [UIColor colorWithRed:0.5 green:0.0 blue:0.5 alpha:1.0];
```

In Swift, you call it like this:

```
let color = UIColor(red: 0.5, green: 0.0, blue: 0.5, alpha: 1.0)
```

Accessing Properties

Access and set properties on Objective-C objects in Swift using dot syntax.

```
myTextField.textColor = UIColor.darkGrayColor()
myTextField.text = "Hello world"
if myTextField.editing {
    myTextField.editing = false
}
```

When getting or setting a property, use the name of the property without appending parentheses. Notice that `darkGrayColor` has a set of parentheses. This is because `darkGrayColor` is a class method on `UIColor`, not a property.

In Objective-C, a method that returns a value and takes no arguments can be treated as an implicit getter—and be called using the same syntax as a getter for a property. This is not the case in Swift. In Swift, only properties that are written using the `@property` syntax in Objective-C are imported as properties. Methods are imported and called as described in Working with Methods (page 12).

Working with Methods

When calling Objective-C methods from Swift, use dot syntax.

When Objective-C methods come over to Swift, the first part of an Objective-C selector becomes the base method name and appears outside the parentheses. The first argument appears immediately inside the parentheses, without a name. The rest of the selector pieces correspond to argument names and go inside the parentheses. All selector pieces are required at the call site.

For example, whereas in Objective-C you would do this:

```
[myTableView insertSubview:mySubview atIndex:2];
```

In Swift, you do this:

```
myTableView.insertSubview(mySubview, atIndex: 2)
```

If you're calling a method with no arguments, you must still include the parentheses.

```
myTableView.layoutIfNeeded()
```

id Compatibility

Swift includes a protocol type named `AnyObject` that represents any kind of object, just as `id` does in Objective-C. The `AnyObject` protocol allows you to write type-safe Swift code while maintaining the flexibility of an untyped object. Because of the additional safety provided by the `AnyObject` protocol, Swift imports `id` as `AnyObject`.

For example, as with `id`, you can assign an object of any class type to a constant or variable typed as `AnyObject`. You can also reassign a variable to an object of a different type.

```
var myObject: AnyObject = UITableViewCell()
myObject = NSDate()
```

You can also call any Objective-C method and access any property without casting to a more specific class type. This includes Objective-C compatible methods marked with the `@objc` attribute.

```
let futureDate = myObject.dateByAddingTimeInterval(10)
```

```
let timeSinceNow = myObject.timeIntervalSinceNow
```

However, because the specific type of an object typed as `AnyObject` is not known until runtime, it is possible to inadvertently write unsafe code. Additionally, in contrast with Objective-C, if you invoke a method or access a property that does not exist on an `AnyObject` typed object, it is a runtime error. For example, the following code compiles without complaint and then causes an unrecognized selector error at runtime:

```
myObject.characterAtIndex(5)
// crash, myObject does't respond to that method
```

However, you can take advantage of optionals in Swift to eliminate this common Objective-C error from your code. When you call an Objective-C method on an `AnyObject` type object, the method call actually behaves like an implicitly unwrapped optional. You can use the same optional chaining syntax you would use for optional methods in protocols to optionally invoke a method on `AnyObject`. This same process applies to properties as well.

For example, in the code listing below, the first and second lines are not executed because the `length` property and the `characterAtIndex:` method do not exist on an `NSDate` object. The `myLength` constant is inferred to be an optional `Int`, and is set to `nil`. You can also use an `if-let` statement to conditionally unwrap the result of a method that the object may not respond to, as shown on line three.

```
let myLength = myObject.length?
let myChar = myObject.characterAtIndex?(5)
if let fifthCharacter = myObject.characterAtIndex(5) {
    println("Found \(fifthCharacter) at index 5")
}
```

As with all downcasts in Swift, casting from `AnyObject` to a more specific object type is not guaranteed to succeed and therefore returns an optional value. You can check that optional value to determine whether the cast succeeded.

```
let userDefaults = NSUserDefaults.standardUserDefaults()
let lastRefreshDate: AnyObject? = userDefaults.objectForKey("LastRefreshDate")
if let date = lastRefreshDate as? NSDate {
    println("\(date.timeIntervalSinceReferenceDate)")
}
```

Of course, if you are certain of the type of the object (and know that it is not `nil`), you can force the invocation with the `as` operator.

```
let myDate = lastRefreshDate as NSDate
let timeInterval = myDate.timeIntervalSinceReferenceDate
```

Working with nil

In Objective-C, you work with references to objects using raw pointers that could be `NULL` (also referred to as `nil` in Objective-C). In Swift, all values—including structures and object references—are guaranteed to be non–nil. Instead, you represent a value that could be missing by wrapping the type of the value in an optional type. When you need to indicate that a value is missing, you use the value `nil`. You can read more about optionals in Optionals.

Because Objective-C does not make any guarantees that an object is non-nil, Swift makes all classes in argument types and return types optional in imported Objective-C APIs. Before you use an Objective-C object, you should check to ensure that it is not missing.

In some cases, you might be *absolutely* certain that an Objective-C method or property never returns a `nil` object reference. To make objects in this special scenario more convenient to work with, Swift imports object types as *implicitly unwrapped optionals*. Implicitly unwrapped optional types include all of the safety features of optional types. In addition, you can access the value directly without checking for `nil` or unwrapping it yourself. When you access the value in this kind of optional type without safely unwrapping it first, the implicitly unwrapped optional checks whether the value is missing. If the value is missing, a runtime error occurs. As a result, you should always check and unwrap an implicitly unwrapped optional yourself, unless you are sure that the value cannot be missing.

Extensions

A Swift extension is similar to an Objective-C category. *Extensions* expand the behavior of existing classes, structures, and enumerations, including those defined in Objective-C. You can define an extension on a type from either a system framework or one of your own custom types. Simply import the appropriate module, and refer to the class, structure, or enumeration by the same name that you would use in Objective-C.

For example, you can extend the `UIBezierPath` class to create a simple Bézier path with an equilateral triangle, based on a provided side length and starting point.

```
extension UIBezierPath {

    convenience init(triangleSideLength: Float, origin: CGPoint) {

        self.init()

        let squareRoot = Float(sqrt(3))

        let altitude = (squareRoot * triangleSideLength) / 2

        moveToPoint(origin)

        addLineToPoint(CGPoint(triangleSideLength, origin.x))

        addLineToPoint(CGPoint(triangleSideLength / 2, altitude))

        closePath()

    }

}
```

You can use extensions to add properties (including class and static properties). However, these properties must be computed; extensions can't add stored properties to classes, structures, or enumerations.

This example extends the `CGRect` structure to contain a computed `area` property:

```
extension CGRect {

    var area: CGFloat {

    return width * height

    }

}
let rect = CGRect(x: 0.0, y: 0.0, width: 10.0, height: 50.0)

let area = rect.area

// area: CGFloat = 500.0
```

You can also use extensions to add protocol conformance to a class without subclassing it. If the protocol is defined in Swift, you can also add conformance to it to structures or enumerations, whether defined in Swift or Objective-C.

You cannot use extensions to override existing methods or properties on Objective-C types.

Closures

Objective-C blocks are automatically imported as Swift closures. For example, here is an Objective-C block variable:

```
void (^completionBlock)(NSData *, NSError *) = ^(NSData *data, NSError *error) {/*
 ... */}
```

And here's what it looks like in Swift:

```
let completionBlock: (NSData, NSError) -> Void = {data, error in /* ... */}
```

Swift closures and Objective-C blocks are compatible, so you can pass Swift closures to Objective-C methods that expect blocks. Swift closures and functions have the same type, so you can even pass the name of a Swift function.

Closures have similar capture semantics as blocks but differ in one key way: Variables are mutable rather than copied. In other words, the behavior of `__block` in Objective-C is the default behavior for variables in Swift.

Object Comparison

There are two distinct types of comparison when you compare two objects in Swift. The first, *equality* (==), compares the contents of the objects. The second, *identity* (===), determines whether or not the constants or variables refer to the same object instance.

Swift and Objective-C objects are typically compared in Swift using the == and === operators. Swift provides a default implementation of the == operator for objects that derive from the `NSObject` class. In the implementation of this operator, Swift invokes the `isEqual:` method defined on the `NSObject` class. The `NSObject` class only performs an identity comparison, so you should implement your own `isEqual:` method in classes that derive from the `NSObject` class. Because you can pass Swift objects (including ones not derived from `NSObject`) to Objective-C APIs, you should implement the `isEqual:` method for these classes if you want the Objective-C APIs to compare the contents of the objects rather than their identities.

As part of implementing equality for your class, be sure to implement the `hash` property according to the rules in Object comparison. Further, if you want to use your class as keys in a dictionary, also conform to the `Hashable` protocol and implement the `hashValue` property.

Swift Type Compatibility

When you define a Swift class that inherits from `NSObject` or any other Objective-C class, the class is automatically compatible with Objective-C. All of the steps in this section have already been done for you by the Swift compiler. If you never import a Swift class in Objective-C code, you don't need to worry about type compatibility in this case as well. Otherwise, if your Swift class does not derive from an Objective-C class and you want to use it from Objective-C code, you can use the `@objc` attribute described below.

The `@objc` attribute makes your Swift API available in Objective-C and the Objective-C runtime. In other words, you can use the `@objc` attribute before any Swift method, property, or class that you want to use from Objective-C code. If your class inherits from an Objective-C class, the compiler inserts the attribute for you. The compiler also adds the attribute to every method and property in a class that is itself marked with the `@objc` attribute. When you use the `@IBOutlet`, `@IBAction`, or `@NSManaged` attribute, the `@objc` attribute is added as well. This attribute is also useful when you're working with Objective-C classes that use selectors to implement the target-action design pattern—for example, `NSTimer` or `UIButton`.

When you use a Swift API from Objective-C, the compiler typically performs a direct translation. For example, the Swift API `func playSong(name: String)` is imported as `− (void)playSong:(NSString *)name` in Objective-C. However, there is one exception: When you use a Swift initializer in Objective-C, the compiler adds the text "initWith" to the beginning of the method and properly capitalizes the first character in the original initializer. For example, this Swift initializer `init (songName: String, artist: String)` is imported as `− (instancetype)initWithSongName:(NSString *)songName artist:(NSString *)artist` in Objective-C.

Swift also provides a variant of the `@objc` attribute that allows you to specify name for your symbol in Objective-C. For example, if the name of your Swift class contains a character that isn't supported by Objective-C, you can provide an alternative name to use in Objective-C. If you provide an Objective-C name for a Swift function, use Objective-C selector syntax. Remember to add a colon (`:`) wherever a parameter follows a selector piece.

```
@objc(Squirrel)
class Белка {
    @objc(initWithName:)
    init (имя: String) { /*...*/ }
    @objc(hideNuts:inTree:)
    func прячьОрехи(Int, вДереве: Дерево) { /*...*/ }
}
```

When you use the `@objc(<#name#>)` attribute on a Swift class, the class is made available in Objective-C without any namespacing. As a result, this attribute can also be useful when you migrate an archivable Objective-C class to Swift. Because archived objects store the name of their class in the archive, you should use the `@objc(<#name#>)` attribute to specify the same name as your Objective-C class so that older archives can be unarchived by your new Swift class.

Objective-C Selectors

An Objective-C selector is a type that refers to the name of an Objective-C method. In Swift, Objective-C selectors are represented by the `Selector` structure. You can construct a selector with a string literal, such as `let mySelector: Selector = "tappedButton:"`. Because string literals can be automatically converted to selectors, you can pass a string literal to any method that accepts a selector.

```
import UIKit
class MyViewController: UIViewController {

    let myButton = UIButton(frame: CGRect(x: 0, y: 0, width: 100, height: 50))


    init(nibName nibNameOrNil: String!, bundle nibBundleOrNil: NSBundle!) {
        super.init(nibName: nibName, bundle: nibBundle)
        myButton.targetForAction("tappedButton:", withSender: self)
    }


    func tappedButton(sender: UIButton!) {
        println("tapped button")
    }
}
```

> **Note:** The `performSelector:` method and related selector-invoking methods are not imported in Swift because they are inherently unsafe.

If your Swift class inherits from an Objective-C class, all of the methods and properties in the class are available as Objective-C selectors. Otherwise, if your Swift class does not inherit from an Objective-C class, you need to prefix the symbol you want to use as a selector with the `@objc` attribute, as described in Swift Type Compatibility (page 17).

Writing Swift Classes with Objective-C Behavior

Interoperability lets you define Swift classes that incorporate Objective-C behavior. You can subclass Objective-C classes, adopt Objective-C protocols, and take advantage of other Objective-C functionality when writing a Swift class. This means that you can create classes based on familiar, established behavior in Objective-C and enhance them with Swift's modern and powerful language features.

Inheriting from Objective-C Classes

In Swift, you can define subclasses of Objective-C classes. To create a Swift class that inherits from an Objective-C class, add a colon (`:`) after the name of the Swift class, followed by the name of the Objective-C class.

```
import UIKit


class MySwiftViewController: UIViewController {
    // define the class
}
```

You get all the functionality offered by the superclass in Objective-C. If you provide your own implementations of the superclass's methods, remember to use the `override` keyword.

Adopting Protocols

In Swift, you can adopt protocols that are defined in Objective-C. Like Swift protocols, any Objective-C protocols go in a comma-separated list following the name of a class's superclass, if any.

```
class MySwiftViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource {
    // define the class
}
```

Objective-C protocols come in as Swift protocols. If you want to refer to the `UITableViewDelegate` protocol in Swift code, refer to it as `UITableViewDelegate` (as compared to `id<UITableViewDelegate>` in Objective-C).

Because the namespace of classes and protocols is unified in Swift, the `NSObject` protocol in Objective-C is remapped to `NSObjectProtocol` in Swift.

Writing Initializers and Deinitializers

The Swift compiler ensures that your initializers do not leave any properties in your class uninitialized to increase the safety and predictability of your code. Additionally, unlike Objective-C, in Swift there is no separate memory allocation method to invoke. You use native Swift initialization syntax even when you are working with Objective-C classes—Swift converts Objective-C initialization methods to Swift initializers. You can read more about implementing your own initializers in Initializers.

When you want to perform additional clean-up before your class is deallocated, you can implement a deninitializer instead of the `dealloc` method. Swift deinitializers are called automatically, just before instance deallocation happens. Swift automatically calls the superclass deinitializer after invoking your subclass's deinitializer. When you are working with an Objective-C class or your Swift class inherits from an Objective-C class, Swift calls your class's superclass `dealloc` method for you as well. You can read more about implementing your own deinitializers in Deinitializers.

Integrating with Interface Builder

The Swift compiler includes attributes that enable Interface Builder features for your Swift classes. As in Objective-C, you can use outlets, actions, and live rendering in Swift.

Working with Outlets and Actions

Outlets and actions allow you to connect your source code to user interface objects in Interface Builder. To use outlets and actions in Swift, insert `@IBOutlet` or `@IBAction` just before the property or method declaration. You use the same `@IBOutlet` attribute to declare an outlet collection—just specify an array for the type.

When you declare an outlet in Swift, the compiler automatically converts the type to a weak implicitly unwrapped optional and assigns it an initial value of `nil`. In effect, the compiler replaces `@IBOutlet var name: Type` with `@IBOutlet weak var name: Type! = nil`. The compiler converts the type to an implicitly unwrapped

optional so that you aren't required to assign a value in an initializer. It is implicitly unwrapped because after your class is initialized from a storyboard or `xib` file, you can assume that the outlet has been connected. Outlets are weak by default because the outlets you create usually have weak relationships.

For example, the following Swift code declares a class with an outlet, an outlet collection, and an action:

```swift
class MyViewController: UIViewController {

    @IBOutlet var button: UIButton

    @IBOutlet var textFields: UITextField[]

    @IBAction func buttonTapped(AnyObject) {

        println("button tapped!")

    }

}
```

Because the sender parameter of the `buttonTapped:` method wasn't used, the parameter name can be omitted.

Live Rendering

You can use two different attributes—`@IBDesignable` and `@IBInspectable`—to enable live, interactive custom view design in Interface Builder. When you create a custom view that inherits from `UIView` or `NSView`, you can add the `@IBDesignable` attribute just before the class declaration. After you add the custom view to Interface Builder (by setting the custom class of the view in the inspector pane), Interface Builder renders your view in the canvas.

> **Note:** Live rendering can be used only from frameworks.

You can also add the `@IBInspectable` attribute to properties with types compatible with user defined runtime attributes. After you add your custom view to Interface Builder, you can edit these properties in the inspector.

```swift
@IBDesignable

class MyCustomView: UIView {

    @IBInspectable var textColor: UIColor

    @IBInspectable var iconHeight: CGFloat

    /* ... */

}
```

Specifying Property Attributes

In Objective-C, properties have a range of potential attributes that specify additional information about a property's behavior. In Swift, you specify these property attributes in a different way.

Strong and Weak

Swift properties are strong by default. Use the `weak` keyword to indicate that a property has a weak reference to the object stored as its value. This keyword can be used only for properties that are optional class types. For more information, see Attributes.

Read/Write and Read-Only

In Swift, there are no `readwrite` and `readonly` attributes. When declaring a stored property, use `let` to make it read-only, and use `var` to make it read/write. When declaring a computed property, provide a getter only to make it read-only and provide both a getter and setter to make it read/write. For more information, see Properties.

Copy Semantics

In Swift, the Objective-C `copy` property attribute translates to `@NSCopying`. The type of the property must conform to the `NSCopying` protocol. For more information, see Attributes.

Implementing Core Data Managed Object Subclasses

Core Data provides the underlying storage and implementation of properties in subclasses of the `NSManagedObject` class. Add the `@NSManaged` attribute before each property definition in your managed object subclass that corresponds to an attribute or relationship in your Core Data model. Like the `@dynamic` attribute in Objective-C, the `@NSManaged` attribute informs the Swift compiler that the storage and implementation of a property will be provided at runtime. However, unlike `@dynamic`, the `@NSManaged` attribute is available only for Core Data support.

Working with Cocoa Data Types

As part of its interoperability with Objective-C, Swift offers convenient and efficient ways of working with Cocoa data types.

Swift automatically converts some Objective-C types to Swift types, and some Swift types to Objective-C types. There are also a number of data types in Swift and Objective-C that can be used interchangeably. Data types that are convertible or can be used interchangeably are referred to as *bridged* data types. For example, in Swift code, you can pass an `Array` value to a method expecting an `NSArray` object. You can also cast between a bridged type and its counterpart. When you cast between bridged types with `as`—or by explicitly providing the type of constant or variable—Swift bridges the data type.

Swift also provides a convenient overlay for interfacing with Foundation data types, letting you work with them using a syntax that feels natural and unified with the rest of the Swift language.

Strings

Swift automatically bridges between the `String` type and the `NSString` class. This means that anywhere you use an `NSString` object, you can use a Swift `String` type instead and gain the benefits of both types—the `String` type's interpolation and Swift-designed APIs and the `NSString` class's broad functionality. For this reason, you should almost never need to use the `NSString` class directly in your own code. In fact, when Swift imports Objective-C APIs, it replaces all of the `NSString` types with `String` types. When your Objective-C code uses a Swift class, the importer replaces all of the `String` types with `NSString` in imported API.

To enable string bridging, just import Foundation. For example, you can call `capitalizedString`—a method on the `NSString` class—on a Swift string, and Swift automatically bridges the Swift `String` to an `NSString` object and calls the method. The method even returns a Swift `String` type, because it was converted during import.

```
import Foundation
let greeting = "hello, world!"
let capitalizedGreeting = greeting.capitalizedString
// capitalizedGreeting: String = Hello, World!
```

If you do need to use an NSString object, you can convert it to a Swift String value by casting it. The String type can always be converted from an NSString object to a Swift String value so there's no need to use the optional version of the type casting operator (as?). You can also create an NSString object from a string literal by explicitly typing the constant or variable.

```
import Foundation

let myString: NSString = "123"

if let integerValue = (myString as String).toInt() {

    println("\(myString) is the integer \(integerValue)")

}
```

Localization

In Objective-C, you typically used the NSLocalizedString family of macros to localize strings. This set of macros includes NSLocalizedString, NSLocalizedStringFromTable, NSLocalizedStringFromTableInBundle, and NSLocalizedStringWithDefaultValue. In Swift you can use a single function that provides the same functionality as the entire set of NSLocalizedString macros—NSLocalizedString(key:tableName:bundle:value:comment:). The NSLocalizedString function provides default values for the tableName, bundle, and value arguments. Use it as you would use the macro it replaces.

Numbers

Swift automatically bridges certain native number types, such as Int and Float, to NSNumber. This bridging lets you create an NSNumber from one of these types:

```
let n = 42
let m: NSNumber = n
```

It also allows you to pass a value of type Int, for example, to an argument expecting an NSNumber. However, note that because NSNumber can contain a variety of different types, you cannot pass it to something expecting an Int value.

All of the following types are automatically bridged to NSNumber:

- Int

- UInt

- `Float`

- `Double`

- `Bool`

Collection Classes

Swift automatically bridges the `NSArray` and `NSDictionary` classes to their native Swift equivalents. This means you can take advantage of Swift's powerful algorithms and natural syntax for working with collections—and use Foundation and Swift collection types interchangeably.

Arrays

Swift automatically bridges between the `Array` type and the `NSArray` class. When you bridge from an `NSArray` object to a Swift array, the resulting array is of type `AnyObject[]`. An object is `AnyObject` compatible if it is an instance of an Objective-C or Swift class, or if the object can be bridged to one. You can bridge any `NSArray` object to a Swift array because all Objective-C objects are `AnyObject` compatible. Because all `NSArray` objects can be bridged to Swift arrays, the Swift compiler replaces the `NSArray` class with `AnyObject[]` when it imports Objective-C APIs.

After you bridge an `NSArray` object to a Swift array, you can also *downcast* the array to a more specific type. Unlike casting from the `NSArray` class to the `AnyObject[]` type, downcasting from `AnyObject` to a more specific type is not guaranteed to succeed. The compiler cannot know for certain until runtime that all of the elements in the array can be downcasted to the type you specified. As a result, downcasting from `AnyObject[]` to `SomeType[]` returns an optional value. For example, if you know that a Swift array contains only instances of the `UIView` class (or a subclass of the `UIView` class), you can downcast the array of `AnyObject` type elements to an array of `UIView` objects. If any element in the Swift array is not actually a `UIView` object at runtime, the cast returns `nil`.

```
let swiftArray = foundationArray as AnyObject[]
if let downcastedSwiftArray = swiftArray as? UIView[] {
    // downcastedSwiftArray contains only UIView objects
}
```

You can also downcast directly from an `NSArray` object to a Swift array of a specific type in a for loop:

```
for aView: UIView! in foundationArray {
    // aView is of type UIView
```

```
    }
```

> **Note:** This cast is a forced cast, and will result in a runtime error if the cast does not succeed.

When you bridge from a Swift array to an `NSArray` object, the elements in the Swift array must be `AnyObject` compatible. For example, a Swift array of type `Int[]` contains `Int` structure elements. The `Int` type is not an instance of a class, but because the `Int` type bridges to the `NSNumber` class, the `Int` type is `AnyObject` compatible. Therefore, you can bridge a Swift array of type `Int[]` to an `NSArray` object. If an element in a Swift array is not `AnyObject` compatible, a runtime error occurs when you bridge to an `NSArray` object.

You can also create an `NSArray` object directly from a Swift array literal, following the same bridging rules outlined above. When you explicitly type a constant or variable as an `NSArray` object and assign it an array literal, Swift creates an `NSArray` object instead of a Swift array.

```
let schoolSupplies: NSArray = ["Pencil", "Eraser", "Notebook"]
// schoolSupplies is an NSArray object containing NSString objects
```

In the example above, the Swift array literal contains three `String` literals. Because the `String` type bridges to the `NSString` class, the array literal is bridged to an `NSArray` object and the assignment to `schoolSupplies` succeeds.

When you use a Swift class or protocol in Objective-C code, the importer replaces all Swift arrays of any type in imported API with `NSArray`. If you pass an `NSArray` object to a Swift API that expects the elements to be of a different type, a runtime error occurs. If a Swift API returns a Swift array that cannot be bridged to `NSArray`, a runtime error occurs.

Dictionaries

Information forthcoming.

Foundation Data Types

Swift provides a convenient overlay for interfacing with data types defined in the Foundation framework. Use this overlay to work with types like `NSSize` and `NSPoint`, using a syntax that feels natural and unified with the rest of the Swift language. For example, you can create an `NSSize` structure using this syntax:

```
let size = NSSize(width: 20, height: 40)
```

The overlay also lets you call Foundation functions on structures in a natural way.

```
let rect = NSRect(x: 50, y: 50, width: 100, height: 100)
let width = rect.width    // equivalent of NSWidth(rect)
let maxX = rect.maxY      // equivalent of NSMaxY(rect)
```

Swift bridges `NSUInteger` and `NSInteger` to `Int`. Both of these types come over as `Int` in Foundation APIs. `Int` is used for consistency whenever possible in Swift, but the `UInt` type is available if you require an unsigned integer type.

Foundation Functions

`NSLog` is available in Swift for logging to the system console. You use the same formatting syntax you would use in Objective-C.

```
NSLog("%.7f", pi)          // Logs "3.1415927" to the console
```

However, Swift also has print functions like `print` and `println` available. These functions are simple, powerful, and versatile due to Swift's string interpolation. They don't print to the system console but are available for general printing needs.

`NSAssert` functions do not carry over to Swift. Instead, use the `assert` function.

Core Foundation

Core Foundation types are automatically imported as full-fledged Swift classes. Wherever memory management annotations have been provided, Swift automatically manages the memory of Core Foundation objects, including Core Foundation objects that you instantiate yourself. In Swift, you can use each pair of toll-free bridged Foundation and Core Foundation types interchangeably. You can also bridge some toll-free bridged Core Foundation types to Swift standard library types if you cast to a bridging Foundation type first.

Remapped Types

When Swift imports Core Foundation types, the compiler remaps the names of these types. The compiler removes *Ref* from the end of each type name because all Swift classes are reference types, therefore the suffix is redundant.

The Core Foundation `CFTypeRef` type completely remaps to the `AnyObject` type. Wherever you would use `CFTypeRef`, you should now use `AnyObject` in your code.

Memory Managed Objects

Core Foundation objects returned from annotated APIs are automatically memory managed in Swift—you do not need to invoke the `CFRetain`, `CFRelease`, or `CFAutorelease` functions yourself. If you return Core Foundation objects from your own C functions and Objective-C methods, annotate them with either `CF_RETURNS_RETAINED` or `CF_RETURNS_NOT_RETAINED`. The compiler automatically inserts memory management calls when it compiles Swift code that invokes these APIs. If you use only annotated APIs that do not indirectly return Core Foundation objects, you can skip the rest of this section. Otherwise, continue on to learn about working with unmanaged Core Foundation objects.

Unmanaged Objects

When Swift imports APIs that have not been annotated, the compiler cannot automatically memory manage the returned Core Foundation objects. Swift wraps these returned Core Foundation objects in an Unmanaged<T> structure. All indirectly returned Core Foundation objects are unmanaged as well. For example, here's an unannotated C function:

```
CFStringRef StringByAddingTwoStrings(CFStringRef string1, CFStringRef string2)
```

And here's how Swift imports it:

```
func StringByAddingTwoStrings(CFString!, CFString!) -> Unmanaged<CFString>!
```

When you receive an unmanaged object from an unannotated API, you should immediately convert it to a memory managed object before you work with it. That way, Swift can handle memory management for you. The Unmanaged<T> structure provides two methods to convert an unmanaged object to a memory managed object—`takeUnretainedValue()` and `takeRetainedValue()`. Both of these methods return the original, unwrapped type of the object. You choose which method to use based on whether the API you are invoking returns an unretained or retained object.

For example, suppose the C function above does not retain the `CFString` object before returning it. To start using the object, you use the `takeUnretainedValue()` function.

```
let memoryManagedResult = StringByAddingTwoStrings(str1, str2).takeUnretainedValue()
// memoryManagedResult is a memory managed CFString
```

You can also invoke the `retain()`, `release()`, and `autorelease()` methods on unmanaged objects, but this approach is not recommended.

Adopting Cocoa Design Patterns

One aid in writing well-designed, resilient apps is to use Cocoa's established design patterns. Many of these patterns rely on classes defined in Objective-C. Because of Swift's interoperability with Objective-C, you can take advantage of these common patterns in your Swift code. In many cases, you can use Swift language features to extend or simplify existing Cocoa patterns, making them more powerful and easier to use.

Delegation

In both Swift and Objective-C, delegation is often expressed with a protocol that defines the interaction and a conforming delegate property. In contrast with Objective-C, when you implement delegation in Swift, the pattern stays the same but the implementation changes. Just as in Objective-C, before you send a message to the delegate you check to see whether it's `nil`—and if the method is optional, you check to see whether the delegate responds to the selector. In Swift, these questions can be answered while maintaining type safety. The code listing below illustrates the following process:

1. Check that `myDelegate` is not `nil`.

2. Check that `myDelegate` implements the method `window:willUseFullScreenContentSize:`.

3. If 1 and 2 hold true, invoke the method and assign the result of the method to the value named `fullScreenSize`.

4. Print the return value of the method.

```
// @inteface MyObject : NSObject
// @property (nonatomic, weak) id<NSWindowDelegate> delegate;
// @end
if let fullScreenSize = myDelegate?.window?(myWindow, willUseFullScreenContentSize:
 mySize) {
    println(NSStringFromSize(fullScreenSize))
}
```

> **Note:** In a pure Swift app, type the `delegate` property as an optional `NSWindowDelegate` object and assign it an initial value of `nil`.

Lazy Initialization

Information forthcoming. You can read more about lazy initialization in Lazy Stored Properties.

Error Reporting

Error reporting in Swift follows the same pattern it does in Objective-C, with the added benefit of offering optional return values. In the simplest case, you return a `Bool` value from the function to indicate whether or not it succeeded. When you need to report the reason for the error, you can add to the function an `NSError` out parameter of type `NSErrorPointer`. This type is roughly equivalent to Objective-C's `NSError **`, with additional memory safety and optional typing. You can use the prefix `&` operator to pass in a reference to an optional `NSError` type as an `NSErrorPointer` object, as shown in the code listing below.

```
var writeError : NSError?
let written = myString.writeToFile(path, atomically: false,
    encoding: NSUTF8StringEncoding,
    error: &writeError)
if !written {
    if let error = writeError {
        println("write failure: \(error.localizedDescription)")
    }
}
```

When you implement your own functions that need to configure an `NSErrorPointer` object, you set the `NSErrorPointer` object's `memory` property to an `NSError` object you create. Make sure you check that the caller passed a non-`nil` `NSErrorPointer` object first:

```
func contentsForType(typeName: String! error: NSErrorPointer) -> AnyObject! {
    if cannotProduceContentsForType(typeName) {
        if error {
            error.memory = NSError(domain: domain, code: code, userInfo: [:])
```

```
        }
        return nil
    }
    // ...
}
```

Key-Value Observing

Information forthcoming.

Target-Action

Target-action is a common Cocoa design pattern in which one object sends a message to another object when a specific event occurs. The target-action model is fundamentally similar in Swift and Objective-C. In Swift, you use the `Selector` type to refer to Objective-C selectors. For an example of using target-action in Swift code, see Objective-C Selectors (page 18).

Introspection

In Objective-C, you use the `isKindOfClass:` method to check whether an object is of a certain class type, and the `conformsToProtocol:` method to check whether an object conforms to a specified protocol. In Swift, you accomplish this task by using the `is` operator to check for a type, or the `as?` operator to downcast to that type.

You can check whether an instance is of a certain subclass type by using the `is` operator. The `is` operator returns `true` if the instance is of that subclass type, and `false` if it is not.

```
if object is UIButton {
    // object is of type UIButton
} else {
    // object is not of type UIButton
}
```

You can also try and downcast to the subclass type by using the `as?` operator. The `as?` operator returns an optional value that can be bound to a constant using an `if-let` statement.

```
if let button = object as? UIButton {

    // object is successfully cast to type UIButton and bound to button

} else {

    // object could not be cast to type UIButton

}
```

For more information, see Type Casting.

Checking for and casting to a protocol follows exactly the same syntax as checking for and casting to a class. Here is an example of using the `as?` operator to check for protocol conformance:

```
if let dataSource = object as? UITableViewDataSource {

    // object conforms to UITableViewDataSource and is bound to dataSource

} else {

    // object not conform to UITableViewDataSource

}
```

Note that after this cast, the `dataSource` constant is of type `UITableViewDataSource`, so you can only call methods and access properties defined on the `UITableViewDataSource` protocol. You must cast it back to another type to perform other operations.

For more information, see Protocols.

Interacting with C APIs

As part of its interoperability with Objective-C, Swift maintains compatibility with a number of C language types and features. Swift also provides a way of working with common C constructs and patterns, in case your code requires it.

Primitive Types

Swift provides equivalents of C primitive integer types—for example, `char`, `int`, `float`, and `double`. However, there is no implicit conversion between these types and core Swift integer types, such as `Int`. Therefore, use these types if your code specifically requires them, but use `Int` wherever possible otherwise.

C Type	Swift Type
`bool`	`CBool`
`char`, `signed char`	`CChar`
`unsigned char`	`CUnsignedChar`
`short`	`CShort`
`unsigned short`	`CUnsignedShort`
`int`	`CInt`
`unsigned int`	`CUnsignedInt`
`long`	`CLong`
`unsigned long`	`CUnsignedLong`
`long long`	`CLongLong`
`unsigned long long`	`CUnsignedLongLong`
`wchar_t`	`CWideChar`
`char16_t`	`CChar16`

C Type	Swift Type
char32_t	CChar32
float	CFloat
double	CDouble

Enumerations

Swift imports as a Swift enumeration any C-style enumeration marked with the NS_ENUM macro. This means that the prefixes to enumeration value names are truncated when they are imported into Swift, whether they're defined in system frameworks or in custom code. For example, see this Objective-C enumeration:

```
typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
};
```

In Swift, it's imported like this:

```
enum UITableViewCellStyle: Int {
    case Default
    case Value1
    case Value2
    case Subtitle
}
```

When you refer to an enumeration value, use the value name with a leading dot (.).

```
let cellStyle: UITableViewCellStyle = .Default
```

Swift also imports options marked with the `NS_OPTIONS` macro. Whereas options behave similarly to imported enumerations, options can also support some bitwise operations, such as `&`, `|`, and `~`. In Objective-C, you represent an empty option set with the constant zero (0). In Swift, use `nil` to represent the absence of any options.

Pointers

Swift avoids giving you direct access to pointers whenever possible. However, there are various pointer types available for your use when you need direct access to memory. The following tables use `Type` as a placeholder type name to indicate syntax for the mappings.

For arguments, the following mappings apply:

C Syntax	Swift Syntax
const void *	CConstVoidPointer
void *	CMutableVoidPointer
const Type *	CConstPointer<Type>
Type *	CMutablePointer<Type>

For return types, variables, and argument types more than one pointer level deep, the following mappings apply:

C Syntax	Swift Syntax
void *	COpaquePointer
Type *	UnsafePointer<Type>

For class types, the following mappings apply:

C Syntax	Swift Syntax
Type * const *	CConstPointer<Type>
Type * __strong *	CMutablePointer<Type>
Type **	AutoreleasingUnsafePointer<Type>

C Mutable Pointers

When a function is declared as taking a `CMutablePointer<Type>` argument, it can accept any of the following:

- `nil`, which is passed as a null pointer

- A `CMutablePointer<Type>` value

- An in-out expression whose operand is a stored lvalue of type `Type`, which is passed as the address of the lvalue

- An in-out `Type[]` value, which is passed as a pointer to the start of the array, and lifetime-extended for the duration of the call

If you have declared a function like this one:

```
func takesAMutablePointer(x: CMutablePointer<Float>) { /*...*/ }
```

You can call it in any of the following ways:

```
var x: Float = 0.0
var p: CMutablePointer<Float> = nil
var a: Float[] = [1.0, 2.0, 3.0]
takesAMutablePointer(nil)
takesAMutablePointer(p)
takesAMutablePointer(&x)
takesAMutablePointer(&a)
```

When a function is declared as taking a `CMutableVoidPointer` argument, it can accept the same operands as `CMutablePointer<Type>` for any type `Type`.

If you have declared a function like this one:

```
func takesAMutableVoidPointer(x: CMutableVoidPointer)  { /* ... */ }
```

You can call it in any of the following ways:

```
var x: Float = 0.0, y: Int = 0
var p: CMutablePointer<Float> = nil, q: CMutablePointer<Int> = nil
var a: Float[] = [1.0, 2.0, 3.0], b: Int = [1, 2, 3]
```

```
takesAMutableVoidPointer(nil)

takesAMutableVoidPointer(p)

takesAMutableVoidPointer(q)

takesAMutableVoidPointer(&x)

takesAMutableVoidPointer(&y)

takesAMutableVoidPointer(&a)

takesAMutableVoidPointer(&b)
```

C Constant Pointers

When a function is declared as taking a `CConstPointer<Type>` argument, it can accept any of the following:

- `nil`, which is passed as a null pointer

- A `CMutablePointer<Type>`, `CMutableVoidPointer`, `CConstPointer<Type>`, `CConstVoidPointer`, or `AutoreleasingUnsafePointer<Type>` value, which is converted to `CConstPointer<Type>` if necessary

- An in-out expression whose operand is an lvalue of type `Type`, which is passed as the address of the lvalue

- A `Type[]` value, which is passed as a pointer to the start of the array, and lifetime-extended for the duration of the call

If you have declared a function like this one:

```
func takesAConstPointer(x: CConstPointer<Float>) { /*...*/ }
```

You can call it in any of the following ways:

```
var x: Float = 0.0

var p: CConstPointer<Float> = nil

takesAConstPointer(nil)

takesAConstPointer(p)

takesAConstPointer(&x)

takesAConstPointer([1.0, 2.0, 3.0])
```

When a function is declared as taking a `CConstVoidPointer` argument, it can accept the same operands as `CConstPointer<Type>` for any type `Type`.

If you have declared a function like this one:

```
func takesAConstVoidPointer(x: CConstVoidPointer)  { /* ... */ }
```

You can call it in any of the following ways:

```
var x: Float = 0.0, y: Int = 0
var p: CConstPointer<Float> = nil, q: CConstPointer<Int> = nil
takesAConstVoidPointer(nil)
takesAConstVoidPointer(p)
takesAConstVoidPointer(q)
takesAConstVoidPointer(&x)
takesAConstVoidPointer(&y)
takesAConstVoidPointer([1.0, 2.0, 3.0])
takesAConstVoidPointer([1, 2, 3])
```

AutoreleasingUnsafePointer

When a function is declared as taking an `AutoreleasingUnsafePointer<Type>`, it can accept any of the following:

- `nil`, which is passed as a null pointer

- An `AutoreleasingUnsafePointer<Type>` value

- An in-out expression, whose operand is primitive-copied to a temporary nonowning buffer. The address of that buffer is passed to the callee, and on return, the value in the buffer is loaded, retained, and reassigned into the operand.

Note that this list does not include arrays.

If you have declared a function like this one:

```
func takesAnAutoreleasingPointer(x: AutoreleasingUnsafePointer<NSDate?>) { /* ...
 */ }
```

You can call it in any of the following ways:

```
var x: NSDate? = nil
var p: AutoreleasingUnsafePointer<NSDate?> = nil
```

```
takesAnAutoreleasingPointer(nil)

takesAnAutoreleasingPointer(p)

takesAnAutoreleasingPointer(&x)
```

Note that C function pointers are not imported in Swift.

Global Constants

Global constants defined in C and Objective-C source files are automatically imported by the Swift compiler as Swift global constants.

Preprocessor Directives

The Swift compiler does not include a preprocessor. Instead, it takes advantage of compile-time attributes, build configurations, and language features to accomplish the same functionality. For this reason, preprocessor directives are not imported in Swift.

Simple Macros

Where you typically used the `#define` directive to define a primitive constant in C and Objective-C, in Swift you use a global constant instead. For example, the constant definition `#define FADE_ANIMATION_DURATION 0.35` can be better expressed in Swift with `let FADE_ANIMATION_DURATION = 0.35`. Because simple constant-like macros map directly to Swift global variables, the compiler automatically imports simple macros defined in C and Objective-C source files.

Complex Macros

Complex macros are used in C and Objective-C but have no counterpart in Swift. Complex macros are macros that do not define constants, including parenthesized, function-like macros. You use complex macros in C and Objective-C to avoid type-checking constraints or to avoid retyping large amounts of boilerplate code. However, macros can make debugging and refactoring difficult. In Swift, you can use functions and generics to achieve the same results without any compromises. Therefore, the complex macros that are in C and Objective-C source files are not made available to your Swift code.

Build Configurations

Swift code and Objective-C code are conditionally compiled in different ways. Swift code can be conditionally compiled based on the evaluation of *build configurations*. Build configurations include the literal `true` and `false` values, command line flags, and the platform-testing functions listed in the table below. You can specify command line flags using `–D <#flag#>`.

Function	Valid arguments
`os()`	`OSX`, `iOS`
`arch()`	`x86_64`, `arm`, `arm64`, `i386`

> **Note:** The `arch(arm)` build configuration does not return `true` for ARM 64 devices. The `arch(i386)` build configuration returns `true` when code is compiled for the 32-bit iOS simulator.

A simple conditional compilation statement takes the following form:

```
#if build configuration

    statements

#else

    statements

#endif
```

The *statements* consist of zero or more valid Swift statements, which can include expressions, statements, and control flow statements. You can add additional build configuration requirements to a conditional compilation statement with the `&&` and `||` operators, negate build configurations with the `!` operator, and add condition blocks with `#elseif`:

```
#if build configuration && !build configuration

    statements

#elseif build configuration

    statements

#else

    statements

#endif
```

In contrast with condition compilation statements in the C preprocessor, conditional compilation statements in Swift must completely surround blocks of code that are self-contained and syntactically valid. This is because all Swift code is syntax checked, even when it is not compiled.

Mix and Match

Swift and Objective-C in the Same Project

Swift's compatibility with Objective-C lets you create a project that contains files written in either language. You can use this feature, called *mix and match*, to write apps that have a mixed-language codebase. Using mix and match, you can implement part of your app's functionality using the latest Swift features and seamlessly incorporate it back into your existing Objective-C codebase.

Mix and Match Overview

Objective-C and Swift files can coexist in a single project, whether the project was originally an Objective-C or Swift project. You can simply add a file of the other language directly to an existing project. This natural workflow makes creating mixed-language app and framework targets as straightforward as creating an app or framework target written in a single language.

The process for working with mixed-language targets differs slightly depending on whether you're writing an app or a framework. The general import model for working with both languages within the same target is depicted below and described in more detail in the following sections.
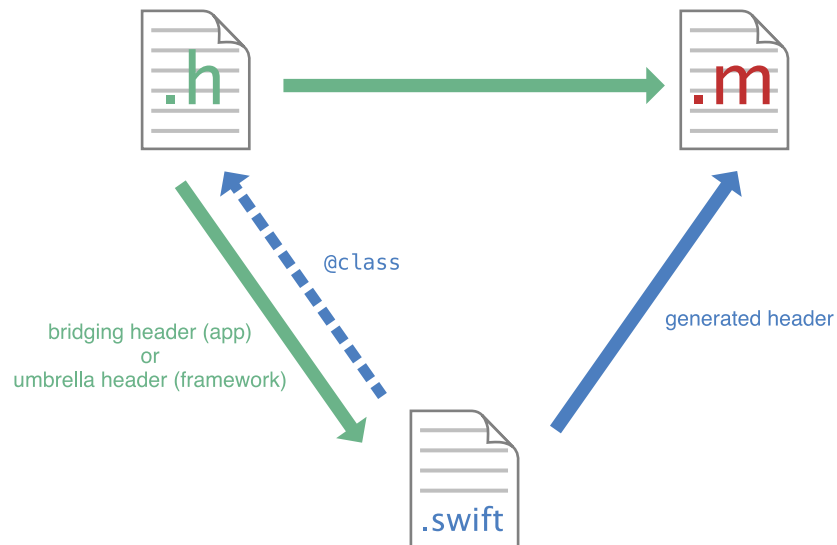
Importing Code from Within the Same App Target

If you're writing a mixed-language app, you may need to access your Objective-C code from Swift and your Swift code from Objective-C. The process described in this section applies to non-framework targets.

Importing Objective-C into Swift

To import a set of Objective-C files in the same app target as your Swift code, you rely on an *Objective-C bridging header* to expose those files to Swift. Xcode offers to create this header file when you add a Swift file to an existing Objective-C app, or an Objective-C file to an existing Swift app.
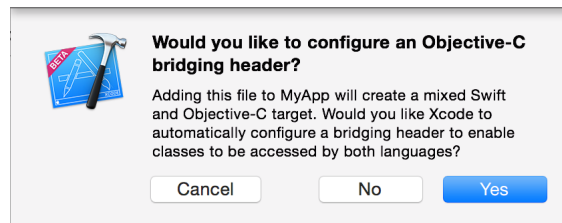
If you accept, Xcode creates the header file along with the file you were creating, and names it by your product module name followed by adding "-Bridging-Header.h". For information on the product module name, see Naming Your Product Module (page 50).

You'll need to edit this file to expose your Objective-C code to your Swift code.

To import Objective-C code into Swift from the same target

1. In your Objective-C bridging header file, import every Objective-C header you want to expose to Swift. For example:

```
#import "XYZCustomCell.h"
#import "XYZCustomView.h"
#import "XYZCustomViewController.h"
```

2. Under Build Settings, make sure the Objective-C Bridging Header build setting under Swift Compiler - Code Generation has a path to the header. The path must be directly to the file itself, not the directory that it's in.

   The path should be relative to your project, similar to the way your Info.plist path is specified in Build Settings. In most cases, you should not need to modify this setting.

Any public Objective-C headers listed in this bridging header file will be visible to Swift. The Objective-C functionality will be available in any Swift file within that target automatically, without any import statements. Use your custom Objective-C code with the same Swift syntax you use with system classes.

```
let myCell = XYZCustomCell()
myCell.subtitle = "A custom cell"
```

Importing Swift into Objective-C

When you import Swift code into Objective-C, you rely on an *Xcode-generated header* file to expose those files to Objective-C. This automatically-generated file is an Objective-C header that declares all of the Swift interfaces in your target. It can be thought of as an umbrella header for your Swift code. The name of this header is your product module name followed by adding "-Swift.h". For information on the product module name, see Naming Your Product Module (page 50).

You don't need to do anything special to create this file—you just need to import it to use its contents in your Objective-C code. Note that the Swift interfaces in the generated header include references to all of the Objective-C types used in them. If you use your own Objective-C types in your Swift code, make sure to import the Objective-C headers for those types prior to importing the Swift generated header into the Objective-C `.m` file you want to access the Swift code from.

To import Swift code into Objective-C from the same target

- Import the Swift code from that target into any Objective-C `.m` file within that target using this syntax, and substituting the appropriate name:

  ```
  #import "ProductModuleName−Swift.h"
  ```

Any Swift files in your target will be visible in Objective-C `.m` files containing this import statement. For information on using Swift from Objective-C code, see Using Swift from Objective-C (page 48).

	Import into Swift	Import into Objective-C
Swift code	No import statement	`#import "ProductModuleName−Swift.h"`
Objective-C code	No import statement; Objective-C bridging header required	`#import "Header.h"`

Importing Code from Within the Same Framework Target

If you're writing a mixed-language framework, you may need to access your Objective-C code from Swift and your Swift code from Objective-C.

Importing Objective-C into Swift

To import a set of Objective-C files in the same framework target as your Swift code, you'll need to import those files into the Objective-C umbrella header for the framework.

To import Objective-C code into Swift from the same framework

1. Under Build Settings, in Packaging, make sure the Defines Module setting for that framework target is set to Yes.

2. In your umbrella header file, import every Objective-C header you want to expose to Swift. For example:

```
#import <XYZ/XYZCustomCell.h>
#import <XYZ/XYZCustomView.h>
#import <XYZ/XYZCustomViewController.h>
```

Swift will see every header you expose publicly in your umbrella header. The contents of the Objective-C files in that framework will be available in any Swift file within that framework target automatically, without any import statements. Use your custom Objective-C code with the same Swift syntax you use with system classes.

```
let myCell = XYZCustomCell()
myCell.subtitle = "A custom cell"
```

Importing Swift into Objective-C

To import a set of Swift files in the same framework target as your Objective-C code, you don't need to import anything into the umbrella header for the framework. Instead, import the Xcode-generated header file for your Swift code into any Objective-C `.m` file you want to use that code from.

To import Swift code into Objective-C from the same framework

1. Under Build Settings, in Packaging, make sure the Defines Module setting for that framework target is set to Yes.

2. Import the Swift code from that framework target into any Objective-C `.m` file within that framework target using this syntax, and substituting the appropriate names:

```
#import <ProductName/ProductModuleName−Swift.h>
```

Any Swift files in your framework target will be visible in Objective-C `.m` files containing this import statement. For information on using Swift from Objective-C code, see Using Swift from Objective-C (page 48).

	Import into Swift	Import into Objective-C
Swift code	No import statement	`#import <ProductName/ProductModuleName−Swift.h>`
Objective-C code	No import statement; Objective-C umbrella header required	`#import "Header.h"`

Importing External Frameworks

You can import external frameworks that have a pure Objective-C codebase, a pure Swift codebase, or a mixed-language codebase. The process for importing an external framework is the same whether the framework is written in a single language or contains files from both languages. When you import an external framework, make sure the Defines Module build setting for the framework you're importing is set to Yes.

You can import a framework into any Swift file within a different target using the following syntax:

```
import FrameworkName
```

You can import a framework into any Objective-C `.m` file within a different target using the following syntax:

```
@import FrameworkName;
```

	Import into Swift	Import into Objective-C
Any language framework	`import FrameworkName`	`@import FrameworkName;`

Using Swift from Objective-C

Once you import your Swift code into Objective-C, use regular Objective-C syntax for working with Swift classes.

```
MySwiftClass *swiftObject = [[MySwiftClass alloc] init];

[swiftObject swiftMethod];
```

A Swift class or protocol must be marked with the `@objc` attribute to be accessible and usable in Objective-C. This attribute tells the compiler that this piece of Swift code can be accessed from Objective-C. If your Swift class is a descendant of an Objective-C class, the compiler automatically adds the `@objc` attribute for you. For more information, see Swift Type Compatibility (page 17).

You'll have access to anything within a class or protocol that's marked with the `@objc` attribute as long as it's compatible with Objective-C. This excludes Swift-only features such as those listed here:

- Generics

- Tuples

- Enumerations defined in Swift

- Structures defined in Swift

- Top-level functions defined in Swift

- Global variables defined in Swift

- Typealiases defined in Swift

- Swift-style variadics

- Nested types

- Curried functions

For example, a method that takes a generic type as an argument or returns a tuple will not be usable from Objective-C.

To avoid cyclical references, don't import Swift into an Objective-C header file. Instead, you can forward declare a Swift class to use it in an Objective-C header. However, note that you cannot subclass a Swift class in Objective-C.

To reference a Swift class in an Objective-C header file

- Forward declare the Swift class you're using:

```
// MyObjcClass.h


@class MySwiftClass;
```

```
@interface MyObjcClass : NSObject
- (MySwiftClass *)returnSwiftObject;
/* ... */
@end
```

Naming Your Product Module

The name of the Xcode-generated header for Swift code, and the name of the Objective-C bridging header that Xcode creates for you, are generated from your product module name. By default, your product module name is the same as your product name. However, if your product name has any nonalphanumeric characters, such as a period (.), they are replaced with an underscore (_) in your product module name. If the name begins with a number, the first number is replaced with an underscore.

You can also provide a custom name for the product module name, and Xcode will use this when naming the bridging and generated headers. To do this, change the Product Module Name build setting.

Troubleshooting Tips and Reminders

- Treat your Swift and Objective-C files as the same collection of code, and watch out for naming collisions.

- If you're working with frameworks, make sure the Defines Module build setting under Packaging is set to Yes.

- If you're working with the Objective-C bridging header, make sure the Objective-C Bridging Header build setting under Swift Compiler - Code Generation has a path to the header that's relative to your project. The path must be directly to the file itself, not just to the directory that it's in.

- Xcode uses your product module name—not your target name—when naming the Objective-C bridging header and the generated header for your Swift code. For information on product module naming, see Naming Your Product Module (page 50).

- To be accessible and usable in Objective-C, a Swift class must be a descendant of an Objective-C class or it must be marked `@objc`.

- When you bring Swift code into Objective-C, remember that Objective-C won't be able to translate certain features that are specific to Swift. For a list, see Using Swift from Objective-C (page 48).

- If you use your own Objective-C types in your Swift code, make sure to import the Objective-C headers for those types prior to importing the Swift generated header into the Objective-C `.m` file you want to access the Swift code from.

Migration

- [Migrating Your Objective-C Code to Swift](navigation) (page 52)

Migrating Your Objective-C Code to Swift

Migration provides an opportunity to revisit an existing Objective-C app and improve its architecture, logic, and performance by replacing pieces of it in Swift. For a straightforward, incremental migration of an app, you'll be using the tools learned earlier—mix and match plus interoperability. Mix-and-match functionality makes it easy to choose which features and functionality to implement in Swift, and which to leave in Objective-C. Interoperability makes it possible to integrate those features back into Objective-C code with no hassle. Use these tools to explore Swift's extensive functionality and integrate it back into your existing Objective-C app without having to rewrite the entire app in Swift at once.

Preparing Your Objective-C Code for Migration

Before you begin migrating your codebase, make sure that your Objective-C and Swift code will have optimal compatibility. This means tidying up and modernizing your existing Objective-C codebase. Your existing code should follow modern coding practices to make it easier to interact with Swift seamlessly. For a short list of practices to adopt before moving forward, see *Adopting Modern Objective-C*.

The Migration Process

The most effective approach for migrating code to Swift is on a per-file basis—that is, one class at a time. Because you can't subclass Swift classes in Objective-C, it's best to choose a class in your app that doesn't have any subclasses. You'll replace the `.m` and `.h` files for that class with a single `.swift` file. Everything from your implementation and interface will go directly into this single Swift file. You won't create a header file; Xcode generates a header automatically in case you need to reference it.

Before You Start

- ✓ Create a Swift class for your corresponding Objective-C `.m` and `.h` files by choosing File > New > File > (iOS *or* OS X) > Other > Swift. You can use the same or a different name than your Objective-C class. Class prefixes are optional in Swift.

- ✓ Import relevant system frameworks.

- ✓ Fill out an Objective-C bridging header if you need to access Objective-C code from the same app target in your Swift file. For instructions, see .

✔ To make your Swift class accessible and usable back in Objective-C, make it a descendant of an Objective-C class or mark it with the `@objc` attribute. To specify a particular name for the class to use in Objective-C, mark it with `@objc(<#name#>)`, where `<#name#>` is the name that your Objective-C code will use to reference the Swift class. For more information on `@objc`, see Swift Type Compatibility (page 17).

As You Work

✔ You can set up your Swift class to integrate Objective-C behavior by subclassing Objective-C classes, adopting Objective-C protocols, and more. For more information, see Writing Swift Classes with Objective-C Behavior (page 19).

✔ As you work with Objective-C APIs, you'll need to know how Swift translates certain Objective-C language features. For more information, see Interacting with Objective-C APIs (page 10).

✔ When writing Swift code that incorporates Cocoa frameworks, remember that certain types are bridged, which means you can work with Swift types in place of Objective-C types. For more information, see Working with Cocoa Data Types (page 23).

✔ As you incorporate Cocoa patterns into your Swift class, see Adopting Cocoa Design Patterns (page 30) for information on translating common design patterns.

✔ For considerations on translating your properties from Objective-C to Swift, read Properties.

✔ Use the `@objc(<#name#>)` attribute to provide Objective-C names for properties and methods when necessary. For example, you can mark a property called `enabled` to have a getter named `isEnabled` in Objective-C like this:

```
var enabled: Bool {
  @objc(isEnabled) get {
    /* ... */
  }
}
```

✔ Denote instance (−) and class (+) methods with `func` and `class func`, respectively.

✔ Declare simple macros as global constants, and translate complex macros into functions.

After You Finish

✔ Update import statements in your Objective-C code (to `#import "ProductModuleName−Swift.h"`), as described in Importing Code from Within the Same App Target (page 45).

✔ Remove the original Objective-C `.m` file from the target by deselecting the target membership checkbox. Don't delete the `.m` and `.h` files immediately; use them to troubleshoot.

✔ Update your code to use the Swift class name instead of the Objective-C name if you gave the Swift class a different name.

Troubleshooting Tips and Reminders

Each migration experience is different depending on your existing codebase. However, there are some general steps and tools to help you troubleshoot your code migration:

- Remember that you cannot subclass a Swift class in Objective-C. Therefore, the class you migrate cannot have any Objective-C subclasses in your app.

- Once you migrate a class to Swift, you must remove the corresponding `.m` file from the target before building to avoid a duplicate symbol error.

- To be accessible and usable in Objective-C, a Swift class must be a descendant of an Objective-C class or it must be marked `@objc`.

- When you bring Swift code into Objective-C, remember that Objective-C won't be able to translate certain features that are specific to Swift. For a list, see Using Swift from Objective-C (page 48).

- Command-click a Swift class name to see its generated header.

- Option-click a symbol to see implicit information about it, like its type, attributes, and documentation comments.