



Using Swift with Cocoa and Objective-C

Swift 3 Edition



Getting Started

Basic Setup

Swift is designed to provide seamless compatibility with Cocoa and Objective-C. You can use Objective-C APIs in Swift, and you can use Swift APIs in Objective-C. This makes Swift an easy, convenient, and powerful tool to integrate into your development workflow.

This guide covers three important aspects of Swift and Objective-C 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 Cocoa app development using Swift, create a new Swift project from one of the provided Xcode templates.

To create a Swift project in Xcode

1. Choose File > New > Project > (iOS, watchOS, tvOS, or macOS) > Application > *your template of choice*.
2. Click the Language pop-up menu and choose Swift.

Choose options for your new project:

Product Name:

Organization Name:

Organization Identifier:

Bundle Identifier:

Language: 

Devices: 

Use Core Data

Include Unit Tests

Include UI Tests

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—all of the information about a class, function, or constant resides in a single `.swift` file. This is discussed in more detail in [Swift and Objective-C in the Same Project](#).

From here, you can start experimenting by writing Swift code in the app delegate or a new Swift file you create by choosing File > New > File > (iOS, watchOS, tvOS, *or* macOS) > Source > Swift.

Requirements

Creating an app using Swift 3.0 requires Xcode 8.0 or newer, as

well as the following base SDK requirements:

Platform	Base SDK Requirement
macOS	10.12
iOS	10.0
watchOS	3.0
tvOS	10.0

The Swift compiler and Xcode enforce a minimum deployment target of iOS 7 or macOS 10.9. Setting an earlier deployment target results in a build failure.

NOTE

Executables built from the command line expect to find the Swift libraries in their `@rpath`. If you plan to ship a Swift executable built from the command line, you'll need to ship the Swift dynamic libraries as well. Swift executables built from within Xcode have the runtime statically linked.

Understanding the Swift Import Process

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

Any Objective-C framework or C library that supports *modules* 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 use Foundation APIs from a Swift file, add the following import statement to the top of the file:



```
import Foundation
```

With this import statement, that Swift file can now access all of Foundation’s classes, protocols, methods, properties, and constants.

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 process 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 have the following effects:

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

For more information on using Objective-C in Swift, see [Interacting with Objective-C APIs](#).

NOTE

You cannot import C++ code directly into Swift. Instead, create an Objective-C or C wrapper for C++ 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](#).

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 using Swift initializer syntax.

Objective-C initializers begin with `init`, or `initWith`: if the initializer takes one or more arguments. When an Objective-C initializer is imported by Swift, the `init` prefix becomes an `init` keyword to indicate that the method is a Swift initializer. If the initializer takes an argument, the `With` is removed and the rest of the selector is divided up into named parameters accordingly.

Consider the following Objective-C initializer declarations:

```
1 - (instancetype)init;  
2 - (instancetype)initWithFrame:(CGRect)frame  
3 style:  
    (UITableViewStyle)style;
```

Here are the equivalent Swift initializer declarations:

```
1 init() { /* ... */ }  
2 init(frame: CGRect, style: UITableViewStyle) { /*  
    ... */ }
```

The differences between Objective-C and Swift syntax are all the more apparent when instantiating objects.

In Objective-C, you do this:

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

In Swift, you do this:

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

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

You can provide an explicit type when assigning to a constant or variable, or you can omit the type and have Swift infer the type automatically from the initializer.

```
let myTextField = UITextField(frame: CGRect(x: 0.0,  
                                             y: 0.0, width: 200.0, height: 40.0))
```

These `UITableView` and `UITextField` objects are the same objects that you'd instantiate 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 their respective types.

Class Factory Methods and Convenience Initializers

For consistency and simplicity, Objective-C class factory methods are imported as convenience initializers in Swift. This allows them to be used with the same 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)
```

Failable Initialization

In Objective-C, initializers directly return the object they initialize. To inform the caller when initialization has failed, an Objective-C initializer can return `nil`. In Swift, this pattern is built into a language feature called *failable initialization*.

Many Objective-C initializers in system frameworks have been audited to indicate whether initialization can fail. You can indicate whether initializers in your own Objective-C classes can fail using *nullability annotations*, as described in [Nullability and Optionals](#). Objective-C initializers that indicate whether they're failable are imported as either `init(...)`—if initialization cannot fail—or `init?(...)`—if initialization can fail. Otherwise, Objective-C initializers are imported as `init!(...)`.

For example, the `UIImage(contentsOfFile:)` initializer can fail to initialize a `UIImage` object if an image file doesn't exist at the provided path. You can use optional binding to unwrap the result of a failable initializer if initialization is successful.

```
1  if let image = UIImage(contentsOfFile:  
2      "MyImage.png") {  
3      // loaded the image successfully  
4  } else {  
5      // could not load the image  
6  }
```

Accessing Properties

Objective-C property declarations using the `@property` syntax are imported as Swift properties in the following way:

- Properties with the nullability property attributes (`nonnull`, `nullable`, and `null_resettable`) are imported as Swift properties with optional or nonoptional type as described in [Nullability and Optionals](#).
- Properties with the `readonly` property attribute are imported as Swift computed properties with a getter (`{ get }`).
- Properties with the `weak` property attribute are imported as Swift properties marked with the `weak` keyword (`weak var`).
- Properties with an ownership property attribute other than `weak` (that is, `assign`, `copy`, `strong`, or `unsafe_unretained`)

are imported as Swift properties with the appropriate storage.

- Properties with the `class` property attribute are imported as Swift type properties.
- Atomicity property attributes (`atomic` and `nonatomic`) are not reflected in the corresponding Swift property declaration, but the atomicity guarantees of the Objective-C implementation still hold when the imported property is accessed from Swift.
- Accessor property attributes (`getter=` and `setter=`) are ignored by Swift.

You access properties on Objective-C objects in Swift using dot syntax, using the name of the property without parentheses.

For example, you can set the `textColor` and `text` properties of a

`UITextField` object with the following code:

```
1 myTextField.textColor = UIColor.darkGray  
2 myTextField.text = "Hello world"
```

Objective-C methods that return a value and take no arguments can be called like an Objective-C property using dot syntax. However, these are imported by Swift as instance methods, as only Objective-C `@property` declarations are imported by Swift as properties. Methods are imported and called as described in [Working with Methods](#).

Working with Methods

You can call Objective-C methods from Swift using dot syntax.

When Objective-C methods are imported into Swift, the first part of the Objective-C selector becomes the base method name and appears before the parentheses. The first argument appears immediately inside the parentheses, without a name. The rest of the selector pieces correspond to argument names and appear 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, at: 2)
```

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

```
myTableView.layoutIfNeeded()
```

id Compatibility

The Objective-C `id` type is imported by Swift as the `Any` type. At compile time and runtime, the compiler introduces a universal bridging conversion operation when a Swift value or object is passed into Objective-C as an `id` parameter. When `id` values are imported into Swift as `Any`, the runtime automatically handles bridging back to either class references or Swift value types.

```
1 var x: Any = "hello" as String  
2 x as? String // String with value "hello"  
3 x as? NSString // NSString with value "hello"  
4  
5 x = "goodbye" as NSString  
6 x as? String // String with value "goodbye"  
7 x as? NSString // NSString with value "goodbye"
```

Downcasting Any

When working with objects of type `Any` where the underlying type is known or could be reasonably determined, it is often useful to downcast those objects to a more specific type. However, because the `Any` type can refer to any type, a downcast to a more specific type is not guaranteed to succeed.

You can use the conditional type cast operator (`as?`), which returns an optional value of the type you are trying to downcast to:

```
1 let userDefaults = UserDefaults.standard  
2 let lastRefreshDate = userDefaults.object(forKey:  
    "LastRefreshDate") // lastRefreshDate is of  
    type Any?  
3  
4 if let date = lastRefreshDate as? Date {  
    print("\(date.timeIntervalSinceReferenceDate)")  
5 }
```

If you are certain of the type of the object, you can use the forced downcast operator (`as!`) instead.

```
1 let myDate = lastRefreshDate as! Date  
2 let timeInterval =  
    myDate.timeIntervalSinceReferenceDate
```

However, if a forced downcast fails, a runtime error is triggered:

```
let myDate = lastRefreshDate as! String // Error
```

Dynamic Method Lookup

Swift also includes an `AnyObject` type that represents some kind of object and has the special ability to look up any `@objc` method

dynamically. This allows you to write maintain flexibility of untyped access to Objective-C APIs that return `id` values.

For example, you can assign an object of any class type to a constant or variable of `AnyObject` type and reassign a variable to an object of a different type. You can also call any Objective-C method and access any property on an `AnyObject` value without casting to a more specific class type.

```
1 var myObject: AnyObject = UITableViewCell()  
2  
3 myObject = NSDate()  
4  
5 let futureDate = myObject.addTimeInterval(10)  
6  
7 let timeSinceNow = myObject.timeIntervalSinceNow
```

Unrecognized Selectors and Optional Chaining

Because the specific type of an `AnyObject` value is not known until runtime, it is possible to inadvertently write unsafe code. In Swift as well as Objective-C, attempting to call a method that does not exist triggers an unrecognized selector error.

For example, the following code compiles without a compiler warning, but triggers an error at runtime:

```
1 myObject.character(at: 5)  
2 // crash, myObject doesn't respond to that method
```

Swift uses optionals to guard against such unsafe behavior. When

you call a method on a value of `AnyObject` type, that method call 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`.

For example, in the code listing below, the first and second lines are not executed because the `count` property and the `character(at:)` method do not exist on an `NSDate` object. The `myCount` 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.

```
1 // myObject has AnyObject type and NSDate value  
2 let myCount = myObject.count  
3 // myCount has Int? type and nil value  
4 let myChar = myObject.character?(at: 5)  
5 // myChar has unichar? type and nil value  
6 if let fifthCharacter = myObject.character?(at: 5)  
    {  
        7 print("Found \(fifthCharacter) at index 5")  
8    }  
9 // conditional branch not executed
```

NOTE

Although Swift does not require forced unwrapping when calling methods on values of type `AnyObject`, it is recommended as a way to safeguard against unexpected behavior.

Nullability and Optionals

In Objective-C, you work with references to objects using raw pointers that could be `NULL` (referred to as `nil` in Objective-C). In Swift, all values—including structures and object references—are guaranteed to be non-null. 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`. For more information about optionals, see Optionals in *The Swift Programming Language (Swift 3)*.

Objective-C can use nullability annotations to designate whether a parameter type, property type, or return type, can have a `NULL` or `nil` value. Individual type declarations can be audited using the `_Nullable` and `_Nonnull` annotations, individual property declarations can be audited using the `nullable`, `nonnull` and `null_resettable` property attributes, or entire regions can be audited for nullability using the `NS_ASSUME_NONNULL_BEGIN` and `NS_ASSUME_NONNULL_END` macros. If no nullability information is provided for a type, Swift cannot distinguish between optional and nonoptional references, and imports it as an implicitly unwrapped optional.

- Types declared to be *nonnullable*, either with a `_Nonnull` annotation or in an audited region, are imported by Swift as

a *nonoptional*.

- Types declared to be *nullable* with a `_Nullable` annotation, are imported by Swift as an *optional*.
- Types declared without a nullability annotation are imported by Swift as an *implicitly unwrapped optional*.

For example, consider the following Objective-C declarations:

```
1 @property (nullable) id nullableProperty;  
2 @property (nonnull) id nonNullProperty;  
3 @property id unannotatedProperty;  
4  
5 NS_ASSUME_NONNULL_BEGIN
```

- 6 – (`id`)`returnsNonNullValue`;
- 7 – (`void`)`takesNonNullParameter:(id)value`;
- 8 `NS_ASSUME_NONNULL_END`
- 9
- (`nullable id`)`returnsNullableValue`;
- (`void`)`takesNullableParameter:(nullable id)value`;
- (`id`)`returnsUnannotatedValue`;
- (`void`)`takesUnannotatedParameter:(id)value`;

Here's how they're imported by Swift:

```
1 var nullableProperty: Any?  
2  
3 var nonNullProperty: Any  
4  
5 func returnsNonNullValue() -> Any  
6  
7 func takesNonNullParameter(value: Any)  
8  
9 func returnsNullableValue() -> Any?  
10 func takesNullableParameter(value: Any?)
```

```
func returnsUnannotatedValue() -> Any!
```

```
func takesUnannotatedParameter(value: Any!)
```

Most of the Objective-C system frameworks, including Foundation, already provide nullability annotations, allowing you to work with values in an idiomatic and type-safe manner.

Protocol-Qualified Classes

Objective-C protocol-qualified classes are imported by Swift as protocol type values. For example, given the following Objective-C method that performs an operation on the specified class:

```
- (void)doSomethingForClass:  
    (Class<NSCoding>)codingClass;
```

Here's how Swift imports it:

```
func doSomething(for codingClass: NSCoding.Type)
```

Lightweight Generics

Objective-C type declarations using lightweight generic parameterization are imported by Swift with information about the type of their contents preserved. For example, given the following Objective-C property declarations:

```
1 @property NSArray<NSDate *> *dates;  
2 @property NSCache<NSObject *,  
    id<NSDiscardableContent>> *cachedData;  
3 @property NSDictionary <NSString *,  
    NSArray<NSLocale *>> *supportedLocales;
```

Here's how Swift imports them:

```
1 var dates: [Date]  
2 var cachedData: NSCache<AnyObject,  
    NSDiscardableContent>  
3 var supportedLocales: [String: [Locale]]
```

A parameterized class written in Objective-C is imported into Swift as a generic class with the same number of type parameters. All Objective-C generic type parameters imported by Swift have a type constraint that requires that type to be a class (`T: Any`). If the Objective-C generic parameterization specifies a class qualification, the imported Swift class has a constraint that requires that type to be a subclass of the specified class. If the Objective-C generic parameterization specifies a protocol qualification, the imported Swift class has a constraint that requires that type to conform to the

specified protocol. For an unspecialized Objective-C type, Swift infers the generic parameterization for the imported class type constraints. For example, given the following Objective-C class and category declarations:

```
1 @interface List<T: id<NSCopying>> : NSObject
2 - (List<T> *)listByAppendingItemsInList:(List<T>
3                                     *)otherList;
4
5 @end
6
7 @interface ListContainer : NSObject
8 - (List<NSValue *> *)listOfValues;
```

```
7 @end  
8  
9 @interface ListContainer : ObjectList  
- (List *)listOfObjects;  
@end
```

Here's how they're imported by Swift:

```
1 class List<T: NSCopying> : NSObject {  
2     func listByAppendingItemsInList(otherList:  
        List<T>) -> List<T>
```

```
3    }
4
5    class ListContainer : NSObject {
6        func listOfValues() -> List<NSValue>
7    }
8
9    extension ListContainer {
10       func listOfObjects() -> List<NSCopying>
11   }
```

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.

```
1  extension UIBezierPath {  
2      convenience init(triangleSideLength: CGFloat,
```

```
origin: CGPoint) {  
    self.init()  
  
    let squareRoot = CGFloat(sqrt(3.0))  
  
    let altitude = (squareRoot *  
        triangleSideLength) / 2  
  
    move(to: origin)  
  
    addLine(to: CGPoint(x: origin.x +  
        triangleSideLength, y: origin.y))  
  
    addLine(to: CGPoint(x: origin.x +  
        triangleSideLength / 2, y: origin.y +
```

```
    altitude))  
9     close()  
}  
}  
}
```

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:

```
1 extension CGRect {  
2     var area: CGFloat {  
3         return width * height  
4     }  
5 }  
6 let rect = CGRect(x: 0.0, y: 0.0, width: 10.0,  
7                     height: 50.0)  
8 let area = rect.area
```

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 with Objective-C block calling convention, denoted by the `@convention(block)` attribute. For example, here is an Objective-C block variable:

```
1 void (^completionBlock)(NSData *) = ^(NSData *data)
2 {
3     // ...
4 }
```

And here's what it looks like in Swift:

```
1 let completionBlock: (Data) -> Void = { data in
2     // ...
3 }
```

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.

Avoiding Strong Reference Cycles When Capturing `self`

In Objective-C, if you need to capture `self` in a block, it's important to consider the memory management implications.

Blocks maintain strong references to any captured objects, including `self`. If `self` maintains a strong reference to the block, such as a

copying property, this would create a strong reference cycle. To avoid this, you can instead have the block capture a weak reference to `self`:

```
1  __weak typeof(self) weakSelf = self;  
2  
3  self.block = ^{  
4      __strong typeof(self) strongSelf = weakSelf;  
5      [strongSelf doSomething];  
6  };
```

Like blocks in Objective-C, closures in Swift also maintain strong references to any captured objects, including `self`. To prevent a strong reference cycle, you can specify `self` to be `unowned` in a

closure's capture list:

```
1 self.closure = { [unowned self] in  
2     self.doSomething()  
3 }
```

For more information, see Resolving Strong Reference Cycles for Closures in *The Swift Programming Language (Swift 3)*.

Object Comparison

There are two distinct types of comparison you can make between 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 provides default implementations of the `==` and `==≡` operators and adopts the `Equatable` protocol for objects that derive from the `NSObject` class. The default implementation of the `==` operator invokes the `isEqual:` method, and the default implementation of the `==≡` operator checks pointer equality. You should not override the equality or identity operators for types imported from Objective-C.

The base implementation of the `isEqual:` provided by the `NSObject` class is equivalent to an identity check by pointer equality. You can override `isEqual:` in a subclass to have Swift and Objective-C APIs determine equality based on the contents of objects rather than their identities. For more information about implementing comparison logic, see Object comparison in *Cocoa Core Competencies*.

NOTE

Swift automatically provides implementations for the logical complements of the equality and identity operators (`!=` and `!==`). These should not be overridden.

Hashing

Swift imports Objective-C declarations of `NSDictionary` that don't specify a class qualification for the key type as a `Dictionary` with the `Key` type `AnyHashable`. Similarly, `NSSet` declarations without a class-qualified object type are imported by Swift as a `Set` with the `Element` type `AnyHashable`. If an `NSDictionary` or `NSSet` declaration does parameterize its key or object type, respectively, that type is used instead. For example, given the following Objective-C

declarations:

```
1 @property NSDictionary *unqualifiedDictionary;  
2 @property NSDictionary<NSString *, NSDate *>  
    *qualifiedDictionary;  
3  
4 @property NSSet *unqualifiedSet;  
5 @property NSSet<NSString *> *qualifiedSet;
```

Here's how Swift imports them:

```
1 var unqualifiedDictionary: [AnyHashable: Any]  
2  
3  
4 var unqualifiedSet: Set<AnyHashable>  
5 var qualifiedSet: Set<String>
```

The `AnyHashable` type is used by Swift when importing Objective-C declarations with an unspecified or `id` type that cannot be otherwise be imported as `Any` because the type needs to conform to the `Hashable` protocol. The `AnyHashable` type is implicitly converted from any `Hashable` type, and you can use the `as?` and `as!` operators to cast from `AnyHashable` to a more specific type.

For more information, see the [AnyHashable API reference](#).

Swift Type Compatibility

When you create a Swift class that descends from an Objective-C class, the class and its members—properties, methods, subscripts, and initializers—that are compatible with Objective-C are automatically available from Objective-C. This excludes Swift-only features, such as those listed here:

- Generics
- Tuples
- Enumerations defined in Swift without `Int` raw value type
- 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

Swift APIs are translated into Objective-C similar to how Objective-C APIs are translated into Swift, but in reverse:

- Swift optional types are annotated as `__nullable`.
- Swift nonoptional types are annotated as `__nonnull`.
- Swift constant stored properties and computed properties become read-only Objective-C properties.
- Swift variable stored properties become read-write

Objective-C properties.

- Swift type properties become Objective-C properties with the `class` property attribute.
- Swift type methods become Objective-C class methods.
- Swift initializers and instance methods become Objective-C instance methods.
- Swift methods that throw errors become Objective-C methods with an `NSError **` parameter. If the Swift method has no parameters, `AndReturnError:` is appended to the Objective-C method name, otherwise `error:` is appended. If a Swift method does not specify a return type, the corresponding Objective-C method has a `BOOL` return type. If the Swift method returns a nonoptional type, the corresponding Objective-C method has an optional return type.

For example, consider the following Swift declaration:

```
1 class Jukebox: NSObject {  
2     var library: Set<String>  
3  
4     var nowPlaying: String?  
5  
6     var isCurrentlyPlaying: Bool {  
7         return nowPlaying != nil  
8     }  
9 }
```

```
class var favoritesPlaylist: [String] {  
    // return an array of song names  
}  
  
init(songs: String...) {  
    self.library = Set<String>(songs)  
}  
  
func playSong(named name: String) throws {  
    // play song or throw an error if unavailable
```

```
    }  
}  
}
```

Here's how it's imported by Objective-C:

```
1 @interface Jukebox : NS0bject  
2  
3     @property (nonatomic, strong, nonnull)  
4             NSSet<NSString *> *library;  
5  
6     @property (nonatomic, copy, nullable) NSString  
7             *nowPlaying;  
8  
9     @property (nonatomic, readonly,  
10             getter=isCurrentlyPlaying) BOOL
```

```
        currentlyPlaying;  
5     @property (nonatomic, class, readonly, nonnull)  
          NSArray<NSString *> * favoritesPlaylist;  
6     - (nonnullinstancetype)initWithSongs:  
          (NSArray<NSString *> * __nonnull)songs  
         _OBJC_DESIGNATED_INITIALIZER;  
7     - (BOOL)playSong:(NSString * __nonnull)name error:  
          (NSError * __nullable *  
           __null_unspecified)error;  
8     @end
```

NOTE

You cannot subclass a Swift class in Objective-C.

Configuring Swift Interfaces in Objective-C

In some cases, you need finer grained control over how your Swift API is exposed to Objective-C. You can use the `@objc(name)` attribute to change the name of a class, property, method, enumeration type, or enumeration case declaration in your interface as it's exposed to Objective-C code.

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.

```
1 @objc(Color)
2 enum Цвет: Int {
3     @objc(Red)
4     case Красный
5
6     @objc(Black)
7     case Черный
8 }
```

```
@objc(Squirrel)

class Белка: NSObject {

    @objc(color)

    var цвет: Цвет = .Красный

    @objc(initWithName:)
    init (имя: String) {

        // ...
    }
}
```

```
@objc(hideNuts:inTree:)

func прячьОрехи(количество: Int, вДереве дерево:

    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 migrating 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.

NOTE

Conversely, Swift also provides the `@nonobjc` attribute, which makes a Swift declaration unavailable in Objective-C. You can use it to resolve circularity for bridging methods and to allow overloading of methods for classes imported by Objective-C. If an Objective-C method is overridden by a Swift method that cannot be represented in Objective-C, such as by specifying a parameter to be a variable, that method must be marked `@nonobjc`.

Requiring Dynamic Dispatch

When Swift APIs are imported by the Objective-C runtime, there

are no guarantees of dynamic dispatch for properties, methods, subscripts, or initializers. The Swift compiler may still devirtualize or inline member access to optimize the performance of your code, bypassing the Objective-C runtime.

You can use the `dynamic` modifier to require that access to members be dynamically dispatched through the Objective-C runtime. Requiring dynamic dispatch is rarely necessary. However, it is necessary when using APIs like key-value observing or the `method_exchangeImplementations` function in the Objective-C runtime, which dynamically replace the implementation of a method at runtime. If the Swift compiler inlined the implementation of the method or devirtualized access to it, the new implementation would not be used.

NOTE

Declarations marked with the `dynamic` modifier cannot also be marked with the `@nonobjc` attribute.

Selectors

In Objective-C, a 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, and can be constructed using the `#selector` expression. To create a selector for a method that can be called from Objective-C, pass the name of the method, such as `#selector(MyViewController.tappedButton(sender:))`. To construct a selector for a property's Objective-C getter or setter

method, pass the property name prefixed by the `getter:` or `setter:` label, such as `#selector(getter: MyViewController.myButton)`.

```
1 import UIKit  
2  
3 class MyViewController: UIViewController {  
4  
5     let myButton = UIButton(frame: CGRect(x: 0, y:  
6         0, width: 100, height: 50))  
7  
8     override init?(NibName nibNameOrNil: String?,  
9                     bundle nibBundleOrNil: Bundle?) {  
10         super.init(nibName: nibNameOrNil, bundle:  
11             nibBundleOrNil)
```

```
7     let action =  
8  
9         #selector(MyViewController.tappedButton)  
10        myButton.addTarget(self, action: action,  
11                            forControlEvents: .touchUpInside)  
12  
13    }  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
999  
1000
```

```
required init?(coder: NSCoder) {  
    super.init(coder: coder)  
}  
}
```

NOTE

An Objective-C method reference can be parenthesized, and it can use the `as` operator to disambiguate between overloaded functions, such as
`#selector(((UIView.insert(subview:at:)) as (UIView) -> (UIView, Int) -> Void)).`

Unsafe Invocation of Objective-C Methods

You can invoke an Objective-C method on an Objective-C compatible object using a selector with the `perform(_:)` method or one of its variants. Invoking a method using a selector is inherently unsafe, because the compiler cannot make any guarantees about the result, or even guarantee whether the object responds to the selector. Therefore, using these APIs is strongly discouraged unless your code specifically relies on the dynamic method resolution provided by the Objective-C runtime. For example, using these APIs would be appropriate if you need to implement a class that uses the target-action design pattern in its interface, like `NSResponder` does. In most cases, it's safer and more convenient to cast the object to `AnyObject` and use optional chaining with a method call as described in [id Compatibility](#).

The methods that perform a selector synchronously, such as

`perform(_:)`, return an implicitly unwrapped optional unmanaged pointer to an `AnyObject` instance (`Unmanaged<AnyObject>!`), because the type and ownership of the value returned by performing the selector can't be determined at compile time. In contrast, the methods that perform a selector on a specific thread or after a delay, such as `perform(_:on:with:waitForCompletion:modes:)` and `perform(_:with:afterDelay:)`, don't return a value. See [Unmanaged Objects](#) for more information.

```
1 let string: NSString = "Hello, Cocoa!"  
2  
3 let selector =  
4     #selector(NSString.lowercased(with:))  
5  
6 let locale = Locale.current  
7  
8 if let result = string.perform(selector, with:  
9     locale) {  
10  
11     print(result.takeUnretainedValue())  
12  
13 }  
14  
15 // Prints "hello, cocoa!"
```

Attempting to invoke a method on an object with an unrecognized

selector causes the receiver to call `doesNotRecognizeSelector(_:)`, which by default raises an `NSInvalidArgumentException` exception.

```
1 let array: NSArray = ["delta", "alpha", "zulu"]
2
3 // Not a compile-time error because NSDictionary
4 // has this selector.
5
6 let selector =
7     #selector(NSDictionary.allKeysForObject)
8
9 // Raises an exception because NSArray does not
10 // respond to this selector.
11
12 array.perform(selector)
```

Keys and Key Paths

In Objective-C, a key is a string that identifies a specific property of an object; a key path is a string of dot-separated keys that specifies a sequence of object properties to traverse. Keys and key paths are frequently used for key-value coding (KVC), a mechanism for indirectly accessing an object's attributes and relationships using string identifiers. Keys and key paths are also used for key-value observing (KVO), a mechanism that enables an object to be notified directly when a property of another object changes.

In Swift, you can use the `#keyPath` expression to generate compiler-checked keys and key paths that can be used by KVC methods like `value(forKey:)` and `value(forKeyPath:)` and KVO methods like `addObserver(_:forKeyPath:options:context:)`. The `#keyPath`

expression accepts chained method or property references, such as `#keyPath(Person.bestFriend.name)`.

NOTE

The syntax of a `#keyPath` expression is similar to the syntax of a `#selector` expression, as described in [Selectors](#).

```
1 class Person: NSObject {
2     var name: String
3     var friends: [Person] = []
4     var bestFriend: Person? = nil
```

```
5
6    init(name: String) {
7        self.name = name
8    }
9 }

let gabrielle = Person(name: "Gabrielle")

let jim = Person(name: "Jim")

let yuanyuan = Person(name: "Yuanyuan")

gabrielle.friends = [jim, yuanyuan]
```

```
gabrielle.bestFriend = yuanyuan

#keyPath(Person.name)

// "name"

gabrielle.value(forKey: #keyPath(Person.name))

// "Gabrielle"

#keyPath(Person.bestFriend.name)

// "bestFriend.name"

gabrielle.value(forKeyPath:
    #keyPath(Person.bestFriend.name))
```

```
// "Yuanyuan"

#keyPath(Person.friends.name)

// "friends.name"

gabrielle.value(forKeyPath:
    #keyPath(Person.friends.name))

// ["Yuanyuan", "Jim"]
```

Writing Swift Classes and Protocols with Objective-C Behavior

Interoperability lets you write Swift code that incorporates Objective-C behavior. You can subclass Objective-C classes, declare and adopt Objective-C protocols, and take advantage of other Objective-C functionality when writing Swift code. This means that you can create classes and protocols 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.

```
1 import UIKit  
2 class MySwiftViewController: UIViewController {  
3     // define the class  
4 }
```

A Swift subclass gets all the functionality offered by the superclass in Objective-C.

To provide your own implementations of the superclass's methods, use the `override` modifier. The compiler automatically infers the name of the overridden Objective-C method that matches the Swift method name. You can use the `@objc(name)` attribute to explicitly specify the corresponding Objective-C symbol.

NSCoding

The `NSCoding` protocol requires that conforming types implement the required initializer `init(coder:)` and the required method `encode(with:)`. Classes that adopt `NSCoding` directly must implement this method. Subclasses of classes that adopt `NSCoding` that have one or more custom initializers or any properties without initial values must also implement this method.

For objects that are loaded from Storyboards or archived to disk using the `NSUserDefaults` or `NSKeyedArchiver` classes, you must provide a full implementation of this initializer. However, you might not need to implement an initializer for types that are expected to or cannot be instantiated in this way.

Adopting Protocols

Objective-C protocols are imported as Swift protocols, which can be adopted by a class in a comma-separated list following the name of a class's superclass, if any.

```
1 class MySwiftViewController: UIViewController,  
2     UITableViewDelegate, UITableViewDataSource  
3 {  
4     // define the class  
5 }  
6
```

To declare a type that conforms to a single protocol in Swift code, use the protocol name directly as its type (as compared to `id<SomeProtocol>` in Objective-C). To declare a type that conforms to multiple protocols in Swift code, use a protocol composition, which takes the form `SomeProtocol & AnotherProtocol` (as compared to `id<SomeProtocol, AnotherProtocol>` in Objective-C).

```
1 var textFieldDelegate: UITextFieldDelegate  
2 var tableViewController: UITableViewDataSource &  
    UITableViewDelegate
```

NOTE

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

When a Swift initializer, property, subscript, or method is used to satisfy a requirement of an Objective-C protocol, the compiler automatically infers the name to match the requirement, similar to what is done for overridden methods. You can use the `@objc(name)`

attribute to explicitly specify the corresponding Objective-C symbol.

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. For more information about implementing your own initializers, see Initializers in *The Swift Programming Language (Swift 3)*.

When you want to perform additional clean-up before your class is deallocated, you can implement a deinitializer 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' 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. For more information about implementing your own deinitializers, see Deinitializers in *The Swift Programming Language (Swift 3)*.

Using Swift Class Names with Objective-C APIs

Swift classes are namespaced based on the module they are compiled in, even when used from Objective-C code. Unlike

Objective-C, where all classes are part of a global namespace—and must not have the same name—Swift classes can be disambiguated based on the module they reside in. For example, the fully qualified name of a Swift class named `DataManager` in a framework named `MyFramework` is `MyFramework.DataManager`. A Swift app target is a module itself, so the fully qualified name of a Swift class named `Observer` in an app called `MyGreatApp` is `MyGreatApp.Observer`.

In order to preserve namespacing when a Swift class is used in Objective-C code, Swift classes are exposed to the Objective-C runtime with their fully qualified names. Therefore, when you work with APIs that operate on the string representation of a Swift class, you must include the fully qualified name of the class. For example, when you create a document-based Mac app, you provide the name of your `NSDocument` subclass in your app's `Info.plist` file. In Swift, you must use the full name of your document subclass, including the module name derived from the name of your app or framework.

In the example below, the `NSClassFromString(_:)` function is used to retrieve a reference to a class from its string representation. In order to retrieve a Swift class, the fully qualified name, including the name of the app, is used.

```
let myPersonClass: AnyClass? =  
    NSClassFromString("MyGreatApp.Person")
```

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, you should make the type of the outlet an implicitly unwrapped optional. This way, you can let the storyboard connect the outlets at runtime, after initialization. When your class is initialized from a storyboard or `.xib` file, you can assume that the outlet has been connected.

For example, the following Swift code declares a class with an

outlet, an outlet collection, and an action:

```
1 class MyViewController: UIViewController {  
2     @IBOutlet weak var button: UIButton!  
3     @IBOutlet var textFields: [UITextField]!  
4     @IBAction func buttonTapped(sender: AnyObject)  
5     {  
6         print("button tapped!")  
7     }  
}
```

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 the `UIView` class or the `NSView` class, 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.

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.

```
1 @IBDesignable  
2  
3     @IBInspectable var textColor: UIColor  
4  
5     @IBInspectable var iconHeight: CGFloat  
6  
7     // ...  
8 }
```

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 in *The Swift Programming Language (Swift 3)*.

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 in *The Swift Programming Language (Swift 3)*.

Implementing Core Data Managed Object Subclasses

Core Data provides the underlying storage and implementation of properties in subclasses of the `NSManagedObject` class. Core Data also provides the implementation of instance methods that you use to add and remove objects from to-many relationships. You use the `@NSManaged` attribute to inform the Swift compiler that Core Data provides the storage and implementation of a declaration at runtime.

Add the `@NSManaged` attribute to each property or method declaration in your managed object subclass that corresponds to an attribute or relationship in your Core Data model. For example, consider a Core Data entity called “Person” with a String attribute “name” and a to-many relationship “friends”:

ENTITIES**E Person**

FETCH REQUESTS

CONFIGURATIONS

C Default**▼ Attributes**

Attribute	Type ^
S name	String
+	-

▼ Relationships

Relationship ^	Destination	Inverse
M friends	Person	◊ friends ◊
+	-	

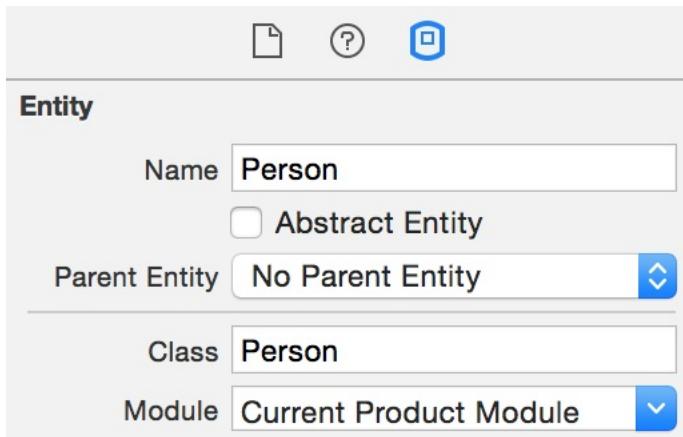
Choose “Create NSManagedObject Subclass...” from the “Editor” menu to generate corresponding Swift code for the NSManagedObject subclass, Person:

```
1 // Person+CoreDataClass.swift  
2  
3 import CoreData  
4  
5 class Person: NSManagedObject {  
6  
7     // Insert code here to add functionality to  
8     // your managed object subclass  
9 }  
10  
11 // Person+CoreDataProperties.swift
```

```
extension Person {  
    @NSManaged var name: String  
  
    @NSManaged var friends: NSSet  
}
```

The `name` and `friends` properties are both declared with the `@NSManaged` attribute to indicate that Core Data provides their implementation and storage at runtime.

To configure a Swift subclass of `NSManagedObject` for use by a Core Data model entity, open the model entity inspector in Xcode, enter the class name into the Class field, and choose “Current Product Module” from the Module field drop-down list.



Declaring Protocols

In Swift, you can define protocols that Objective-C classes can conform to. To create a Swift protocol that can be adopted by an Objective-C class, mark the `protocol` declaration with the `@objc` attribute.

```
1 import UIKit  
2  
3 @objc protocol MyCustomProtocol {  
4  
5     var people: [Person] { get }  
6  
7     func tableView(_ tableView: UITableView,  
8                     configure cell: UITableViewCell, forPerson:  
9                         person: Person)  
10  
11    @objc optional func tableView(_ tableView:  
12                                 UITableView, willDisplay cell:  
13                                     UITableViewCell, forRowAt index: IndexPath)
```

```
8 }
```

```
UITableViewController, forPerson person: Person)
```

A protocol declares all initializers, properties, subscripts, and methods that an Objective-C class must implement in order to conform to the protocol. Any optional protocol requirements must be marked with the `@objc` attribute and have the `optional` modifier.

An Objective-C class can conform a protocol declared in Swift in the same way that it would an Objective-C protocol, by implementing the required methods.

```
1
```

```
@interface MyCustomController: UIViewController  
    <MyCustomProtocol>
```

```
2 @property (nonatomic, strong) NSArray<Person *>
3   *people;
4
5 @implementation MyCustomController
6
7
8 - (void)tableView:(UITableView *)tableView
9   configure:(UITableViewCell *)cell
forPerson:(Person *)person
```

```
{  
    // Configure cell  
}  
  
@end
```

Working with Cocoa Frameworks

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

Swift automatically converts some Objective-C types to Swift types, and some Swift types to Objective-C types. Types that can be converted between Objective-C and Swift are referred to as *bridged* types. For example, in Swift code, you can pass a `String` value to an Objective-C method declared to take an `NSString` parameter. In addition, many of the Cocoa frameworks, including Foundation, AppKit, and UIKit refine their APIs to be more natural in Swift. For example, the `NSCoder` method `decodeObjectOfClass(_:forKey:)` uses Swift generics to provide a stronger type signature.

Foundation

The Foundation framework provides a base layer of functionality for apps and frameworks, including data storage, text processing, dates and times, sorting and filtering, persistence, and networking.

Bridged Types

The Swift Foundation overlay provides the following bridged value types for the following Objective-C reference types:

Objective-C Reference Type

Swift Value Type

NSAffineTransform	AffineTransform
NSArray	Array
NSDateCalendar	Calendar
NSCharacterSet	CharacterSet
NSData	Data
NSDateComponents	DateComponents
NSDateInterval	DateInterval
NSDate	Date
NSDecimalNumber	Decimal
NSDictionary	Dictionary

NSIndexPath	IndexPath
NSMutableIndexSet	IndexSet
NSLocale	Locale
NSMeasurement	Measurement
NSNotification	Notification
NSNumber	Bool, Double, Float, Int, UInt
NSPersonNameComponents	PersonNameComponents
NSSet	Set
NSString	String

NSTimeZone	TimeZone
NSURLComponents	URLComponents
NSURLQueryItem	URLQueryItem
NSURL	URL
NSURLComponents	URLComponents
NSURLQueryItem	URLQueryItem
NSURLRequest	URLRequest
NSUUID	UUID

These value types have the same functionality as their corresponding reference types. Class clusters that include

immutable and mutable subclasses are bridged to a single value type. Swift code uses `var` and `let` to control mutability, so it doesn't need both classes. The corresponding reference types can be accessed with their original `NS` class name prefix.

Anywhere you can use a bridged Objective-C reference type, you can use the Swift value type instead. This lets you take advantage of the functionality available on the reference type's implementation in a way that is natural in Swift code. For this reason, you should almost never need to use a bridged reference type directly in your own code. In fact, when Swift code imports Objective-C APIs, the importer replaces Objective-C reference types with their corresponding value types. Likewise, when Objective-C code imports Swift APIs, the importer also replaces Swift value types with their corresponding Objective-C reference types.

One of the primary advantages of value types over reference types

is that they make it easier to reason about your code. For more information about value types, see Classes and Structures in *The Swift Programming Language (Swift 3)*, and WWDC 2015 session 414 [Building Better Apps with Value Types in Swift](#).

If you do need to use a bridged Foundation object, You can cast between bridged types using the `as` type casting operator.

Renamed Types

The Swift Foundation overlay renames classes and protocols, as well as related enumerations and constants.

Imported Foundation classes and protocols drop their `NS` prefix, with the following exceptions:

- Classes specific to Objective-C or inherently tied to the Objective-C runtime, such as `NSObject`, `NSAutoreleasePool`, `NSEException`, and `NSProxy`
- Platform-specific classes, such as `NSBackgroundActivity`, `NSUserNotification`, and `NSXPConnection`
- Classes that have a value type equivalent, as described in [Bridged Types](#), such as `NSString`, `NSDictionary`, and `NSURL`
- Classes that do not have a value type equivalent but are planned to have one in the near future, such as `NSAttributedString`, `NSRegularExpression`, and `NSPredicate`

Foundation classes often declare enumeration or constant types. When importing these types, Swift moves them to be nested types of their related types. For example, the `NSJSONReadingOptions` option set is imported as `JSONSerialization.ReadingOptions`.

Strings

Swift bridges between the `String` type and the `NSString` class. You can create an `NSString` object by casting a `String` value using the `as` operator. You can also create an `NSString` object using a string literal by explicitly providing a type annotation.

```
1 import Foundation  
2  
3 let string: String = "abc"  
4  
5 let bridgedString: NSString = string as NSString  
6  
7 let stringLiteral: NSString = "123"
```

```
6 if let integerValue = Int(stringLiteral as String)  
7 {  
8     print("\(stringLiteral) is the integer \  
9         (integerValue)")  
10 }  
11 // Prints "123 is the integer 123"
```

NOTE

The Swift `String` type is composed of encoding-independent Unicode characters, and provides support for accessing those characters in various Unicode representations. The `NSString` class encodes a Unicode-compliant text string, represented as a sequence of UTF-16 code units. `NSString` methods that express length, character indexes, or ranges in terms of 16-bit platform-endian values have corresponding Swift `String` methods that use `String.Index` and `Range<String.Index>` values rather than `Int` and `NSRange` values.

Numbers

Swift bridges between the `NSNumber` class and Swift arithmetic types, including `Int`, `Double`, and `Bool`.

You can create an `NSNumber` object by casting a Swift number value using the `as` operator. Because `NSNumber` can contain a variety of different types, you must use the `as?` operator when casting to a Swift number type. You can also create an `NSNumber` object using a floating-point, integer, or Boolean literal by explicitly providing a type annotation.

```
1 import Foundation  
2  
3 let number = 42  
4  
5 let integerLiteral: NSNumber = 5  
6  
7 let floatLiteral: NSNumber = 3.14159  
8  
9 let booleanLiteral: NSNumber = true
```

NOTE

Objective-C platform-adaptive integer types, such as `NSUInteger` and `NSInteger`, are bridged to `Int`.

Arrays

Swift bridges between the `Array` type and the `NSArray` class. When you bridge from an `NSArray` object with a parameterized type to a Swift array, the element type of the resulting array is bridged as well. If an `NSArray` object does not specify a parameterized type, it is bridged to a Swift array of type `[Any]`.

For example, consider the following Objective-C declarations:

```
1 @property NSArray *objects;  
2  
3     - (NSArray<NSDate *> *)datesBeforeDate:(NSDate  
4                                         *)date;  
  
      - (void)addDatesParsedFromTimestamps:  
                                         (NSArray<NSString *> *)timestamps;
```

Here's how Swift imports them:

```
1 var objects: [Any]  
2  
3 var dates: [Date]  
4  
5 func datesBeforeDate(date: Date) -> [Date]  
6  
7 func addDatesParsedFromTimestamps(timestamps:  
8     [String])
```

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.

```
1 let schoolSupplies: NSArray = ["Pencil", "Eraser",  
2                                "Notebook"]  
  
// schoolSupplies is an NSArray object containing  
// three values
```

Sets

In addition to arrays, Swift bridges between the `Set` type and the `NSSet` class. When you bridge from an `NSSet` object with a parameterized type to a Swift set, the resulting set is of type `Set<ObjectType>`. If an `NSSet` object does not specify a parameterized type, it is bridged to a Swift set of type

Set<AnyHashable>.

For example, consider the following Objective-C declarations:

```
1 @property NSSet *objects;  
2 @property NSSet<NSString *> *words;  
3 - (NSSet<NSString *> *)wordsMatchingPredicate:  
    (NSPredicate *)predicate;  
4 - (void)removeWords:(NSSet<NSString *> *)words;
```

Here's how Swift imports them:

```
1 var objects: Set<AnyHashable>  
2  
3 var words: Set<String>  
4  
3 func wordsMatchingPredicate(predicate: NSPredicate)  
    -> Set<String>  
4  
4 func removeWords(words: Set<String>)
```

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

```
1 let amenities: NSSet = ["Sauna", "Steam Room",  
2 "Jacuzzi"]  
  
// amenities is an NSSet object containing three  
values
```

Dictionaries

Swift also bridges between the `Dictionary` type and the `NSDictionary` class. When you bridge from an `NSDictionary` object with parameterized types to a Swift dictionary, the resulting dictionary is of type `[Key: Value]`. If an `NSDictionary` object does not specify parameterized types, it is bridged to a Swift dictionary.

of type [AnyHashable: Any].

For example, consider the following Objective-C declarations:

```
1 @property NSDictionary *keyedObjects;  
2 @property NSDictionary<NSURL *, NSData *>  
    *cachedData;  
3 - (NSDictionary<NSURL *, NSNumber *>  
    *)fileSizesForURLsWithSuffix:(NSString  
    *)suffix;  
4 - (void)setCacheExpirations:(NSDictionary<NSURL *,  
    NSDate *> *)expirations;
```

Here's how Swift imports them:

```
1 var keyedObjects: [AnyHashable: Any]  
2 var cachedData: [URL: Data]  
3 func fileSizesForURLsWithSuffix(suffix: String) ->  
    [URL: NSNumber]  
4 func setCacheExpirations(expirations: [URL:  
    NSDate])
```

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

NSDictionary object instead of a Swift dictionary.

```
1 let medalRankings: NSDictionary = ["Gold": "1st  
Place", "Silver": "2nd Place", "Bronze":  
"3rd Place"]  
  
2 // medalRankings is an NSDictionary object  
// containing three key-value pairs
```

Core Foundation

Core Foundation types are imported as 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, you can annotate them with either the `CF_RETURNS_RETAINED` or `CF_RETURNS_NOT_RETAINED` macro to automatically insert memory management calls. You can also use the `CF_IMPLICIT_BRIDGING_ENABLED` and

`CF_IMPLICIT_BRIDGING_DISABLED` macros to enclose C function declarations that follow Core Foundation ownership policy naming policy in order to infer memory management from naming.

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<Instance>` structure. All indirectly returned

Core Foundation objects are unmanaged as well. For example, here's an unannotated C function:

```
CFStringRef StringByAddingTwoStrings(CFStringRef  
    s1, CFStringRef s2)
```

And here's how Swift imports it:

```
1 func StringByAddingTwoStrings(_: CFString!, _:  
    CFString!) -> Unmanaged<CFString>! {  
2     // ...  
3 }
```

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<Instance>` 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.

```
1 let memoryManagedResult =  
    StringByAddingTwoStrings(str1,  
        str2).takeUnretainedValue()  
  
2 // 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.

For more information, see *Memory Management Programming Guide for Core Foundation* in *Memory Management Programming Guide for Core Foundation*.

Unified Logging

The unified logging system provides an API for capturing messaging across all levels of the system, and is a replacement for the `NSLog` function in the Foundation framework. Unified logging is available in iOS 10.0 and later, macOS 10.12 and later, tvOS 10.0 and later, and watchOS 3.0 and later.

In Swift, you can interact with the unified logging system using the top level `os_log(_:_:log:type:_)` function, found in the `log` submodule of the `os` module.

```
1 import os.log  
2  
3 os_log("This is a log message.")
```

You can format a log message using an `NSString` or `printf` format string along with one or more trailing arguments.

```
1 let fileSize = 1234567890  
2 os_log("Finished downloading file. Size: %{iec-  
        bytes}d", fileSize)
```

You can also specify a log level defined by the logging system, such as Info, Debug, or Error, in order to control how log messages are handled according to the importance of the logging event. For example, information that may be helpful, but isn't essential for troubleshooting errors should be logged at the Info level.

```
os_log("This is additional info that may be helpful  
for troubleshooting.", type: .info)
```

To log a message to a specific subsystem, you can create a new `OSLog` object, specifying the subsystem and category, and pass it as a parameter to the `os_log` function.

```
1 let customLog =  
  
    OSLog("com.your_company.your_subsystem_name.¶  
          "your_category_name")  
  
2 os_log("This is info that may be helpful during  
development or debugging.", log: customLog,  
type: .debug)
```

For more information about the unified logging system, see [Logging](#).

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. Just as in Objective-C, before you send a message that a delegate may not respond to, you ask the delegate whether it responds to the selector. In Swift, you can use optional chaining to invoke an optional protocol method on a possibly `nil` object and unwrap the possible result using `if-let` syntax. 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.

```
1 class MyDelegate: NSObject, NSWindowDelegate {  
2  
3     func window(_ window: NSWindow,  
4                  willUseFullScreenContentSize proposedSize:  
5                  NSSize) -> NSSize {  
6  
7         return proposedSize  
8     }  
9 }  
10  
11 myWindow.delegate = MyDelegate()  
12  
13 if let fullScreenSize =  
14     myWindow.delegate?.window(myWindow,  
15
```

```
willUseFullScreenContentSize: mySize) {  
8    print(NSStringFromSize(fullScreenSize))  
9}
```

Lazy Initialization

A *lazy property* is a property whose underlying value is only initialized when the property is first accessed. Lazy properties are useful when the initial value for a property either requires complex or computationally expensive setup, or cannot be determined until after an instance's initialization is complete.

In Objective-C, a property may override its synthesized getter

method such that the underlying instance variable is conditionally initialized if its value is `nil`:

```
1 @property NSXMLDocument *_XML;  
2  
3 - (NSXMLDocument *)XML {  
4  
5     if (_XML == nil) {  
6  
    _XML = [[[NSXMLDocument alloc]  
8     initWithContentsOfURL:[[NSBundle mainBundle]  
9     URLForResource:@"/path/to/resource"  
10    withExtension:@"xml"] options:0 error:nil];  
11 }  
12 }
```

```
7  
8     return _XML;  
9 }
```

In Swift, a stored property with an initial value can be declared with the `lazy` modifier to have the expression calculating the initial value only evaluated when the property is first accessed:

```
lazy var XML: XMLDocument = try!  
    XMLDocument(contentsOf:  
        Bundle.main.url(forResource: "document",  
            withExtension: "xml")!, options: 0)
```

Because a lazy property is only computed when accessed for a fully-initialized instance it may access constant or variable properties in its default value initialization expression:

```
1  var pattern: String  
2  lazy var regex: NSRegularExpression = try!  
    NSRegularExpression(pattern: self.pattern,  
options: [])
```

For values that require additional setup beyond initialization, you can assign the default value of the property to a self-evaluating closure that returns a fully-initialized value:

```
1 lazy var currencyFormatter: NumberFormatter = {  
2     let formatter = NumberFormatter()  
3     formatter.numberStyle = .currency  
4     formatter.currencySymbol = "₹"  
5     return formatter  
6 }
```

NOTE

If a lazy property has not yet been initialized and is accessed by more than one thread at the same time, there is no guarantee that the property will be initialized only once.

For more information, see Lazy Stored Properties in *The Swift Programming Language (Swift 3)*.

Error Handling

In Cocoa, methods that produce errors take an `NSError` pointer parameter as their last parameter, which populates its argument with an `NSError` object if an error occurs. Swift automatically translates Objective-C methods that produce errors into methods that throw an error according to Swift's native error handling functionality.

NOTE

Methods that *consume* errors, such as delegate methods or methods that take a completion handler with an `NSError` object argument, do not become methods that throw when imported by Swift.

For example, consider the following Objective-C method from `NSFileManager`:

```
1 - (BOOL)removeItemAtURL:(NSURL *)URL  
2           error:(NSError **)error;
```

In Swift, it's imported like this:

```
func removeItem(at: URL) throws
```

Notice that the `removeItem(at:)` method is imported by Swift with a `Void` return type, no `error` parameter, and a `throws` declaration.

If the last non-block parameter of an Objective-C method is of type `NSError **`, Swift replaces it with the `throws` keyword, to indicate that the method can throw an error. If the Objective-C method's error parameter is also its first parameter, Swift attempts to simplify the method name further, by removing the “WithError” or “AndReturnError” suffix, if present, from the first part of the selector. If another method is declared with the resulting selector, the method name is not changed.

If an error producing Objective-C method returns a `BOOL` value to indicate the success or failure of a method call, Swift changes the return type of the function to `Void`. Similarly, if an error producing

Objective-C method returns a `nil` value to indicate the failure of a method call, Swift changes the return type of the function to a nonoptional type.

Otherwise, if no convention can be inferred, the method is left intact.

NOTE

Use the `NS_SWIFT_NO_THROW` macro on an Objective-C method declaration that produces an `NSError` to prevent it from being imported by Swift as a method that throws.

Catching and Handling an Error

In Objective-C, error handling is opt-in, meaning that errors produced by calling a method are ignored unless an error pointer is provided. In Swift, calling a method that throws requires explicit error handling.

Here's an example of how to handle an error when calling a method in Objective-C:

```
4 NSError *error = nil;
5 BOOL success = [fileManager moveItemAtURL:fromURL
6                           toURL:toURL error:&error];
7
8 }
```

And here's the equivalent code in Swift:

```
1 let fileManager = FileManager.default
2
3 let fromURL = URL(fileURLWithPath: "/path/to/old")
4
5 let toURL = URL(fileURLWithPath: "/path/to/new")
6
7 do {
8
9     try fileManager.moveItem(at: fromURL, to:
10
11         toURL)
12
13 } catch let error as NSError {
14
15     print("Error: \(error.domain)")
16
17 }
```

Additionally, you can use `catch` clauses to match on particular error

codes as a convenient way to differentiate possible failure conditions:

```
1 do {
2     try fileManager.moveItem(at: fromURL, to:
3         toURL)
4
5 } catch CocoaError.fileNoSuchFile {
6     print("Error: no such file exists")
7 } catch CocoaError.readFileUnsupportedScheme {
8     print("Error: unsupported scheme (should be
9         'file://')")
10 }
```

Converting Errors to Optional Values

In Objective-C, you pass `NULL` for the error parameter when you only care whether there was an error, not what specific error occurred. In Swift, you write `try?` to change a throwing expression into one that returns an optional value, and then check whether the value is `nil`.

For example, the `NSFileManager` instance method `URL(for:in:appropriateForURL:create:)` returns a URL in the specified search path and domain, or produces an error if an appropriate URL does not exist and cannot be created. In Objective-C, the success or failure of the method can be determined by whether an `NSURL` object is returned.

1

```
NSFileManager *fileManager = [NSFileManager
```

```
    defaultManager];  
2  
3 NSURL *tmpURL = [fileManager  
URLForDirectory:NSCachesDirectory  
4 inDomain:NSUserDomainMask  
5 appropriateForURL:nil  
6 create:YES  
7 error:nil];  
8
```

```
9     if (tmpURL != nil) {  
10        // ...  
11    }  
12}
```

You can do the same in Swift as follows:

```
1 let fileManager = FileManager.default
2 if let tmpURL = try? fileManager.url(for:
3     .cachesDirectory, in: .userDomainMask,
4     appropriateFor: nil, create: true) {
5     // ...
6 }
```

Throwing an Error

If an error occurs in an Objective-C method, that error is used to populate the error pointer argument of that method:

```
1 // an error occurred  
2 if (errorPtr) {  
3     *errorPtr = [NSError  
4         errorWithDomain:NSUTFLErrorDomain  
5         code:NSUTFLErrorCannotOpenFile  
6         userInfo:nil];  
7 }
```

If an error occurs in a Swift method, the error is thrown, and automatically propagated to the caller:

```
1 // an error occurred  
2 throw NSError(domain: NSURLErrorDomain, code:  
                  NSURLErrorCannotOpenFile, userInfo: nil)
```

If Objective-C code calls a Swift method that throws an error, the error is automatically propagated to the error pointer argument of the bridged Objective-C method.

For example, consider the `read(from ofType:)` method in `NSDocument`. In Objective-C, this method's last parameter is of type `NSError **`. When overriding this method in a Swift subclass of `NSDocument`, the method replaces its error parameter and throws instead.

```
1 class SerializedDocument: NSDocument {
```

```
2     static let ErrorDomain =
3         "com.example.error.serialized-document"
4
5
6     var representedObject: [String: Any] = [:]
7
8     override func read(from fileWrapper:
9                         FileWrapper, ofType typeName: String)
10    throws {
11        guard let data =
12            fileWrapper.regularFileContents else {
13                throw NSError(domain: ErrorDomain,
14                              code: -1,
15                              userInfo: nil)
16            }
17
18        let decoder = JSONDecoder()
19        let object = try decoder.decode([String: Any].self,
20                                         from: data)
21
22        self.representedObject = object
23    }
24
```



```
        throw NSError(domain:  
                      SerializedDocument.ErrorDomain, code: -1,  
                      userInfo: nil)  
  
    }  
  
}  
  
}
```

If the method is unable to create an object with the regular file contents of the document, it throws an `NSError` object. If the method is called from Swift code, the error is propagated to its calling scope. If the method is called from Objective-C code, the error instead populates the error pointer argument.

In Objective-C, error handling is opt-in, meaning that errors produced by calling a method are ignored unless you provide an error pointer. In Swift, calling a method that throws requires explicit error handling.

NOTE

Although Swift error handling resembles exception handling in Objective-C, it is entirely separate functionality. If an Objective-C method throws an exception during runtime, Swift triggers a runtime error. There is no way to recover from Objective-C exceptions directly in Swift. Any exception handling behavior must be implemented in Objective-C code used by Swift.

Key-Value Observing

Key-value observing is a mechanism that allows objects to be notified of changes to specified properties of other objects. You can use key-value observing with a Swift class, as long as the class inherits from the `NSObject` class. You can use these three steps to implement key-value observing in Swift.

1. Add the `dynamic` modifier to any property you want to observe. For more information on `dynamic`, see [Requiring Dynamic Dispatch](#).

```
1 class MyObjectToObserve: NSObject {  
2     dynamic var myDate = NSDate()  
3     func updateDate() {  
4         myDate = NSDate()  
5     }  
6 }
```

2. Create a global context variable.

```
private var myContext = 0
```

3. Add an observer for the key-path, override the

`observeValue(for:of:change:context:)` method, and remove the observer in `deinit`.

```
1  class MyObserver: NSObject {
2
3      var objectToObserve = MyObjectToObserve()
4
5      override init() {
6          super.init()
7
8          objectToObserve.addObserver(self,
9              forKeyPath: "myDate", options: .new,
10             context: &myContext)
11
12     }
13 }
```

```
8 override func observeValue(forKeyPath
 9   keyPath: String?, of object: Any?,
10   change: [NSKeyValueChangeKey : Any]?,
11   context: UnsafeMutableRawPointer?) {
12
13   if context == &myContext {
14     if let newValue = change? [.newKey] {
15       print("Date changed: \(newValue)")
16     }
17   } else {
18     super.observeValue(forKeyPath:
```

```
    keyPath, of: object, change: change,
    context: context)

15        }

16    }

17

18    deinit {
19
        objectToObserve.removeObserver(self,
            forKeyPath: "myDate", context:
            &myContext)
20    }
```

Undo

In Cocoa, you register operations with `NSUndoManager` to allow users to reverse that operation's effect. You can take advantage of Cocoa's undo architecture in Swift just as you would in Objective-C.

Objects in an app's responder chain—that is, subclasses of `NSResponder` on macOS and `UIResponder` on iOS—have a read-only `undoManager` property that returns an optional `NSUndoManager` value, which manages the undo stack for the app. Whenever an action is taken by the user, such as editing the text in a control or deleting an

item at a selected row, an undo operation can be registered with the undo manager to allow the user to reverse the effect of that operation. An undo operation records the steps necessary to counteract its corresponding operation, such as setting the text of a control back to its original value or adding a deleted item back into a table.

NSUndoManager supports two ways to register undo operations: a “simple undo”, which performs a selector with a single object argument, and an “invocation-based undo”, which uses an NSInvocation object that takes any number and any type of arguments.

For example, consider a simple Task model, which is used by a ToDoListController to display a list of tasks to complete:

```
1 | class Task {
```

```
2     var text: String  
3  
4  
5     init(text: String) {  
6         self.text = text  
7     }  
8 }  
  
9  
class ToDoListController: NSViewController,  
    NSTableViewDataSource, NSTableViewDelegate
```

```
{  
  
    @IBOutlet var tableView: NSTableView!  
  
    var tasks: [Task] = []  
  
    // ...  
  
}
```

For methods that take more than one argument, you can create an undo operation using an `NSInvocation`, which invokes the method with arguments that effectively revert the app to its previous state:

```
1 @IBOutlet var remainingLabel: NSTextView!
```

```
2
3 func mark(task: Task, asCompleted completed: Bool)
4 {
5     if let target =
6         undoManager?.prepare(withInvocationTarget:
7             self) as? ToDoListController {
8
9         target.mark(task: task, asCompleted:
10            !completed)
11
12     undoManager?.setActionName(NSLocalizedString(
13         comment: "Mark As Completed"))
14 }
```

```
7
}
8
9    task.completed = completed
tableView.reloadData()
let numberRemaining = tasks.filter{ $0.completed
}.count
remainingLabel.string = String(format:
NSLocalizedString("todo.task.remaining",
comment: "Tasks Remaining: %d"),
numberRemaining)
```

```
    numberRemaining)  
}  
}
```

The `prepare(withInvocationTarget:)` method returns a proxy to the specified target. By casting to `ToDoListController`, this return value can make the corresponding call to `mark(task:asCompleted:)` directly.

For more information, see *Undo Architecture*.

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 [Selectors](#).

Singleton

Singletons provide a globally accessible, shared instance of an object. You can create your own singletons as a way to provide a unified access point to a resource or service that's shared across an app, such as an audio channel to play sound effects or a network manager to make HTTP requests.

In Objective-C, you can ensure that only one instance of a singleton object is created by wrapping its initialization in a call the

`dispatch_once` function, which executes a block once and only once for the lifetime of an app:

```
1 + (instancetype)sharedInstance {  
2  
2     static id _sharedInstance = nil;  
3  
3     static dispatch_once_t onceToken;  
4  
4     dispatch_once(&onceToken, ^{  
5  
5         _sharedInstance = [[self alloc] init];  
6  
6     });  
7  
7  
8     return _sharedInstance;  
9 }
```

In Swift, you can simply use a static type property, which is

guaranteed to be lazily initialized only once, even when accessed across multiple threads simultaneously:

```
1 class Singleton {  
2     static let sharedInstance = Singleton()  
3 }
```

If you need to perform additional setup beyond initialization, you can assign the result of the invocation of a closure to the global constant:

```
1 class Singleton {  
2     static let sharedInstance: Singleton = {  
3         let instance = Singleton()  
4         // setup code  
5         return instance  
6     }()  
7 }
```

For more information, see Type Properties in *The Swift Programming Language (Swift 3)*.

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.

```
1 if object is UIButton {  
2     // object is of type UIButton  
3 } else {  
4     // object is not of type UIButton  
5 }
```

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.

```
1 if let button = object as? UIButton {  
2     // object is successfully cast to type UIButton  
3     // and bound to button  
4 } else {  
5     // object could not be cast to type UIButton  
6 }
```

For more information, see Type Casting in *The Swift Programming Language (Swift 3)*.

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:

```
1 if let dataSource = object as?  
2     UITableViewDataSource {  
3         // object conforms to UITableViewDataSource and  
4         // is bound to dataSource  
5     } else {  
6         // object not conform to UITableViewDataSource  
7     }  
8 }
```

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 in *The Swift Programming Language (Swift 3)*.

Serializing

Serialization allows you to encode and decode objects in your application to and from architecture-independent representations, such as JSON or property lists. These representations can then be written to a file or transmitted to another process locally or over a network.

In Objective-C, you can use the Foundation framework classes `NSJSONSerialization` and `NSPropertyListSerialization` to initialize objects from a decoded JSON or property list serialization value—usually an object of type `NSDictionary<NSString *, id>`. You can

do the same in Swift, but because Swift enforces type safety, additional type casting is required in order to extract and assign values.

For example, consider the following `Venue` structure, which contains a `name` property of type `String`, a `coordinate` property of type `CLLocationCoordinate2D`, and a `category` property of a nested `Category` enumeration type:

```
1 import Foundation  
2  
3 import CoreLocation  
4  
4 struct Venue {  
5     enum Category: String {
```

```
6    case entertainment  
7    case food  
8    case nightlife  
9    case shopping
```

```
}
```

```
var name: String
```

```
var coordinates: CLLocationCoordinate2D
```

```
var category: Category
```

```
}
```

An app that interacts with `Venue` instances may communicate with a web server that vends JSON representations of venues, such as:

```
{  
  "name": "Caffè Macs",  
  "coordinates": {  
    "lat": 37.330576,  
    "lng": -122.029739  
  },  
  "category": "food"  
}
```

You can provide a failable `Venue` initializer that takes an `attributes` parameter of type `[String: Any]` corresponding to the value returned from `NSJSONSerialiation` or `NSPropertyListSerialization`:

```
1  init?(attributes: [String: Any]) {  
2  
2      guard let name = attributes["name"] as? String,  
3  
3          let coordinates = attributes["coordinates"]  
4  
4          as? [String: Double],  
5  
5          let latitude = coordinates["lat"],  
6  
6          let longitude = coordinates["lng"],  
6  
6          let category = Category(rawValue:
```


}

A guard statement consisting of multiple optional binding expressions ensures that the `attributes` argument provides all of the required information in the expected format. If any one of the optional binding expressions fails to assign a value to a constant, the guard statement immediately stops evaluating its condition and executes its `else` branch, which returns `nil`.

You can create a `Venue` from a JSON representation by creating a dictionary of attributes using the `NSJSONSerialization` class and then passing that into the corresponding `Venue` initializer:

```
1  let JSON = "{\"name\": \"Caffè  
Macs\", \"coordinates\": {\"lat\":
```

```
37.330576, "lng":  
-122.029739}, "category": "food"}"  
  
2 let data = JSON.data(using: String.Encoding.utf8)!  
  
3 let attributes = try!  
    JSONSerialization.jsonObject(with: data,  
options: []) as! [String: Any]  
  
4  
  
5 let venue = Venue(attributes: attributes)!  
  
6 print(venue.name)  
  
7 // Prints "Caffè Macs"
```

Validating Serialized Representations

In the previous example, the `Venue` initializer optionally returns an instance only if all of the required information is provided. If not, the initializer simply returns `nil`.

It is often useful to determine and communicate the specific reason why a given collection of values failed to produce a valid instance. To do this, you can refactor the failable initializer into an initializer that throws:

```
1 enum ValidationError: Error {  
2     case missing(String)
```

```
3     case invalid(String)
4 }
5
6 init(attributes: [String: Any]) throws {
7     guard let name = attributes["name"] as? String
8     else {
9
10        throw ValidationError.missing("name")
11    }
12
13    guard let coordinates = attributes["coordinates"]
14        as? [Double] else { return nil }
15
16    let location = Location(name: name!, coordinates: coordinates)
17
18    return location
19}
```

```
    as? [String: Double] else {
        throw ValidationError.missing("coordinates")
    }

guard let latitude = coordinates["lat"],
      let longitude = coordinates["lng"]
else {
    throw
    ValidationError.invalid("coordinates")
}
```

```
guard let categoryName = attributes["category"] as?  
    String else {  
  
    throw ValidationError.missing("category")  
  
}
```

```
guard let category = Category(rawValue:  
    categoryName) else {  
  
    throw ValidationError.invalid("category")  
  
}
```

```
        self.name = name  
  
        self.coordinates = CLLocationCoordinate2D(latitude:  
            latitude, longitude: longitude)  
  
        self.category = category  
  
    }  
}
```

Instead of capturing all of the `attributes` values at once in a single `guard` statement, this initializer checks each value individually and throws an error if any particular value is either missing or invalid.

For instance, if the provided JSON didn't have a value for the key `"name"`, the initializer would throw the enumeration value

ValidationError.missing with an associated value corresponding to the "name" field:

```
{  
  "coordinates": {  
    "lat": 37.7842,  
    "lng": -122.4016  
  },  
  "category": "Convention Center"  
}
```

1 | let JSON = "{\"coordinates\": {\"lat\": 37.7842,

```
        \\"lng\\": -122.4016}, \\"category\\":  
        \\\"Convention Center\\\"}  
  
2 let data = JSON.data(using: String.Encoding.utf8)!  
  
3 let attributes = try!  
    JSONSerialization.jsonObject(with: data,  
options: []) as! [String: Any]  
  
4  
  
5 do {  
    let venue = try Venue(attributes: attributes)  
    ...  
7 } catch ValidationError.missing(let field) {  
    ...  
}
```

```
8     print("Missing Field: \\" + field + "\")  
9 }  
  
// Prints "Missing Field: name"
```

Or, if the provided JSON specified all of the required fields but had a value for the "category" key that didn't correspond with the rawValue of any of the defined Category cases, the initializer would throw the enumeration value `ValidationError.invalid` with an associated value corresponding to the "category" field:

```
{  
  "name": "Moscone West",  
  "coordinates": {  
    "lat": 37.7842,  
    "lng": -122.4016  
  },  
  "category": "Convention Center"  
}
```

```
1 let JSON = "{\"name\": \"Moscone West\",  
  \"coordinates\": {\"lat\": 37.7842,
```

```
        \\"lng\\": -122.4016}, \\"category\\":  
        \\\"Convention Center\\\"}  
  
2 let data = JSON.data(using: String.Encoding.utf8)!  
  
3 let attributes = try!  
    JSONSerialization.jsonObject(with: data,  
    options: []) as! [String: Any]  
  
4  
  
5 do {  
    let venue = try Venue(attributes: attributes)  
    ...  
7 } catch ValidationError.invalid(let field) {  
    ...  
}
```

```
8     print("Invalid Field: \(field)")  
9 }  
  
// Prints "Invalid Field: category"
```

Localization

In Objective-C, you typically use the `NSLocalizedString` family of macros to localize strings. These include `NSLocalizedString`, `NSLocalizedStringFromTable`, `NSLocalizedStringFromTableInBundle`, and `NSLocalizedStringWithDefaultValue`. In Swift, the functionality of these macros is made available through a single function: `NSLocalizedString(_:tableName:bundle:value:comment:)`.

Rather than defining separate functions that correspond to each Objective-C macro, the Swift

`NSLocalizedString(_:tableName:bundle:value:)` function specifies default values for the `tableName`, `bundle`, and `value` arguments, so that they may be overridden as necessary.

For example, the most common form of a localized string in an app may only need a localization key and a comment:

```
1 let format = NSLocalizedString("Hello, %@!",  
                                comment: "Hello, {given name}!")  
  
2 let name = "Mei"  
  
3 let greeting = String(format: format, arguments:  
                        [name as CVarArg])  
  
4 print(greeting)  
  
5 // Prints "Hello, Mei!"
```

Or, an app may require more complex usage in order to use localization resources from a separate bundle:

```
1 if let path = Bundle.main.path(forResource:  
2     "Localization", ofType: "strings",  
3     inDirectory: nil, forLocalization: "ja"),  
4  
5     let bundle = Bundle(path: path) {  
6         let translation = NSLocalizedString("Hello",  
7             bundle: bundle, comment: "")  
8         print(translation)  
9     }  
10    // Prints "こんにちは"
```

For more information, see *Internationalization and Localization*

Autorelease Pools

Autorelease pool blocks allow objects to relinquish ownership without being deallocated immediately. Typically, you don't need to create your own autorelease pool blocks, but there are some situations in which either you must—such as when spawning a secondary thread—or it is beneficial to do so—such as when writing a loop that creates many temporary objects.

In Objective-C, autorelease pool blocks are marked using `@autoreleasepool`. In Swift, you can use the `autoreleasepool(_ :)` function to execute a closure within an autorelease pool block.

```
1 import Foundation  
2  
3 autoreleasepool {  
4     // code that creates autoreleased objects.  
5 }
```

For more information, see *Advanced Memory Management Programming Guide*.

API Availability

Some classes and methods are not available to all versions of all

platforms that your app targets. To ensure that your app can accommodate any differences in functionality, you check the availability those APIs.

In Objective-C, you use the `respondsToSelector:` and `instancesRespondToSelector:` methods to check for the availability of a class or instance method. Without a check, the method call throws an `NSInvalidArgumentException` “unrecognized selector sent to instance” exception. For example, the `requestWhenInUseAuthorization` method is only available to instances of `CLLocationManager` starting in iOS 8.0 and macOS 10.10:

```
1 if ([CLLocationManager
      instancesRespondToSelector:@selector(requestW
      {
2 // Method is available for use.
3 } else {
4 // Method is not available.
5 }
```

In Swift, attempting to call a method that is not supported on all targeted platform versions causes a compile-time error.

Here's the previous example, in Swift:

```
1 let locationManager = CLLocationManager()  
2  
2 locationManager.requestWhenInUseAuthorization()  
3 // error: only available on iOS 8.0 or newer
```

If the app targets a version of iOS prior to 8.0 or macOS prior to 10.10, `requestWhenInUseAuthorization()` is unavailable, so the compiler reports an error.

Swift code can use the availability of APIs as a condition **at runtime**. Availability checks can be used in place of a condition in a control flow statement, such as an `if`, `guard`, or `while` statement.

Taking the previous example, you can check availability in an `if` statement to call `requestWhenInUseAuthorization()` only if the method is available at runtime:

```
1 let locationManager = CLLocationManager()  
2  
3 if #available(iOS 8.0, macOS 10.10, *) {  
4     locationManager.requestWhenInUseAuthorization()
```

Alternatively, you can check availability in a `guard` statement, which exits out of scope unless the current target satisfies the specified requirements. This approach simplifies the logic of handling different platform capabilities.

```
1 let locationManager = CLLocationManager()  
2  
3 guard #available(iOS 8.0, macOS 10.10, *) else {  
4     return }  
5  
6 locationManager.requestWhenInUseAuthorization()
```

Each platform argument consists of one of platform names listed below, followed by corresponding version number. The last argument is an asterisk (*), which is used to handle potential future platforms.

Platform Names:

- iOS
- iOSApplicationExtension
- macOS

- macOSApplicationExtension
- watchOS
- watchOSApplicationExtension
- tvOS
- tvOSApplicationExtension

All of the Cocoa APIs provide availability information, so you can be confident the code you write works as expected on any of the platforms your app targets.

You can denote the availability of your own APIs by annotating declarations with the `@available` attribute. The `@available` attribute uses the same syntax as the `#available` runtime check, with the platform version requirements provided as comma-delimited arguments.

For example:

```
1 @available(iOS 8.0, macOS 10.10, *)  
2 func useShinyNewFeature() {  
3     // ...  
4 }
```

NOTE

A method annotated with the `@available` attribute can safely use APIs available to the specified platform requirements without the use of an explicit availability check.

Processing Command-Line Arguments

On macOS, you typically open an app by clicking its icon in the Dock or Launchpad, or by double-clicking its icon from the Finder. However, you can also open an app programmatically and pass command-line arguments from Terminal.

You can get a list of any command-line arguments that are specified at launch by accessing the `Process.arguments` type property. This is equivalent to accessing the `arguments` property on `NSProcessInfo.processInfo()`.

```
$ /path/to/app --argumentName value
```

```
1  for argument in Process.arguments {  
2      print(argument)  
3  }  
4  // prints "/path/to/app"  
5  // prints "--argumentName"  
6  // prints "value"
```

NOTE

The first element in `Process.arguments` is always a path to the executable. Any command-line arguments that are specified at launch begin at `Process.arguments[1]`.

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
<code>bool</code>	<code>CBool</code>
<code>char</code> , <code>signed char</code>	<code>CChar</code>
<code>unsigned char</code>	<code>CUnsignedChar</code>
<code>short</code>	<code>CShort</code>
<code>unsigned short</code>	<code>CUnsignedShort</code>
<code>int</code>	<code>CInt</code>
<code>unsigned int</code>	<code>CUnsignedInt</code>

long	CLong
unsigned long	CUntsignedLong
long long	CLongLong
unsigned long long	CUntsignedLongLong
wchar_t	CWideChar
char16_t	CChar16
char32_t	CChar32
float	CFloat
double	CDouble

Global Constants

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

Imported Constant Enumerations and Structures

In Objective-C, constants are often used to provide a list of possible values for properties and method parameters. You can annotate an Objective-C `typedef` declaration with the `NS_STRING_ENUM` or `NS_EXTENSIBLE_STRING_ENUM` macro to have constants of that type imported by Swift as members of a common type.

Constants that represent a fixed set of possible values can be

imported as an enumeration by adding the `NS_STRING_ENUM` macro. For example, consider the following Objective-C declarations for string constants of type `TrafficLightColor`:

```
1  typedef NSString * TrafficLightColor  
          NS_STRING_ENUM;  
  
2  
  
3  TrafficLightColor const TrafficLightColorRed;  
  
4  TrafficLightColor const TrafficLightColorYellow;  
  
5  TrafficLightColor const TrafficLightColorGreen;
```

Here's how Swift imports them:

```
1 enum TrafficLightColor : String {  
2     case red  
3     case yellow  
4     case green  
5 }
```

Constants that represent an extensible set of possible values can be imported as a structure by adding the `NS_EXTENSIBLE_STRING_ENUM` macro. For example, consider the following Objective-C declarations for string constants of type `StateOfMatter`:

```
1 typedef NSString * StateOfMatter  
2  
3 NS_EXTENSIBLE_STRING_ENUM;  
4  
5 StateOfMatter const StateOfMatterSolid;  
6  
7 StateOfMatter const StateOfMatterLiquid;  
8  
9 StateOfMatter const StateOfMatterGas;
```

Here's how Swift imports them:

```
1 struct StateOfMatter : RawRepresentable {  
2     typealias RawValue = String  
3  
4     init(rawValue: RawValue)  
5  
6     var rawValue: RawValue { get }  
7  
8     static var solid: StateOfMatter { get }  
9     static var liquid: StateOfMatter { get }  
10    static var gas: StateOfMatter { get }  
11}
```

Constants imported from a type declaration marked with the `NS_EXTENSIBLE_STRING_ENUM` macro can be extended in Swift code to add new values.

```
1 extension StateOfMatter {  
2     static var plasma: StateOfMatter {  
3         return StateOfMatter(rawValue: "plasma")  
4     }  
5 }
```

Functions

Swift imports any function declared in a C header as a Swift global function. For example, consider the following C function declarations:

```
1 int product(int multiplier, int multiplicand);  
2 int quotient(int dividend, int divisor, int  
               *remainder);  
3  
4 struct Point2D createPoint2D(float x, float y);  
5 float distance(struct Point2D from, struct Point2D  
                  to);
```

Here's how Swift imports them:

```
1 func product(_ multiplier: Int32, _ multiplicand:  
              Int32) -> Int32  
  
2 func quotient(_ dividend: Int32, _ divisor: Int32,  
                 _ remainder: UnsafeMutablePointer<Int32>) -  
                 > Int32  
  
3  
  
4 func createPoint2D(_ x: Float, _ y: Float) ->  
                    Point2D  
  
5 func distance(_ from: Point2D, _ to: Point2D) ->  
                    Float
```

Variadic Functions

In Swift, you can call C variadic functions, such as `vasprintf`, using the `getVaList(_:_)` or `withVaList(_:_:_)` functions. The `getVaList(_:_)` function takes an array of `CVarArg` values and returns a `CVaListPointer` value, whereas the `withVaList(_:_:_)` provides this value within the body a closure parameter rather than returning it directly. The resulting `CVaListPointer` value is then passed to the `va_list` argument of the C variadic function.

For example, here's how to call the `vasprintf` function in Swift:

```
1 func swiftprintf(format: String, arguments:  
                      CVarArg...) -> String? {
```

```
2     return withVaList(arguments) { va_list in
3
4         var buffer: UnsafeMutablePointer<Int8>? =
5             nil
6
7         return format.withCString { CString in
8             guard vasprintf(&buffer, CString,
9                 va_list) != 0 else {
10                 return nil
11             }
12         }
13     }
14 }
```

```
    }

}

print(swiftprintf(format: " $\sqrt{2} \approx %g$ ", arguments:
    sqrt(2.0))!)

// Prints " $\sqrt{2} \approx 1.41421$ "
```

NOTE

Optional pointers cannot be passed to the `withVaList(_:_:invoke:)` function. Instead, use the `Int.init(bitPattern:)` initializer to interpret the optional pointer as an `Int`, which has the same C variadic calling conventions as a pointer on all supported platforms.

Structures

Swift imports any C structure declared in a C header as a Swift structure. The imported Swift structure contains a stored property for each C structure field and an initializer whose parameters correspond to the stored properties. If all of the imported members have default values, Swift also provides a default initializer that takes no arguments. For example, given the following C structure:

```
1 struct Color {  
2     float r, g, b;  
3 };  
4 typedef struct Color Color;
```

Here's the corresponding Swift type:

```
1 public struct Color {  
2     var r: Float  
3     var g: Float  
4     var b: Float  
5  
6     init()  
7     init(r: Float, g: Float, b: Float)  
8 }
```

Importing Functions as Type Members

C APIs, such as the Core Foundation framework, often provide functions that create, access, or modify C structures. You can use the `CF_SWIFT_NAME` macro in your own code to have Swift import C functions as members of the imported structure type. For example, given the following C function declarations:

```
1 Color ColorCreateWithCMYK(float c, float m, float
  y, float k)
  CF_SWIFT_NAME(Color.init(c:m:y:k));
2
3 float ColorGetHue(Color color)
  CF_SWIFT_NAME(getter:Color.hue(self:));
```

```
4 void ColorSetColorHue(Color color, float hue)
      CF_SWIFT_NAME(setter:Color.hue(self:newValue:
5
6 Color ColorDarkenColor(Color color, float amount)
      CF_SWIFT_NAME(Color.darken(self:amount:));
7
8 extern const Color ColorBondiBlue
      CF_SWIFT_NAME(Color.bondiBlue);
9
```

```
Color ColorGetCalibrationColor(void)
    CF_SWIFT_NAME(getter:Color.calibration());

Color ColorSetCalibrationColor(Color color)
    CF_SWIFT_NAME(setter:Color.calibration(newValue))
```

Here's how Swift imports them as type members:

```
1 extension Color {
2     init(c: Float, m: Float, y: Float, k: Float)
3
4     var hue: Float { get set }
```

```
5  
6    func darken(amount: Float) -> Color  
7  
8    static var bondiBlue: Color  
9  
10   static var calibration: Color  
11 }
```

The argument passed to the `CF_SWIFT_NAME` macro uses the same syntax as the `#selector` expression. The use of `self` in a `CF_SWIFT_NAME` argument is used for instance methods to refer to the method receiver.

NOTE

You cannot reorder or change the number of arguments for type members imported using the `CF_SWIFT_NAME` macro. To provide a refined Swift interface to C functionality, create an overlay by declaring new Swift functions that make the necessary C function calls in their implementation.

Enumerations

Swift imports any C enumeration marked with the `NS_ENUM` macro as a Swift enumeration with an `Int` raw value type. The prefixes to C enumeration case names are removed when they are imported into Swift, whether they're defined in system frameworks or in custom code.

For example, the C enumeration below is declared using the `NS_ENUM` macro.

```
1  typedef NS_ENUM(NSInteger, UITableViewCellStyle) {  
2      UITableViewCellStyleDefault,  
3      UITableViewCellStyleValue1,  
4      UITableViewCellStyleValue2,  
5      UITableViewCellStyleSubtitle  
6 };
```

In Swift, it's imported like this:

```
1 enum UITableViewCellStyle: Int {  
2     case `default`  
3     case value1  
4     case value2  
5     case subtitle  
6 }
```

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

```
let cellStyle: UITableViewCellStyle = .default
```

NOTE

C enumerations imported by Swift do not fail when initializing with a raw value that does not correspond to an enumeration case. This is done for compatibility with C, which allows any value to be stored in an enumeration, including values used internally but not exposed in headers.

A C enumeration that is not marked with the `NS_ENUM` or `NS_OPTIONS` macro is imported as a Swift structure. Each member of the C enumeration is imported as a global read-only computed property of the structure's type—not as a member of the Swift structure itself.

For example, the following C enumeration is not declared using the `NS_ENUM` macro:

```
1  typedef enum {
2
2      MessageDispositionUnread = 0,
3
3      MessageDispositionRead = 1,
4
4      MessageDispositionDeleted = -1,
5
5  } MessageDisposition;
```

In Swift, it's imported like this:

```
1 struct MessageDisposition: RawRepresentable,  
2   Equatable {}  
3  
4 var MessageDispositionUnread: MessageDisposition {  
5   get }  
6  
7 var MessageDispositionRead: MessageDisposition {  
8   get }  
9  
10 var MessageDispositionDeleted: MessageDisposition {  
11   get }
```

Swift automatically synthesizes conformance to the Equatable protocol for imported C enumeration types.

Option Sets

Swift also imports C enumerations marked with the `NS_OPTIONS` macro as a Swift option set. Option sets behave similarly to imported enumerations by truncating their prefixes to option value names.

For example, consider the following Objective-C options declaration:

```
1  typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing)  
2  
3      {  
4          UIViewAutoresizingNone = 0,  
5          ...  
6      }
```

3 UIViewAutoresizingFlexibleLeftMargin = 1
 << 0,

4 UIViewAutoresizingFlexibleWidth = 1
 << 1,

5 UIViewAutoresizingFlexibleRightMargin = 1
 << 2,

6 UIViewAutoresizingFlexibleTopMargin = 1
 << 3,

7 UIViewAutoresizingFlexibleHeight = 1
 << 4,

8 UIViewAutoresizingFlexibleBottomMargin = 1

```
9     };
```

<< 5

In Swift, it's imported like this:

```
1  public struct UIViewAutoresizing : OptionSet {  
2      public init(rawValue: UInt)  
3  
4      public static var flexibleLeftMargin:  
5          UIViewAutoresizing { get }  
6  
5      public static var flexibleWidth:  
6          UIViewAutoresizing { get }
```

```
6     public static var flexibleRightMargin:  
7         UIViewAutoresizing { get }  
  
7     public static var flexibleTopMargin:  
8         UIViewAutoresizing { get }  
  
8     public static var flexibleHeight:  
9         UIViewAutoresizing { get }  
  
9     public static var flexibleBottomMargin:  
        UIViewAutoresizing { get }  
}
```

In Objective-C, an option set is a bit mask of integer values. You

use the bitwise OR operator (`|`) to combine option values, and the bitwise AND operator (`&`) to check for option values. You create a new option set from a constant value or expression. An empty option set is represented by the constant zero (`0`).

In Swift, option sets are represented by structures conforming to the `OptionSet` protocol, with static variables for each option value. You can create a new option set value using an array literal, and access option values with a leading dot (`.`), similar to an enumeration. An empty option set can be created from an empty array literal (`[]`) or by calling its default initializer.

NOTE

When importing C enumeration marked with the `NS_OPTIONS` macro, Swift marks any members that have a value of `0` as unavailable, because Swift uses an empty option set to specify no options.

Option sets behave like Swift's `Set` collection type. You use the `insert(_:_:)` or `formUnion(_:_:)` methods to add option values, the `remove(_:_:)` or `subtract(_:_:)` methods to remove option values, and the `contains(_:_:)` method to check for an option value.

```
1 let options: Data.Base64EncodingOptions = [  
2     .lineLength64Characters,  
3     .endLineWithLineFeed  
4 ]  
5 let string = data.base64EncodedString(options:  
    options)
```

Unions

Swift has only partial support of C `union` types. When importing C aggregates containing unions, Swift cannot access unsupported

fields. However, C and Objective-C APIs that have arguments of those types or return values of those types can be used in Swift.

Bit Fields

Swift imports bit fields in structures, such those found in Foundation's `NSDecimal` type, as computed properties. When accessing a computed property corresponding to a bit field, Swift automatically converts the value to and from compatible Swift types.

Unnamed Structure and Union Fields

C `struct` and `union` types may define unnamed `struct` and `union` types as fields. Swift does not support unnamed structures, so these fields are imported with nested types with names that take the form `__Unnamed_fieldName`.

For example, consider a C structure named `Pie` that contains a `crust` field with an unnamed `struct` type and a `filling` field with an unnamed `union` type:

```
1  struct Pie {  
2      struct { bool flakey; } crust;  
3      union { int fruit; int meat; } filling;  
4 }
```

When imported by Swift, the `crust` property has type
`Pie.__Unnamed_crust` and the `filling` property has type
`Pie.__Unnamed_filling`.

Pointers

Whenever possible, Swift avoids giving you direct access to pointers. There are, however, 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 return types, variables, and arguments, the following mappings apply:

C Syntax

Swift Syntax

const Type *

UnsafePointer<Type>

Type *

UnsafeMutablePointer<Type>

For class types, the following mappings apply:

C Syntax	Swift Syntax
Type *	UnsafePointer<Type>
const *	
Type * __strong *	UnsafeMutablePointer<Type>
Type **	AutoreleasingUnsafeMutablePointer<Type>

Pointers to untyped, raw memory are represented in Swift by the `UnsafeRawPointer` and `UnsafeMutableRawPointer` types. If the type of the value pointed to by a C pointer cannot be represented by Swift, such as an incomplete struct type, the pointer is imported as an `OpaquePointer`.

Constant Pointers

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

- An `UnsafePointer<Type>`, `UnsafeMutablePointer<Type>`, or `AutoreleasingUnsafeMutablePointer<Type>` value, which is converted to `UnsafePointer<Type>` if necessary.
- A `String` value, if `Type` is `Int8` or `UInt8`. The string will automatically be converted to UTF8 in a buffer, and a pointer to that buffer is passed to the function.
- An in-out expression that contains a mutable variable, property, or subscript reference of type `Type`, which is passed as a pointer to the address of the left-hand side identifier.

- A [Type] value, which is passed as a pointer to the start of the array.

The pointer passed to the function is guaranteed to be valid only for the duration of the function call. Don't try to persist the pointer and access it after the function has returned.

If you have declared a function like this one:

```
1 func takesAPointer(_ p: UnsafePointer<Float>) {  
2     // ...  
3 }
```

You can call it in any of the following ways:

```
1 var x: Float = 0.0  
2 takesAPointer(&x)  
3 takesAPointer([1.0, 2.0, 3.0])
```

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

If you have declared a function like this one:

```
1 func takesAVoidPointer(_ p: UnsafeRawPointer?) {  
2     // ...  
3 }
```

You can call it in any of the following ways:

```
1 var x: Float = 0.0, y: Int = 0
2 takesAVoidPointer(&x)
3 takesAVoidPointer(&y)
4 takesAVoidPointer([1.0, 2.0, 3.0] as [Float])
5 let intArray = [1, 2, 3]
6 takesAVoidPointer(intArray)
```

Mutable Pointers

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

- An `UnsafeMutablePointer<Type>` value
- An in-out expression of type `Type` that contains a mutable variable, property, or subscript reference, which is passed as a pointer to the address of the mutable value.
- An in-out expression of type `[Type]` that contains a mutable variable, property, or subscript reference, which is passed as a pointer to the start of the array, and is lifetime-extended for the duration of the call

If you have declared a function like this one:

```
1 func takesAMutablePointer(_ p:  
2                             UnsafeMutablePointer<Float>) {  
3  
4 }
```

You can call it in any of the following ways:

```
1 var x: Float = 0.0  
2 var a: [Float] = [1.0, 2.0, 3.0]  
3 takesAMutablePointer(&x)  
4 takesAMutablePointer(&a)
```

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

If you have declared a function like this one:

```
1 func takesAMutableVoidPointer(_ p:  
2                                     UnsafeMutableRawPointer?) {  
3     // ...  
4 }
```

You can call it in any of the following ways:

```
1 var x: Float = 0.0, y: Int = 0
2 var a: [Float] = [1.0, 2.0, 3.0], b: [Int] = [1, 2,
3]
4 takesAMutableVoidPointer(&x)
5 takesAMutableVoidPointer(&y)
6 takesAMutableVoidPointer(&a)
7 takesAMutableVoidPointer(&b)
```

Autoreleasing Pointers

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

- An `AutoreleasingUnsafeMutablePointer<Type>` value
- An in-out expression that contains a mutable variable, property, or subscript reference of type `Type`, which is copied bitwise into 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:

```
1 func takesAnAutoreleasingPointer(_ p:  
2                                     AutoreleasingUnsafeMutablePointer<NSDate?>)  
3 {  
4     // ...  
5 }
```

You can call it in the following way:

```
1 var x: NSDate? = nil  
2 takesAnAutoreleasingPointer(&x)
```

Types that are pointed to are not bridged. For example, NSString ** comes over to Swift as

`AutoreleasingUnsafeMutablePointer<NSString?>`, not
`AutoreleasingUnsafeMutablePointer<String?>`.

Function Pointers

C function pointers are imported into Swift as closures with C function pointer calling convention, denoted by the `@convention(c)` attribute. For example, a function pointer that has the type `int (*) (void)` in C is imported into Swift as `@convention(c) () -> Int32`.

When calling a function that takes a function pointer argument, you can pass a top-level Swift function, a closure literal, or `nil`. You can also pass a closure property of a generic type or a generic method as long as no generic type parameters are referenced in the closure's argument list or body. For example, consider Core Foundation's

`CFArrayCreateMutable(_:_:_:_)` function. The `CFArrayCreateMutable(_:_:_:_)` function takes a `CFArrayCallBacks` structure, which is initialized with function pointer callbacks:

```
1 func customCopyDescription(_ p: UnsafeRawPointer?)  
2     -> Unmanaged<CFString>? {  
3  
4  
5 var callbacks = CFArrayCallBacks(  
6     version: 0,  
7     retain: nil,
```

```
8     release: nil,  
9     copyDescription: customCopyDescription,  
  
equal: { (p1, p2) -> DarwinBoolean in  
  
    // return Bool value  
  
}  
)  
  
var mutableArray = CFArrayCreateMutable(nil, 0,  
    &callbacks)
```

In the example above, the `CFArrayCallbacks` initializer uses `nil`

values as arguments for the `retain` and `release` parameters, the `customCopyDescription(_:)` function as the argument for the `customCopyDescription` parameter, and a closure literal as the argument for the `equal` parameter.

NOTE

Only Swift function types with C function reference calling convention may be used for function pointer arguments. Like a C function pointer, a Swift function type with the `@convention(c)` attribute does not capture the context of its surrounding scope.

For more information, see Type Attributes in *The Swift Programming Language (Swift 3)*.

Null Pointers

In Objective-C, a pointer type declaration can use the `_Nullable` and `_Nonnull` annotations to specify whether or not that pointer may have a `nil` or `NULL` value. In Swift, a null pointer is represented by a `nil` value of an optional pointer type. Pointer initializers taking the integer representation of an address in memory are failable. A non-optional pointer type cannot be assigned a `nil` value.

The following mappings apply:

Objective-C Syntax

const Type * _Nonnull

const Type * _Nullable

const Type *
_Null_unspecified

Swift Syntax

UnsafePointer<Type>

UnsafePointer<Type>?

UnsafePointer<Type>!

NOTE

Prior to Swift 3, both nullable and non-nullable pointers were represented by a non-optional unsafe pointer type. When migrating existing code to the latest version of Swift, you may need to mark any pointers that are initialized with a `nil` literal value to be optional.

Pointer Arithmetic

When working with opaque data types, you may need to perform unsafe pointer operations. You can use the arithmetic operators on Swift pointer values to create new pointers at a specified offset.

```
1 let pointer: UnsafePointer<Int8>
2 let offsetPointer = pointer + 24
3 // offsetPointer is 24 strides ahead of pointer
```

NOTE

See [Data Type Size Calculation](#) for more information about how Swift calculates the sizes of data types and values.

Data Type Size Calculation

In C, the `sizeof` and `alignof` operators return the size and alignment of any variable or data type. In Swift, you use `MemoryLayout<T>` to access the memory layout of the parameterized type `T` through the `size`, `stride`, and `alignment` properties. For example, the `timeval` structure in Darwin has a size and stride of 16 and an alignment of 8:

```
1 print(MemoryLayout<timeval>.size)  
2 // Prints "16"  
  
3 print(MemoryLayout<timeval>.stride)  
4 // Prints "16"  
  
5 print(MemoryLayout<timeval>.alignment)  
6 // Prints "8"
```

You use this when calling C functions from Swift that take the size of a type or value as an argument. For example, the `setsockopt(_:_:_:_:_:_)` function can specify a `timeval` value as a receive timeout option (`S0_RCVTIMEO`) for a socket by passing a pointer to that value and the length of that value:

```
1 let sockfd = socket(AF_INET, SOCK_STREAM, 0)

2 var optval = timeval(tv_sec: 30, tv_usec: 0)

3 let optlen = socklen_t(MemoryLayout<timeval>.size)

4 if setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO,
               &optval, optlen) == 0 {
    // ...
}

5
6 }
```

For more information, see the [MemoryLayout<T> API reference](#).

One-Time Initialization

In C, the `pthread_once()` function in POSIX and the `dispatch_once()` and `dispatch_once_f()` functions in Grand Central Dispatch provide mechanisms for executing initialization code exactly once. In Swift, global constants and stored type properties are guaranteed to be initialized only once, even when accessed across multiple threads simultaneously. Because this functionality is provided through language features, the corresponding POSIX and Grand Central Dispatch C function calls are not exposed by Swift.

Preprocessor Directives

The Swift compiler does not include a preprocessor. Instead, it takes advantage of compile-time attributes, conditional compilation

blocks, 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.

Conditional Compilation Blocks

Swift code and Objective-C code are conditionally compiled in different ways. Swift code can be conditionally compiled using *conditional compilation blocks*. For example, if you compile the code below using `swift -D DEBUG_LOGGING` to set the `DEBUG_LOGGING` conditional compilation flag, the compiler includes the code in the body of the conditional compilation block.

```
1 #if DEBUG_LOGGING  
2     print("Flag enabled.")  
3 #endif
```

A *compilation condition* can include the literal `true` and `false` values, custom conditional compilation flags (specified using `-D <#flag#>`), and the platform conditions listed in the table below.

Function	Valid arguments
os()	macOS, iOS, watchOS, tvOS, Linux
arch()	x86_64, arm, arm64, i386
swift()	>= followed by a version number

NOTE

The `arch(arm)` platform condition does not return `true` for ARM 64 devices. The `arch(i386)` platform condition returns `true` when code is compiled for the 32-bit iOS simulator.

You can combine compilation conditions using the `&&` and `||`

operators, negate them with the `!` operator, and add branches with `#elseif` and `#else` compilation directives. You can also nest conditional compilation blocks within other conditional compilation blocks.

```
1  #if arch(arm) || arch(arm64)
2
3      #if swift(>=3.0)
4
5          print("Using Swift 3 ARM code")
6
7      #else
8
9          print("Using Swift 2.2 ARM code")
10
11     #endif
12
13 #elseif arch(x86_64)
```

```
8 print("Using 64-bit x86 code.)  
9     #else  
      print("Using general code.")  
#endif
```

In contrast with condition compilation in the C preprocessor, conditional compilation blocks 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. However, there is an exception if the compilation condition includes a `swift()` platform condition: The statements are parsed only if the compiler's version of Swift matches what is specified in the platform condition. This exception ensures that an older compiler doesn't attempt to parse syntax introduced in a

newer version of Swift.

Mix and Match

Swift and Objective-C in the Same Project

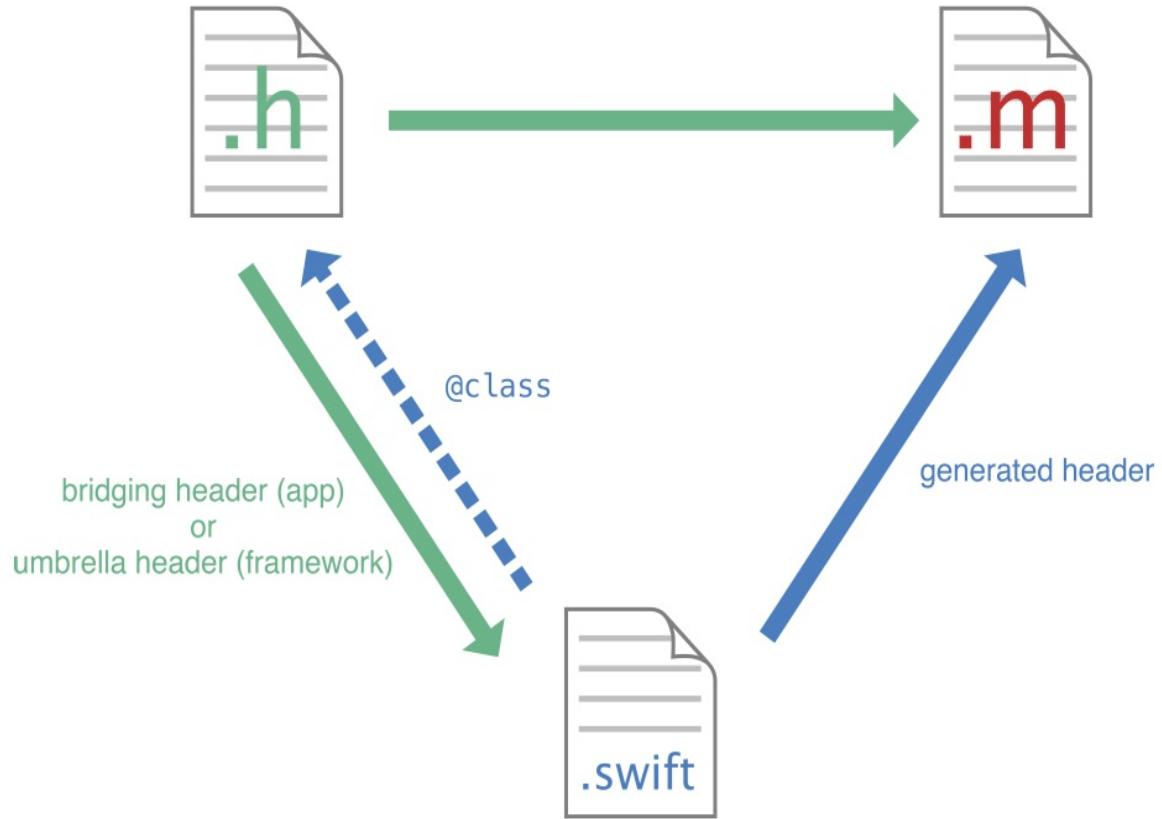
Swift's compatibility with Objective-C lets you create a project that contains files written in either language. You can use this feature, called *mix and match*, to write apps that have a mixed-language codebase. Using mix and match, you can implement part of your app's functionality using the latest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



Would you like to configure an Objective-C bridging header?

Adding this file to MyApp will create a mixed Swift and Objective-C target. Would you like Xcode to automatically configure a bridging header to enable classes to be accessed by both languages?

Cancel

Don't Create

Create Bridging Header

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"`. (You'll learn more about the product module name later, in [Naming Your Product Module](#).)

Alternatively, you can create a bridging header yourself by choosing File > New > File > (iOS, watchOS, tvOS, or macOS) > Source > Header File.

You'll need to edit the bridging header 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:

```
1 #import "XYZCustomCell.h"  
2 #import "XYZCustomView.h"  
3 #import "XYZCustomViewController.h"
```

2. In Build Settings, in Swift Compiler - Code Generation, make sure the Objective-C Bridging Header build setting under has a path to the bridging header file.

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.

```
1 let myCell = XYZCustomCell()  
2 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 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"`. (You'll learn more about the product module name later, in [Naming Your Product Module](#).)

By default, the generated header contains interfaces for Swift declarations marked with the `public` or `open` modifier. It also contains those marked with the `internal` modifier if your app target has an Objective-C bridging header. Declarations marked with the `private` or `fileprivate` modifier do not appear in the generated header. Private declarations are not exposed to Objective-C unless

they are explicitly marked with `@IBAction`, `@IBOutlet`, or `@objc` as well. If your app target is compiled with testing enabled, a unit test target can access any declaration with the `internal` modifier as if they were declared with the `public` modifier by prepending `@testable` to the product module import statement.

For more information on access-level modifiers, see Access Control in *The Swift Programming Language (Swift 3)*.

You don't need to do anything special to create the generated header file—just 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 before 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 "ProductName-Swift.h"
```

The 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](#).

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

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:

```
1 #import <XYZ/XYZCustomCell.h>
2 #import <XYZ/XYZCustomView.h>
3 #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.

```
1 let myOtherCell = XYZCustomCell()  
2 myOtherCell.subtitle = "Another 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 your Swift code from.

Because the generated header for a framework target is part of the framework's public interface, only declarations marked with the `public` or `open` modifier appear in the generated header for a

framework target.

Swift methods and properties that are marked with the `internal` modifier and declared within a class that inherits from an Objective-C class are accessible to the Objective-C runtime. However, they are not accessible at compile time and do not appear in the generated header for a framework target.

For more information on access-level modifiers, see Access Control in *The Swift Programming Language (Swift 3)*.

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>
```

The 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](#).

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

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

Any language framework

```
import  
FrameworkName
```

Import into Objective-C

```
@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.

```
1 MySwiftClass *swiftObject = [[MySwiftClass alloc]  
2  
    init];  
  
[swiftObject swiftMethod];
```

A Swift class must be a descendant of an Objective-C class to be accessible and usable in Objective-C. For more information about what you can access from Objective-C and how the Swift interface is imported, see [Swift Type Compatibility](#).

Referencing a Swift Class or Protocol in an Objective-C Header

To avoid cyclical references, don't import Swift code into an Objective-C header (.h) file. Instead, you can forward declare a Swift class or protocol to reference it in an Objective-C interface.

```
1 // MyObjcClass.h  
2  
3 @class MySwiftClass;  
4  
5 @protocol MySwiftProtocol;  
6  
7 @interface MyObjcClass : NSObject  
8  
9 - (MySwiftClass *)returnSwiftClassInstance;  
10  
11 - (id  
12     <MySwiftProtocol>)returnInstanceAdoptingSwift
```

```
8 // ...
9 @end
```

Forward declarations of Swift classes and protocols can only be used as types for method and property declarations.

Declaring a Swift Protocol That Can Be Adopted by an Objective-C Class

To create a Swift protocol that can be adopted by an Objective-C class, mark the `protocol` declaration with the `@objc` attribute.

```
1 @objc public protocol MySwiftProtocol {  
2     func requiredMethod()  
3  
4     @objc optional func optionalMethod()  
5 }
```

A protocol declares all initializers, properties, subscripts, and methods that an Objective-C class must implement in order to conform to the protocol. Any optional protocol requirements must be marked with the `@objc` attribute and have the `optional` modifier.

Adopting a Swift Protocol in an Objective-C

Implementation

An Objective-C class can adopt a Swift protocol in its implementation (.m) file by importing the Xcode-generated header for Swift code and using a class extension.

```
1 // MyObjcClass.m  
2  
3  
4 @interface MyObjcClass () <MySwiftProtocol>  
5 // ...  
6 @end  
7  
8 @implementation MyObjcClass  
9 // ...  
@end
```

Declaring a Swift Error Type That Can Be Used from Objective-C

Swift enumerations conforming to the `Error` protocol and declared with the `@objc` attribute produce an `NS_ENUM` declaration, as well as an `NSString` constant for the corresponding error domain in the generated header. For example, given the following Swift enumeration declaration:

```
1  @objc public enum CustomError: Int, Error {  
2      case a, b, c  
3 }
```

Here's the corresponding Objective-C declaration in the generated header:

```
1 // Project-Swift.h  
2  
3 typedef SWIFT_ENUM(NSInteger, CustomError) {  
4  
    CustomErrorA = 0,  
4  
    CustomErrorB = 1,  
4  
    CustomErrorC = 2,  
6  
};  
7  
static NSString * const CustomErrorDomain =  
    @"Project.CustomError";
```

Overriding Swift Names for Objective-C Interfaces

The Swift compiler automatically imports Objective-C code as conventional Swift code. It imports Objective-C class factory methods as Swift initializers, and Objective-C enumeration cases truncated names.

There may be edge cases in your code that are not automatically handled. If you need to change the name imported by Swift of an Objective-C method, enumeration case, or option set value, you can use the `NS_SWIFT_NAME` macro to customize how a declaration is imported.

Class Factory Methods

If the Swift compiler fails to identify a class factory method, you can use the `NS_SWIFT_NAME` macro, passing the Swift signature of the initializer to have it imported correctly. For example:



```
+ (instancetype)recordWithRPM:(NSUInteger)RPM  
    NS_SWIFT_NAME(init(RPM:));
```

If the Swift compiler mistakenly identifies a method as a class factory method, you can use the `NS_SWIFT_NAME` macro, passing the Swift signature of the method to have it imported correctly. For example:

```
+ (id)recordWithQuality:(double)quality  
    NS_SWIFT_NAME(record(quality));
```

Enumerations

By default, Swift imports enumerations by truncating enumeration value name prefixes. To customize the name of an enumeration case, you can use the `NS_SWIFT_NAME` macro, passing the Swift enumeration case name. For example:

```
1  typedef NS_ENUM(NSInteger, ABCRecordSide) {  
2      ABCRecordSideA,  
3      ABCRecordSideB NS_SWIFT_NAME(FlipSide),  
4  };
```

Making Objective-C Interfaces Unavailable in Swift

Some Objective-C interfaces may not be suitable or necessary to be exposed as Swift interfaces. To prevent an Objective-C declaration from being imported by Swift, use the `NS_SWIFT_UNAVAILABLE` macro, passing a message directing API consumers to any alternatives that may exist.

For example, an Objective-C class providing a convenience initializer that takes variadic arguments for keys-value pairs may advise a Swift consumer to use a dictionary literal instead:

```
+ (instancetype)collectionWithValues:(NSArray*)values forKeys:(NSArray<NSCopying>*)keys  
NS_SWIFT_UNAVAILABLE("Use a dictionary  
literal instead");
```

Attempting to call the `+collectionWithValues:forKeys:` method from Swift code will result in a compiler error.

Refining Objective-C Declarations

You can use the `NS_REFINED_FOR_SWIFT` macro on an Objective-C method declaration to provide a refined Swift interface in an extension, while keeping the original implementation available to be called from the refined interface. For instance, an Objective-C method that takes one or more pointer arguments could be refined in Swift to return a tuple of values.

- Initializer methods are imported by Swift with double underscores (`__`) prepended to their first argument labels.
- Object subscripting methods are imported by Swift as methods with double underscores (`__`) prepended to their base names, rather than as a Swift subscript, if either the getter or setter method is marked as `NS_REFINED_FOR_SWIFT`.
- Other methods are imported with double underscores (`__`) prepended to their base names.

Given the following Objective-C declarations:

```
1 @interface Color : NSObject  
2  
3 - (void)getRed:(nullable CGFloat *)red  
4  
5         green:(nullable CGFloat *)green  
6  
7         blue:(nullable CGFloat *)blue  
8  
9         alpha:(nullable CGFloat *)alpha  
10  
11 NS_REFINED_FOR_SWIFT;  
12  
13 @end
```

You can provide a refined Swift interface in an extension like this:

```
1  extension Color {  
2  
    var RGBA: (red: CGFloat, green: CGFloat, blue:  
        CGFloat, alpha: CGFloat) {  
        3      var r: CGFloat = 0.0  
        4      var g: CGFloat = 0.0  
        5      var b: CGFloat = 0.0  
        6      var a: CGFloat = 0.0  
        7      __getRed(red: &r, green: &g, blue: &b,  
alpha: &a)  
    }  
}
```

```
8     return (red: r, green: g, blue: b, alpha:  
9         a)  
 }  
 }
```

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.

NOTE

You cannot override the product module name of a framework.

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 (`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 (`SWIFT_OBJC_BRIDGING_HEADER`) build setting under Swift Compiler - Code Generation is set to a path to the bridging header file relative to your project (for example, “`MyApp/MyApp-Bridging-Header.h`”).
- Xcode uses your product module name (`PRODUCT_MODULE_NAME`)—not your target name (`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

- 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](#).
- If you use your own Objective-C types in your Swift code, make sure to import the Objective-C headers for those types before importing the Swift generated header into the Objective-C `.m` file you want to use your Swift code from.
- Swift declarations marked with the `private` or `fileprivate` modifier do not appear in the generated header. Private declarations are not exposed to Objective-C unless they are explicitly marked with `@IBAction`, `@IBOutlet`, or `@objc` as

well.

- For app targets, declarations marked with the `internal` modifier appear in the generated header if the app target has an Objective-C bridging header.
- For framework targets, only declarations with the `public` or `open` modifier appear in the generated header. You can still use Swift methods and properties that are marked with the `internal` modifier from within the Objective-C part of your framework, as long they are declared within a class that inherits from an Objective-C class. For more information on access-level modifiers, see *Access Control in The Swift Programming Language (Swift 3)*.

Migration

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 has 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 goes 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`, `watchOS`, `tvOS`, or `macOS`) `> Source > Swift File`. 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 [Importing Code from Within the Same App Target](#).
- ✓ To make your Swift class accessible and usable back in Objective-C, make it a descendant of an Objective-C class. 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 uses to reference the Swift class. For more information on `@objc`, see [Swift Type Compatibility](#).

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 and Protocols with Objective-C Behavior](#).
- ✓ 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](#).
- ✓ 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 Frameworks](#).
- ✓ As you incorporate Cocoa patterns into your Swift class, see [Adopting Cocoa Design Patterns](#) for information on translating common design patterns.
- ✓ For considerations on translating your properties from

Objective-C to Swift, read Properties in *The Swift Programming Language (Swift 3)*.

- ✓ 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:

```
1  var enabled: Bool {  
2      @objc(isEnabled) get {  
3          // ...  
4      }  
5  }
```

- ✓ 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 "ProductName-Swift.h"`), as described in [Importing Code from Within the Same App Target](#).
- ✓ 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

Even though each migration experience is different depending on your existing codebase, 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.
- 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](#).
- 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.

Revision History

Document Revision History

This table describes the changes to *Using Swift with Cocoa and Objective-C*.

Date	Notes
	<ul style="list-style-type: none">• Updated for Swift 3.0.• Added information about bridged value types and renamed types in the Swift Foundation overlay to the <u>Working with Cocoa Frameworks</u> chapter.

- Added information about global constants annotated with the `NS_STRING_ENUM` and `NS_EXTENSIBLE_STRING_ENUM` macros and functions annotated with the `CF_SWIFT_NAME` macro to the [Interacting with C APIs](#) chapter.
- Added information about the `#keyPath` expression to the [Keys and Key Paths](#) section.
- Added information about the `#selector` expression to the [Selectors](#) section.
- Added information about imported Objective-C classes with generic parameterization to the [Lightweight Generics](#) section.

2016-
09-13

- Added information about Objective-C class properties to the [Accessing Properties](#) section.
- Added information about how nullable pointers are imported as optionals to the [Null Pointers](#) section.
- Added information about performing unsafe pointer operations to the [Pointer Arithmetic](#) section.
- Added a note to the [Enumerations](#) section about initializing imported C enumerations with raw values.
- Added a note to the [Option Sets](#) section with information about imported C enumerations cases that have a value of `0`.

- Added information about defining a Swift protocol that Objective-C classes can conform to in the [Declaring Protocols](#) and [Declaring a Swift Protocol That Can Be Adopted by an Objective-C Class](#) sections.
- Added information about autorelease pool blocks and the `autoreleasepool(_:)` function in the [Autorelease Pools](#) section.
- Added information about the `os_log(_:_:dso:log:type:_:)` function in the [Unified Logging](#) section.
- Updated the [Interacting with Objective-C APIs](#) chapter with information about how Swift imports Objective-C `id` types as Any.

- Updated the [Hashing](#) section with information about the `AnyHashable` type.
- Updated the [Pointers](#) section with information about the `UnsafeRawPointer` and `UnsafeMutableRawPointer` types.
- Updated the [Data Type Size Calculation](#) section with information about the `MemoryLayout` type.
- Updated the [Swift and Objective-C in the Same Project](#) chapter with information about the `open` and `fileprivate` access modifiers.
- Updated for Swift 2.2.

- Added information about thread-safe initialization to the [One-Time Initialization](#) section of the [Interacting with C APIs](#) chapter.
- Added the [Unnamed Structure and Union Fields](#) section to the [Interacting with C APIs](#) chapter.
- Added the [Variadic Functions](#) section to the [Interacting with C APIs](#) chapter.
- Updated the [Pointers](#) section with information about the duration of pointers passed to `inout` parameters.
- Updated the [Setting Up Your Swift Environment](#) section with information about base SDK requirements for Swift apps.

2016-
03-21

- Updated the [Configuring Swift Interfaces in Objective-C](#) section with information about using the `@objc(name)` attribute with Swift enumerations.
- Updated the [Selectors](#) section with information about the Swift `#selector` expression.
- Updated the [Conditional Compilation Blocks](#) section with information about the `swift` build configuration.
- Updated the [Object Comparison](#) section and [Hashing](#) section in the [Interacting with Objective-C APIs](#) chapter.
- Updated and expanded the [Swift Type](#)

[Compatibility](#) section in the [Interacting with Objective-C APIs](#) chapter.

- Updated and expanded the Bridging Collections section in the [Working with Cocoa Frameworks](#) chapter.
- Fixed the `NS_SWIFT_NAME` example from the [Overriding Swift Names for Objective-C Interfaces](#) section in the [Swift and Objective-C in the Same Project](#) chapter.
- Updated for Swift 2.1.
- Added the [Bit Fields](#) section and updated the

[Unions](#) section in the [Interacting with C APIs](#) chapter.

- Updated the [Enumerations](#) section with information about how C enumerations not marked with the `NS_ENUM` or `NS_OPTIONS` macro are imported by Swift.
- Updated the [API Availability](#) section and the [Conditional Compilation Blocks](#) section with information about tvOS.
- Clarified the behavior of Objective-C atomicity attributes in Swift in the [Accessing Properties](#) section.
- Added various references to tvOS and watchOS platform support.

2015-
10-20

- Updated screenshots of Xcode user interface.
- Simplified the instructions in the [Basic Setup](#) chapter.
- Updated the introductions of the [Interacting with Objective-C APIs](#) chapter.
- Clarified the behavior of the `AnyObject` protocol in the [`id` Compatibility](#) section.
- Added the [Unrecognized Selectors and Optional Chaining](#) section and the [Downcasting `Any`](#) section to the [Interacting with Objective-C APIs](#) chapter.

- Updated for Swift 2.0.
- Added the [Catching and Handling an Error](#) section to the [Adopting Cocoa Design Patterns](#) chapter.
- Added the [Throwing an Error](#) section to the [Adopting Cocoa Design Patterns](#) chapter.
- Added the [Converting Errors to Optional Values](#) section to the [Adopting Cocoa Design Patterns](#) chapter with information about the `try?` keyword.
- Added the [Refining Objective-C Declarations](#) section to the [Swift and Objective-C in the Same Project](#) chapter with information about the `NS_REFINED_FOR_SWIFT` macro.

- Updated the discussion of the `@NSManaged` attribute in the [Implementing Core Data Managed Object Subclasses](#) section, now that the attribute can be applied to certain instance methods.
- Added the [Unsafe Invocation of Objective-C Methods](#) section to the [Interacting with Objective-C APIs](#) chapter now that the `performSelector` API family is available in Swift.
- Updated the [Error Handling](#) section with information about the `NS_ERROR_RESULT` Objective-C macro.
- Updated the [Configuring Swift Interfaces in Objective-C](#) section with information about

using the `@objc` attribute for Swift enumerations.

- Updated the Bridging Collections section to discuss Swift bridging of parameterized Objective-C collection classes.
- Updated the [Error Handling](#) section to reflect that Swift removes the `WithError` suffix from the selector of an error-producing Objective-C method, when that method is imported as a Swift method that throws.
- Updated the [Failable Initialization](#) section with information about nullability annotations.
- Updated the [Swift Type Compatibility](#) and [Configuring Swift Interfaces in Objective-C](#)

sections now that the `@objc` attribute can only be applied to classes that descend from `NSObject`.

- Updated the [Configuring Swift Interfaces in Objective-C](#) section with information about the `@nonobjc` attribute.
- Added the [Unions](#) section to the [Interacting with C APIs](#) chapter.
- Added the [Option Sets](#) section to the [Interacting with C APIs](#) chapter.
- Added the [API Availability](#) section to the [Adopting Cocoa Design Patterns](#) chapter.
- Added the [Error Handling](#) section to the [Adopting Cocoa Design Patterns](#) chapter.

2015-
09-16

- Updated the [Swift and Objective-C in the Same Project](#) chapter with information about the `@testable` attribute.
- Updated the [Function Pointers](#) section with information about bridging Swift function and closure values with C function pointers.
- Added the [Lazy Initialization](#) section to the [Adopting Cocoa Design Patterns](#) chapter.
- Added the [Serializing](#) section to the [Adopting Cocoa Design Patterns](#) chapter with information about serializing objects.
- Added the [Processing Command-Line Arguments](#) section to the [Adopting Cocoa Design Patterns](#) chapter with information about reading command line flags.

- Added the [Undo](#) section to the [Adopting Cocoa Design Patterns](#) chapter with information about implementing undo support.
- Updated the [Adopting Protocols](#) section to describe how to constrain the type of a variable or constant to multiple protocols.
- Added the [Avoiding Strong Reference Cycles When Capturing self](#) section to the [Interacting with Objective-C APIs](#) chapter.
- Added the [Singleton](#) section to the [Adopting Cocoa Design Patterns](#) chapter.
- Added the [Making Objective-C Interfaces Unavailable in Swift](#) section to the [Swift and](#)

Objective-C in the Same Project chapter.

- Updated the [Accessing Properties](#) section with additional information about how Objective-C property attributes are imported by Swift.
- Added the [NSCoding](#) section to the [Interacting with Objective-C APIs](#) chapter.
- Added the [Memory Managed Objects](#) section to the [Working with Cocoa Frameworks](#) chapter.
- Added the [Sets](#) section to the [Working with Cocoa Frameworks](#) chapter.
- Added a note about Swift target system requirements for executables built from the

command line in the [Setting Up Your Swift Environment](#) section.

- Added to the [Nullability and Optionals](#) section in the [Interacting with Objective-C APIs](#) chapter.
- Added the [Overriding Swift Names for Objective-C Interfaces](#) section to the [Interacting with Objective-C APIs](#) chapter.
- Added the [Data Type Size Calculation](#) section to the [Interacting with C APIs](#) chapter with information about the `sizeof`, `sizeofValue`, `strideof`, and `strideOfValue` functions.
- Updated [Constant Pointers](#) section with information about bridging Swift `String`

values to C pointers.

- Updated the `CFArrayCreateMutable` code sample in the [Function Pointers](#) section.
- Updated for Swift 1.2.
- Updated the [Working with Cocoa Frameworks](#) chapter to use the `as!` operator and bridging semantics in Swift 1.2.
- Fixed the `UIBezierPath` example to correctly create a triangle.

2015-
02-23

2014-
09-15

- Updated for Swift 1.1.
- Added the [Failable Initialization](#) section to the [Interacting with Objective-C APIs](#) chapter.
- Added the [Using Swift Class Names with Objective-C APIs](#) section to the [Writing Swift Classes and Protocols with Objective-C Behavior](#) chapter.
- Corrected the [Live Rendering](#) section which previously stated that live rendering only worked when a designable class was compiled in a framework. This limitation no longer exists.
- Added an implementation of `deinit` and a call to `super` in the last code sample in the

Key-Value Observing section.

- New document that describes various aspects of Swift 1.0 and its compatibility with the Objective-C language and Cocoa/Cocoa Touch frameworks.
- Updated the [Pointers](#) section to reflect that `UnsafePointer` has been replaced with `UnsafeMutablePointer`, `ConstUnsafePointer` has been replaced with `UnsafePointer`, and `AutoreleasingUnsafePointer` has been replaced with

AutoreleasingUnsafeMutablePointer.

- Noted that types that are pointed to, like `NSString **`, are not bridged.
- Added the [Key-Value Observing](#) section to the [Adopting Cocoa Design Patterns](#) chapter.
- Added the [Requiring Dynamic Dispatch](#) section to the [Interacting with Objective-C APIs](#) chapter.
- The `@IBOutlet` attribute no longer implicitly declares a property as optional and weak.
- Noted that the compiler does not implicitly synthesize the `@objc` attribute for private methods and properties.

2014-
08-04

- Updated the [Pointers](#) section to reflect that `CMutablePointer` and `CMutableVoidPointer` have been replaced with `UnsafePointer`, and `CConstPointer` and `CConstVoidPointer` have been replaced with `ConstUnsafePointer`.
- Noted that C function pointers are now imported as `CFunctionPointer`.
- Updated the [Swift and Objective-C in the Same Project](#) chapter with information about access control in mixed-language targets.
- Changed code listings to use the new `Array` and `Dictionary` type syntax.
- Added information about bridging between `Dictionary` and `NSDictionary`.

- Updated the Core Data section to include directions to configure the managed object model.

Copyright and Notices

Apple Inc.

Copyright © 2016 Apple Inc.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Cocoa, Cocoa Touch, Finder, Mac, Numbers, Objective-C, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

Launchpad, Swift, and tvOS are trademarks of Apple Inc.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED “AS IS,” AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the

possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.