Programming iOS 14 Using

SWIFT UI

Get Started With Building iOS 14 With Swift 5 and Xcode

Gary Elmer

Programming iOS 14
Using Swift UI

Get Started With Building iOS 14 With Swift 5 and Xcode

Gary Elmer

Copyright

Table of Contents

CHAPTER ONE

WELCOME TO SWIFTUI

Apple announced the Swift programming language at its 2014 Worldwide Developers Conference (WWCD). Before Swift, Apple used the Objective – C programming language to develop apps for all of its devices (iOS, tvOS, watchOS, maOS and the iPadOS). Apple uses SwiftUI, as a substitute to Objective-C, as an innovative means of designing responsive user interfaces across all of their platforms. Although Swift can be used on many platforms, it is especially a handy program of choice for building applications on Apple devices – ranging from devices that run on WatchOS, MacOS, tvOS and the most popular OS that runs iPhone (iOS). Swift UI understands what a device user doesn't understand. SwiftUI understands what every button on your device is working for and how it can be tapped, and it also entails how device users' enter texts on their devices.

For developers, developing an app is the act of writing Swift codes to control SwiftUI. Swift is a language that says "I want a text field here, a button over there and an image at this place" while the SwiftUI is the exact part of the whole design that actually knows how draw that text at the exact place, hot to make that button and exactly how show that image at the right place.

In summary, Swift is an exciting programming language that you can use to build apps for Apple Watch, iPhone, iPad, Mac, Apple TV and lots more.

This guide will take you on an expensive tour to building the **latest Apple's iOS 14 applications** using the **SwiftUI, Xcode 12 (which is the latest Xcode) and the Swift 5 Programming language (Swift 5.3).**

Before we go in depth into the real deal of designing iOS 14 with Swift, let us take a brief look at the Swift 5.3 programming language as it is an updated Swift language compared to the 4.2 released in 2017.

THE SWIFT 5.3 PROGRAMMING LANGUAGE

The Swift 5.3 Basic

Why Swift?

Swift, Apple's newest programming language, has been designed for iOS, watchOS, tvOS and macOS app development. Most readers who have used Objective-C programming language before will find Swift language very familiar.

Swift has its own types of all of the basic C and Objective-C types; ranging from Int for integers, Bool for Boolean values, String for textual data and the Double and Float for floating-point values.

Like the C programming language, Swift deploys variables to refer to values and store them. Swift, like most programming languages, makes good use of variables that don't change values. **Constants** are variables with unchanging values. Swift's constants are actually more useful and powerful than the constant in C language. Swift utilizes constants all through to make codes used in the development of apps clearer and safer in usage and intent.

Along with some familiar types common to both Swift and many popular languages in programming, Swift now introduces advanced types called **tuples** which was not obtainable in Objective C programming language (Apple uses the objective C language before Swift in 2014). With Tuples, you will be able to build and use values that have been grouped. A tuple, normally, can be utilized to return many values from any function as one compound value.

Swift also brings, with its many features, an optional type that help to handle the lack of a value. The optionals imply either "there is presence of a value which is equal to X" or "no value exists". The optionals in the Swift language are much like the nil with pointers for the Objective-C programming, only that the optional is applicable for any type, and not only for classes. Unlike the nil pointers in the Objective-C, the Swift's optionals are much safer and expressive and they are at the interface of many of Swift's most useful features.

Swift can be regarded as a *type-safe* programming language; this implies that the language gives you a clear shot of the categories of values that can actually work with your code. Let us say some areas in your code need a String data type, safety will not allow you to mistakenly give it an Int data type. In the same vein, the type safety capability in Swift prevents users from

mistakenly giving an optional String data type to a particular piece of code needing a non-optional String. With the type safety in Swift, it becomes especially very easy to see and fix errors in codes during the development phase once they are spotted.

Variables and Constants .

These can be regarded as the beauty of Swift. In your app development, you can assign a particular name like welcomeMessage or highestNumberOfLoginAttempts , and put a value or you can put a prompt with a value of your choosing (like the "Hello" string or the value 10). This actually implies that the number 10 is the maximum attempts that can be made by a user. This will serve as a security measure to prevent unauthorized access to that app. Also, the welcome message when the app is launched will be “Hello.” Once you set a value for your constant, you will not be able to change it anytime later. On the contrary, you can always set another value for your variable at any point in time.

How to declare Variables and Constants

Variables and constants cannot be used without necessarily declaring them in your codes. Constants are declared by the let keyword while the var keyword is used essentially to declare your variables. The below is a typical example of how your Variables and Constants can be deployed to monitor the exact value of attempted login by a particular user;

```
- let maximumNumberOfLoginAttempts = 10
- var currentLoginAttempt = 0
```

The above code actually means;

“Declare a constant named maximumNumberOfLoginAttempts, and assign it a number 10. Then, state a variable tagged currentLoginAttempt, and assign it 0 as the initial value.”

In the above example, the amount of login attempts allowable has been stated as the constant, since the maximum value does not change. The current login attempt counter has been stated as a variable, since the value will actually be increased after each login attempt that fails.

Commas can be utilized to declare multiple variables or constants on a single line. You only have to list the variables and then separate them with a comma.

```
1. var x = 0.0, y = 0.0, z = 0.0
```

Note: always use the let keyword to declare store values that won't change in the code and the values that can change should be mentioned as a variable.

Type Annotations

You can assign a *type annotation* once you have declared your variables or constants, if you wish to have a clear understanding of the categories of values that can be stored by the variable or constant. Enter a type annotation by putting a colon right after the name of the variable or constant, insert a space and then put the name of the type that you wish to use. Look at the below example for the type annotation for the variable named welcomeMessage, which clearly indicate that only String values can be stored by the variable:

```
1. var welcomeMessage: String
```

The colon meaning of the colon in the declaration above is "…of type…," and the above code can be read as:

"Let a variable called welcomeMessage of the String type be declared."

The "of type String" phrase actually denotes "can store any String value."

You can then proceed to set the variable welcomeMessage to any string value without any error:

```
1. welcomeMessage = "Hello"
```

Multiple related values having the same type can be declared on one line that is separated by commas, and then input a single type annotation just right after the variable's name;

```
var red, green, blue: Double
```

It is actually rare in practice that you will be required to write type

annotations. When you input any initial value for a particular variable or constant at the point where it has been defined, Swift will try to infer the data type that can be deployed for that particular variable or constant. If you look at the welcomeMessage above, you will see that no initial value has been given, hence the welcomeMessage variable's type has been specified with a type annotation instead of being inferred.

How to name Constants and Variables

Variables and Constants names are not choosy when it comes to characters designation as they can actually contain any character type (with some few exceptions), even the Unicode characters:

```
1. let π = 3.14159
```

Whitespace characters, private-use Unicode scalar values, arrows, mathematical symbols or the line- and box-drawing characters should not be utilized as characters. Characters cannot, as well, start with a number, but the number can be included somewhere else within the name.

Once a variable or constant of a certain type has been declared, you will not be able to declare such variable or constant again with the same name; neither can you even change it to a store value of another type. Also, you will not be able to convert a variable into a constant and vice versa.

If you plan to assign a variable or a constant the exact same name as a reserved Swift keyword, you will need to add a backticks (`) to the keyword. But it is actually advisable that you don't use keywords as names unless there is a need to use them.

Also, you will be able to change an existing variable's value to another value that is compatible. In the case below; the value of ``friendlyWelcome" has been changed from "Hello!" to "Greetings!":

```
1. var friendlyWelcome = "Hello!"
2. friendlyWelcome = "Greetings!"
3. // friendlyWelcome is now "Greetings!"
```

Once the actual value of a constant has been set, you will not be able to

change its value unlike that of a variable. If you make an attempt to modify the value of a particular constant in your code, you will get an error prompt when the code is compiled.

```
1. let languageName = "Java"
2. languageName = "Java++"
3. // This is a compile-time error: languageName cannot be changed.
```

How to print Constants and Variables

You can print the current value of a particular variable or constant can be printed by using print(_:separator:terminator:).

```
1. print(friendlyWelcome)
2. // Prints "Greetings!"
```

The print(_:separator:terminator:) function is described as a global function that is able to print one or more values to a particular output. While working with Xcode (the Integrated Development Environment for developing apps for Apple devices), for instance, the print(_:separator:terminator:) function will print output in the Xcode's "console" pane. The terminator and separator parameters actually have default values, hence omitting them when you call the function is acceptable. By default, the function will add a line break to terminate the line it prints. If you plan to print a certain value without necessarily having the line break after it, simply use an empty string as terminator—for instance, print(someValue, terminator: "").

Swift deploys *string interpolation* to add the name of a particular variable or constant as a placeholder in a longer string, and to ask Swift to substitute it with the current value of that variable or constant. Use parentheses to wrap the name and then escape the name with a backslash just before the opening parenthesis:

```
1. print("The current value of friendlyWelcome is \(friendlyWelcome)")
2. // Prints "The current value of friendlyWelcome is Greetings!"
```

Comments

Comments are used in a line of codes to include texts that are not executable;

probably as a reminder or a note to ask the users to carry out some actions. The Swift compiler will disregard comments during code compilation.

The Swift's comments and the Comments in the C programming language are very similar. Two forward slashes are used to initiate single line comments.

```
1. // This is a comment.
```

Use the forward slash together with an asterisk (/*) to begin a multiline comment and then end it with an asterisk and a forward-slash (*/):

```
1. /* This is also a comment
2. but is written over multiple lines. */
```

Unlike the multiline comments in the C programming language, you can actually nest a multiline comment in Swift inside another multiline comment. To write a nested comment, start a multiline block and then begin a second multiline comment within the first block. You can then close the second block and then follow it by the first block;

```
1. /* This is the start of the first multiline comment.
2. /* This is the second, nested multiline comment. */
3. This is the end of the first multiline comment. */
```

The Nested multiline comments allow users to comment out large areas of code fast and stress free, even if the code has the multiline comments already.

Semicolons

The Swift language, unlike most other languages, doesn't need users to input a semicolon (;) right after each statement in their code, although anybody can put a semicolon in Swift code if they wish. However, you will need to put semicolons if you plan to write multiple separate statements right on one line:

```
1. let dog = " ?? "; print(dog)
2. // Prints " ?? "
```

Integers

From basic mathematics, *integers* are whole numbers that have no fractional part, such as **57** and -45. Integers can either be **signed (+ve, 0,** or **–ve)** or **unsigned (+ve** or **0).**

With Swift, the signed and unsigned integers in 8 bits, 16 bits, 32 bits, and 64 bit forms are accounted for. Just like the way integers are named in the C language, Swift also follows the same pattern in that an 8-bit unsigned integer will have the **UInt8** type while a 32-bit signed integer will have the **Int32** type.

Integer Bounds

The **min** and **max** properties of integers can be used to check the minimum and maximum values of each integer type;

```
let minValue = UInt8.min  // minValue is equal to 0, and is of type UInt8
let maxValue = UInt8.max  // maxValue is equal to 255, and is of type UInt8
```

Each property has the appropriate-sized number type (like the **UInt8** in the above example) and can be deployed in expressions together with other values of equal type.

Int

In some cases, you don't necessarily have to select a specific integer size to be used in your code as Swift actually provides an extra integer type called **Int** with the same size as the native word size of the current platform;

- If you have a 32-bit platform, the **Int** will be of equal size as **Int32**.
- If you have a 64-bit platform, the **Int** will be of equal size as **Int64**.

You should always deploy **Int** for values of integers in your Swift code unless you are required to work with a particular integer size. This enables interoperability and consistency in your codes. On a 32-bit platform, **Int** can actually store values between -2,147,483,648 and 2,147,483,647, which are actually sufficient for many types of integers.

UInt

Swift also gives an extra integer type called **UInt** with the same size as the native word size of the current platform:

- If you have a 32-bit platform, **UInt** will be of equal size as **UInt32**.
- If you have a 64-bit platform, **UInt** will be of equal size as **U**Int64.

Using **Int** consistently in your code for integer value is better as it allows code interoperability and saves you the stress of converting from one number type to another. Hence, you should only use **UInt** if you actually require an unsigned type of integers that have the same size as that of the native word size of your platform.

Floating-Point Numbers

Floating-point numbers are values that have a fractional part, like 7.4998, 0.5, and -273.15.

The Floating-point types can have a much wider value than the integer type and can be deployed to store larger or smaller values that can be stored in an **Int.** In Swift, there are two forms of the floating-point number types;

- Double stands for a 64-bit floating-point number.
- Float stands for a 32-bit floating-point number.

Double has a precision of say 15 digits while the **Float** has about 6 decimals in precision. The nature and the range of values you are expected to use within your code will actually determine the floating-point type you can utilize. In a case where any of the two types of floating-point is suitable, kindly use the **Double** floating-point type.

Type Safety and Type Inference

The swift language is a typical *type-safe* language. A type safe language allows clarity with the type of values that can actually work with your Swift codes. If any part of your line needs a **String,** you cannot mistakenly substitute an **Int.**

The type-safe in Swift carries out type checks during code compilation and will prompt any wrong types as an error. This enables quick matching and fixing of errors during the compilation stage of your app development.

Type-checking also helps in avoiding errors while working with different kinds of values. This doesn’t mean that you will need to specify the kind of every variable and constant that you are using as Swift is capable of using the type inference to know the appropriate type you are working with. Type inference allows a compiler to automatically know the type of a certain expression during codes compilation just by checking the values you enter.

Due to the advantages provided by the type inference, the Swift language actually needs far lesser type declarations than most other programming languages like the C or Objective-C. Although, you will still have to enter the variables and constants but a large chunk of specifying the type you are working with will be done for you automatically by Swift.

Type inference is especially advantageous when you deploy an initial value to declare a variable or a constant. This is mostly carried out by assigning to the constant or variable a *literal value* (or *literal*) at the point where you are declaring your constant or variables. You can view a literal value as a value that shows in your source code, like 60 and 3.14159 in the below examples.

For instance, if you provide a literal value of 60 to a particular constant without indicating the type, the Swift will automatically pass an **Int** for the constant since you have begin it with a number that resembles an integer;

```
1. let meaningOfLife = 60
2. // meaningOfLife is assumed to be of type Int
```

Also, if no type has been indicated for a floating-point literal, Swift will automatically assumes that you mean to pass a **Double;**

```
1. let pi = 3.14159
2. // pi is inferred to be of type Double
```

Swift will always use Double (and not Float) when assuming the type of floating-point numbers.

If you have combined floating-point literals and integers in an expression, Swift will tend to assume a type of Double from the statement;

```
1. let anotherPi = 3 + 0.14159
2. // anotherPi is equally assumed to be of type Double
```

The literal value of 3 does not really have an explicit type in and of itself; hence an output type **Double** has been inferred owing to the floating-point literal in the addition.

The Swift Tuple

The Swift tuple is probably one of the simplest, yet most useful features defined in the Swift programming language. With a tuple, you can always - albeit temporarily - group multiple values together into one entity. The item you store in a tuple can be of any type since there are no binding rules that say the items must be of the same type. You can construct a tuple to have a Float value, an Int value and a string as you can see below;

let myTuple = (12, 452.433, "This is a String") Swift Data Types, Constants and Variables.

You can access a tuple quickly by referencing the index place (the first value being at index 0). For instance, the code below extract the string resource (found at index position 2) and then assign it to a another string variable;

```
let myTuple = (12, 452.433, "This is a String")
let myString = myTuple.2
print(myString)
```

Alternatively, you can extract all the values in a particular and then assign them to constants or variables in a single statement;

```
let (myFloat, myInt, myString) = myTuple
```

This same method can be deployed to extract some selected values from a tuple and then ignore others by substituting them with an underscore character. When you take a look at the code fragment below, you will see that the code extracts the string and integer values from the tuple and then assign them to variables but the floating-point value has been ignored;

```
var (myInt, _, myString) = myTuple
```

You can assign a name to each value when you are creating a tuple like the one below;

```
let myTuple = (count: 10, length: 432.433, message: "This is a String")
```

The values in the codes can then be referenced by the names given to the values stored in the tuple. For example, let us say that you want to get the result of the *message* string value from the myTuple, you can deploy the line of code below;

```
print(myTuple.message)
```

Tuple has an intrinsic ability to output multiple values from a function.

For example, a (404, "Not Found") tuple describes the status code of an *HTTP status.* An HTTP status code is actually a special value prompted by a web server (as an error) anytime you request a web page from the internet.

The error code is to tell you that the webpage does not exist.

```
1. let http404Error = (404, "Not Found")
2. // http404Error is of type (Int, String), and equals (404, "Not
   Found")
```

The (404, "Not Found") tuple actually group an Int and a String together to assign two different values to the HTTP status code: a number (404) together with a human-readable description. You can describe the (404, "Not Found") tuple as a tuple of type (Int, String)".

The Swift Optional Type

The Swift optional data type is a new data type that is essentially lacking in most other programming languages like the Objective-C programming language. The reason for the optional type is to afford a consistent and safe approach to take care of instances where a constant or a variable may not have any assigned value.

You will be able to declare any variable as an optional variable by putting a **question mark character (?)** right after the type declaration. The code below declares an Int variable (which is optional) named index;

```
var index: Int?
```

The variable named *index*, now, can have an integer value given to it or nothing at all. The compiler and the runtime will see "nothing" as nil (the nil value will be assigned by the compiler).

You can use an "**if statement"** to test whether an optional has an assigned value (Swift Data Types, Constants and Variables) or not.

```
var index: Int?

if index != nil {
    // index variable has a value assigned to it
} else {
    // index variable has no value assigned to it
}
```

If there is a value assigned to an optional, the value will be said to be

“wrapped” within that optional. You can use a method called forced unwrapping to access any value that has been wrapped in an optional. This means that the value will be extracted from the optional data type by putting an exclamation mark (!) in front of the optional name. Check the code below to understand this more;

```
var index: Int?
index = 3
var treeArray = ["Oak", "Pine", "Yew", "Birch"]
if index != nil {
print(treeArray[index!])
} else { print("index does not contain a value")
}
```

The code above simply deploys an optional variable to capture the index into an array of strings representing the identity of the tree. If the index optional variable possesses any assigned value, the name of the tree at that location inside the array will be printed to the console. Since the index is actually an optional type, its actual value has been unwrapped by putting an exclamation mark right after the name of the variable;

```
print(treeArray[index!])
```

Had it been that index has not been unwrapped (if the exclamation mark was omitted from the line above line), an error would have been prompted by the compiler as indicated below;

```
Value of optional type 'Int?' must be unwrapped to a value of type 'Int'
```

As an alternative to the forced unwrapping method, you can actually allocate the value you assigned to an optional to a temporary constant or variable using **optional binding** of the below syntax;

```
if let constantname = optionalName {
}
if var variablename = optionalName {
```

Error Handling in Swift

During code execution, your program may encounter some errors; these errors are usually handled by using **error handling.**

While **Optionals** can use the absence or presence of a value to tell the failure or success of a function, error handling, by contrast, enables you to know the

root cause of failure, and even allow you to propagate the error to another part of the program.

A function will **throw** an error upon seeing an error condition. The function's **catch** will be the one to catch the error and then give an appropriate response.

```
1. func canThrowAnError() throws {
2. // this function may or may not throw an error
3. }
```

A function can always indicate that it will be able to throw an error by inserting the throws keyword in the code declaration. While calling a function that can throw an error, you will have to prepend the keyword "try" in the expression.

Swift will automatically propagate an error out of its current scope until the error is handled by a catch clause.

```
1. do {
2. try canThrowAnError()
3. // no error was thrown
4. } catch {
5. // an error was thrown
6. }
```

A do statement will create a new scope, which will normally allow you to propagate errors to many other clauses.

The codes below show you how to use error handling to actively respond to different errors;

```
1. func makeASandwich() throws {
2. // ...
3. }

4.
5. do {
```

```
6. try makeASandwich()
7. eatASandwich()
8. } catch SandwichError.outOfCleanDishes {
9. washDishes()
10. } catch SandwichError.missingIngredients(let ingredients) {
11. buyGroceries(ingredients)
12. }
```

In the above case, the **makeASandwich()** function will throw an error once there are no clean dishes or if any particular ingredient is missing. Since the **makeASandwich()** can actually throw an error, the function call is wrapped in a try expression. When you wrap the call function in a do statement, any thrown errors are propagated to the catch clauses provided.

The **eatASandwich()** function is called if no error is thrown. But if an error is thrown and it tallies with the **SandwichError.outOfCleanDishes** instance, then the function **washDishes()** will be called. In another way, if an error is thrown and it tallies with the **SandwichError.missingIngredients** instance, then the function **buyGroceries(_:)** will be called with the associated [String] value captured by the catch pattern.

Assertions and Preconditions

Assertions and *preconditions* are routine checks that occur at runtime. They are primarily used to ensure that an important condition is met before any code is executed further. The code execution will normally continue if the Boolean condition in your assertion or precondition evaluates to true. But if the Boolean condition evaluates to false, the program's current state is not valid; code execution will end and the app will stop.

Assertions and preconditions are also used to express any assumptions made and expectations you have during coding to enable you integrate them as part of your code. Assertions will also help you to find mistakes and any incorrect assumptions during app development, and the preconditions will allow you to detect issues in production.

Assertions and preconditions not only allow you to confirm your expectations

at runtime, but they equally form a useful means of documentation within your code. You cannot use assertions and preconditions for recoverable errors or expected errors unlike the error handling. This is due to a fact that a failed assertion or precondition is indicative of an empty program state and it is not possible to catch an assertion that has already failed.

It is also worth noting that using assertions and preconditions is not a replacement for designing code in a way that some invalid conditions will not arise. However, it is very likely that your app keeps terminating more often (once an invalid state occurs) when you use assertions and preconditions to enforce valid state and data. It is always advisable to stop the execution as soon as you detect an invalid state. This helps to reduce the damage initiated by that particular invalid state.

One big difference between assertions and preconditions is at what point they are checked respectively: you check assertion in debug builds while preconditions are checked in both the debug and production builds. In the production builds, the condition in the assertion won't be evaluated. This literally means that you will be able to deploy as many as possible assertions during the app development process and it won't affect production performance.

Debugging with Assertions

You call the assert(_:_:file:line:) function from the Swift standard library when you want to write an assertion. You pass an expression that evaluates to true or false to the function and a special message to be prompted once the result of the condition is false. For instance:

```
1. let age = -3
2. assert(age >= 0, "A person's age can't be less than zero.")
3. // This assertion fails because -3 is not >= 0.
```

In the instance above, the code execution will only continue if age >= 0 amounts to true, meaning if the value of age is nonnegative. If the real value of age is actually negative, like the example above, then age >= 0 will be false, the assertion will fail and the app will be terminated.

If the condition has already been checked by the code, the assertionFailure(_:file:line:) function will be used to tell that an assertion has failed. For instance:

```
1. if age > 10 {
2. print("You can ride the roller-coaster or the ferris wheel.")
3. } else if age >= 0 {
4. print("You can ride the ferris wheel.")
5. } else {
6. assertionFailure("A person's age can't be less than zero.")
7. }
```

Enforcing Preconditions

Deploy a precondition anytime a condition has the chance to be false, but must *surely* be true if you want the code execution to continue. For instance, a precondition can be used to check whether a subscript is not out of bounds, or to check whether a valid value has been passed to the function.

A precondition can be written by calling the precondition(_:_:file:line:) function. You pass an expression that evaluates to true or false to the function and a special message to prompt once the result of the condition is false. For instance:

```
1. // In the implementation of a subscript...
2. precondition(index > 0, "Index must be greater than zero.")
```

The preconditionFailure(_:file:line:) function can also be called to show that a failure has occurred.

CHAPTER TWO

Basic Operators in Swift

An *operator*, in programming language, can be viewed as a special symbol that can be deployed to monitor, combine or change values. For instance, the **addition operator** (**+)** can be deployed to combine (add) two different or similar numbers together e.g **let i = 4 + 7,** while the **logical AND operator** (&&) can be used to combine two Boolean values, e.g if enteredPasswordtwice && locktheDevice.

Some of the common operators from familiar programming languages like Objective-C are also recognized by Swift programming language, though Swift offers improved capabilities to reduce or completely eliminate common errors in coding. To prevent the assignment operator (=) from being used mistakenly when the user means to use the (==) operator, the assignment operator (=) is not programmed to return any value. In order to prevent unintended results while working with digits that become smaller or larger than the value ranges allowable by the type that stores them, the arithmetic operators (+, -, *, /, % and so forth) usually detect and disable value overflow. You can access the value overflow behavior if you want by deploying the Swift's overflow operator.

Swift has range operators which are not obtainable in the C programming language, such as a..<b and a...b, and are often used as shortcut for specifying a range of values.

Terminology

We can have a unary, binary, or ternary operator:

- *Unary* operators work on a single target (like -a). Unary *prefix* operators are displayed right before their target (like !b), while the unary *postfix* operators always appear right after a target (like c!).
- *Binary* operators work on two targets (like 9 + 3) and they are actually *infix* since they usually appear in between their two targets.
- *Ternary* operators work on three targets. Like the C language, the Swift language has just one ternary operator which is the ternary conditional operator (a ? b : c).

The values that operators work on are called *operands*. In the expression 9 + 3, the + symbol (at the middle) is the binary operator and the two operands the + symbol is working on are the numbers 9 and 3.

Assignment Operator

When you want to update a value with another, you use the *assignment operator*. For instance, the assignment operator (a = b) will update the value of a with the actual value of b:

```
1.let x = 8
2.var y = 5
3.x = y
4.// x is now equal to 8
```

If the right part of the assignment contains a tuple with multiple values, the components can be broken into multiple variables or constants at once:

```
1.let (a, b) = (1, 2)
2.// a is equal to 1, and b is equal to 2
```

The assignment operator in Swift, unlike the one in C, does not return a value on its own. For instance, the statement below is not valid;

```
1.if a = b {
2.// This is not valid, because a = b does not return a value.
3.}
```

This feature will ordinarily prevent the assignment operator (=) from being utilized mistakenly when the user actually intends to use the equal to operator (==). Swift avoid mistakes like this by invalidate the "if x = y."

Arithmetic Operators

The Swift language supports all the four standard *arithmetic operators* for all available number types:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

```
1.7 + 2 // equals 10
2.8 - 3 // equals 5
3.4 * 3 // equals 12
4.8.0 / 2.0 // equals 4.0
```

The Swift arithmetic operator's values cannot overflow by default unlike the C and Objective-C arithmetic operator. You can have access to the value overflow behavior by using Swift's overflow operator (such as a &+ b).

The String concatenation also supports the addition operator;

```
1."hello, " + "world" // equals "hello, world"
```

Remainder Operator

For *remainder operator* (c % z), you can calculate how many multiples of z will fit inside c and the value remaining will be returned. This value is called the remainder. In some other programming languages, the remainder operator is called modulo operator.

To carry out 9 % 4, you need to find how many 4s you can find in 9. It is visible that you can fit two 4s inside 9 and you will have 1 as the remainder. This will be written in Swift language as;

```
1. 9 % 4 // equals 1
```

To know the answer for c % z, the equation below will be calculated by the % operator and prompt the remainder as its output:

c = (z x some multiplier) + remainder

The "some multiplier" component is actually the largest value of multiples of z that will fit inside c.

Putting 9 and 4 inside the equation above will yield:

9 = (4 x 2) + 1

The same approach is used when finding the remainder for a particular negative value of a:

```
1. -9 % 4 // equals -1
```

Putting -9 and 4 into the above equation will give:

-9 = (4 x -2) + -1

And you will have a remainder of -1.

The sign of z is ignored for negative numbers of z. This implies that c % z and c % -z always bring the same output.

Unary Minus Operator

You can deploy the prefix -, called the **unary minus operator,** to toggle the sign of a particular numeric value. See below;

```
1. let four = 4
2. let minusFour = -four // minusFour equals -4
3. let plusFour = -minusFour // plusFour equals 4, or "minus minus four"
```

The unary minus operator (-) is usually prepended right before the number it usually operates on, without inserting any white space.

Unary Plus Operator

The *unary plus operator* (+) will return the number or value it works on, without any change:

```
1. let minusFive = -5
2. let alsoMinusFive = +minusFive // alsoMinusFive equals -5
```

Although it can appear like the unary plus operator is not really doing anything; and of course it is not, in the actual sense, but you can deploy the Unary Plus operator to give symmetry in your Swift code for positive values while you use the unary minus operator for negative values.

Compound Assignment Operators

Like the C programming language, the Swift language features *compound assignment operators* which combine assignment operators (=) with another

operation. One case is the *addition assignment operator* (+=):

```
1.var a = 1
2.a += 2
3.// a is now equal to 3
```

The expression a += 2 is the shorthand for a = a + 2. By doing this, the addition and the assignment have been combined into a single operator that carries out both of the tasks at the same time.

Comparison Operators

The following comparison operators are supported with Swift;

- Equal to (x == y)
- Not equal to (x != y)
- Greater than (x > y)
- Less than (x < y)
- Greater than or equal to (a >= y)
- Less than or equal to (x <= y)

A Bool value is always returned by each of the comparison operator to tell whether the statement is true or not;

```
1.8 == 8 // true because 8 is equal to 8
2.5 != 1 // true because 5 is not equal to 1
3.5 > 1 // true because 5 is greater than 1
4.0 < 2 // true because 0 is less than 2
5.1 >= 1 // true because 1 is greater than or equal to 1
6.7 <= 1 // false because 7 is not less than or equal to 1
```

Comparison operators found application in conditional statements like the if statement:

```
1.let name = "John"
2.if name == "John" {
3.print("hello, John")
4.} else {
5.print("I'm sorry \(name), but I don't recognize you")
6.}
7.// Prints "hello, John", because name is indeed equal to "John".
```

Two Tuples can be compared with each other if they actually have the same number of values and are of the same type. You compare tuples from left to right, going from one value at a time, until the comparison is able to find two unequal values. The two unequal values will be compared and the result will actually determine the end result of the tuple comparison. If all of the elements/values are the same, then the tuples are equal. For instance;

```
1.(1, "apple") < (2, "zebra") // true because 1 is less than 2; "apple" and "zebra" are not
  compared
2.(3, "apple") < (4, "bird") // true because 3 is less than 4, and "apple" and "bird" are not
  compared.
3.(4, "dog") == (4, "dog") // true because 4 is equal to 4, and "dog" is equal to "dog"
```

In the above example, you can observe the left-to-right comparison that happened on the first line. Since 1 is actually less than 2, (1, "zebra") is taken to be less than (2, "apple"), and it doesn't matter what other values reside in the tuple. It doesn't even matter if "zebra" is not less than "apple", since the comparison has already been determined by the first element of the tuples. But in a case when you have the first element of the tuples to be the same, their second elements will be compared – you can see what happened in the third line.

You will only be able to compare tuples with a specific operator only if you can apply the operator to each value in the tuples. For instance, as you can see in the code shown below, you will be able to compare two tuples of (String, Int) type since you can compare both String and Int values with the < operator. In another case, you cannot compare two tuples of (String, Bool) type with the "less than <" operator since you cannot apply the < operator

to Bool values.

```
1.("blue", -1) < ("purple", 1) // OK, evaluates to true
2.("blue", false) < ("purple", true) // Error because < can't compare Boolean values
```

Note: You will only be able to compare tuples with fewer elements (up to six) as this is the limit obtainable in the Swift standard library. If you want to compare tuples that contain about seven or more elements, you will need to implement the comparison operator by yourself.

Ternary Conditional Operator

Special operator that has three parts is called the *ternary conditional operator* and it will usually take the form question ? answer1 : answer2. The ternary conditional operator provides a shortcut that can be used to evaluate one of two statements based on whether the question is false or true. If the question is true, the answer1 will be evaluated and its value will be returned. If the question is false, the answer2 will be evaluated and its value will be returned.

The ternary conditional operator provides shorthand for the below code;

```
1.if question {
2.answer1
3.} else {
4.answer2
5.}
```

The case below is one that calculates height for a particular row of table. The row height is 50 points taller than the content height provided that the row possess a header, and will be 20 points taller than the content height if the row lacks a header;

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

The above example is the shorthand for the code below:

```
let contentHeight = 40
let hasHeader = true
let rowHeight: Int
if hasHeader {
rowHeight = contentHeight + 50
} else {
rowHeight = contentHeight + 20
}
// rowHeight is equal to 90
```

In the first example, the way the ternary conditional operator was used means that the rowHeight can actually be set to the correct value on a single line of code, which is rather more concise than the one deployed in the second example.

It becomes very easy to decide which one out of the two expressions will be appropriate. The ternary conditional operator should be used with care, however. It should not be overused as its conciseness can create hard-to-read code. Also, do not combine multiple cases of the ternary conditional operator into a single compound statement.

Logical Operators

The Boolean logic values true and false can be combined or modified by *Logical operators*. Just like the C language, the three essential logical operators are also supported by Swift programming language. The standard operators include;

- Logical NOT (!a)
- Logical AND (a && b)

- Logical OR (a || b)

Logical NOT Operator

The *logical NOT operator* (!a) serves to invert a Boolean value to enable the true becomes false, and the false tending to true.

The logical NOT operator works like a prefix, and usually displays right before the value it is operating on, without any white space. The logical not operator (!a) can be read as “not a”, as you can see in the example below;

```
1.let allowedEntry = false
2.if !allowedEntry {
3.print("ACCESS DENIED")
4.}
5.// Prints "ACCESS DENIED"
```

The statement if !allowedEntry will be read as “if not allowed entry.” The line that follows will only be executed if “not allowed entry” is true; meaning that if allowedEntry is false.

As it was in the example above, you can make your code more readable and concise when you choose your Boolean constant and variable names carefully, while you avoid double confusing logic statements or negatives.

Logical AND Operator

For the *logical AND operator* (a && b), a logical expression is created where all of the values must be true for the overall expression to equally be true.

If any of the values is false, the whole expression is automatically false. As a matter of fact, if the value of the first statement is false, the value of the second statement won’t even be evaluated, since it can no longer make the whole expression equal to true. This is called *short-circuit evaluation*.

The examples below examine two Bool values and access is only allowed if both of the values are true:

```
1.let enteredDoorCode = true
2.let passedRetinaScan = false
3.if enteredDoorCode && passedRetinaScan {
4.print("Welcome!")
5.} else {
6.print("ACCESS DENIED")
7.}
8.// Prints "ACCESS DENIED"
```

Logical OR Operator

The *logical OR operator* (a || b) is just an infix operator designed from two adjacent pipe characters. The Logical OR operator can be utilized to make logical statements where only one of the two values must evaluate to true if the whole expression must be true.

Just like you can observe in the above Logical AND operator, the short-circuit evaluation is used by the Logical OR operator to consider its expressions. If the left part of a Logical OR expression is true, the right part will not be evaluated, since the outcome of the whole expression cannot be changed.

In the below example, the (hasDoorKey) which is the first Bool value is false, but the (knowsOverridePassword) which is the second value is true. Since one value is true, the whole expression will evaluate to true, and access will be allowed:

```
1.let hasDoorKey = false
2.let knowsOverridePassword = true
3.if hasDoorKey || knowsOverridePassword {
4.print("Welcome!")
5.} else {
6.print("ACCESS DENIED")
7.}
8.// Prints "Welcome!"
```

Combining Logical Operators

Longer compound expressions can be created by combining multiple logical operators;

```
1.if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
2.print("Welcome!")
3.} else {
4.print("ACCESS DENIED")
5.}
6.// Prints "Welcome!"
```

The above example deploys multiple && and || operators to make a longer compound expression. However, the && and || operators still work on only two values, so there are three smaller expressions chained together. The above example can thus be read as:

If the correct door code has been entered and it passed the retina scan, or if there is a valid door key, or if the emergency override password is known, then allow access.

From the values of enteredDoorCode, hasDoorKey and passedRetinaScan, the first two subexpressions are false. The whole compound expression, however, still amounts to true since the emergency override password is known.

The Swift logical operator && and || are actually left-associative, this means that compound expressions that have multiple logical operators will evaluate the leftmost subexpression.

Explicit Parentheses

In order to allow a complex statement to be easy to follow and read, it is often useful to insert parentheses when they are not strictly needed. You can make the intention of the door access case described above explicit by putting parentheses in the first part of the compound statement.

```
1.if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
2.print("Welcome!")
3.} else {
4.print("ACCESS DENIED")
5.}
6.// Prints "Welcome!"
```

The parentheses make it obvious that the first two values are taken as part of a separate possible state in the overall logic. The result of the compound statement does not change, but the whole intent of the statement is clear to the reader. Readability over brevity; parentheses provide clear intention of your statement.

Strings and Characters

A string is a chain of characters, like the "hello, world" or "albatross". String types are used to represent Swift strings. There are various ways of accessing the contents of a String.

Working with texts in your code with a fast and Unicode-compliant way is made possible by the Swift's String and Character types. The syntax for creating and manipulating string is readable and lightweight, with a string literal syntax similar to the one in C. String concatenation can be done with the + operator, and managing string mutability is possible by selecting between a variable and a constant, just like most other values in Swift. Strings can also be used to add variables, constants, expression and literals into longer strings. This process is known as **string interpolation.** With this, creating custom value for storage, display and printing becomes easy.

Despite this syntax's simplicity, Swift's String type provides a fast, modern string implementation. Every string is made up of encoding-independent Unicode characters, and gives support for getting those characters in various Unicode representations.

String Literals

Predefined String values can be included within your code as *string literals*. Sequences of characters flanked by double quotation marks (") are called string literals.

Deploy a string literal as a base value for a variable or constant:

```
1. let someString = "Some string literal value"
```

You can see that Swift infers a type of String for the someString constant stipulated in the above example because it is initialized with a string literal value.

Multiline String Literals

A multiline string literal is a string that covers many lines.

```
1.let quotation = """
2.The Red goat wears his spectacles. "Where shall I start,
3.Allow me your Majesty?" he begged.
4.
5."Start at the beginning," the Queen said gravely, "and go on
6.till you reach the end; then stop."
7."""
```

A multiline string literal has all of the lines between the opening and closing quotation marks. The string starts on the first line just after the opening quotation marks (""") and terminates on the line just before the closing quotation marks, which implies that neither of the strings indicated below begin or stop with a line break:

```
1  let singleLineString = "These are the same."
2  let multilineString = """
3  These are the same.
4  """
```

When a source code has a line break in a multiline string literal, the line break will equally appear in the string's value. If you plan to deploy line breaks to make your source code clear and easy to read, but you do not plan to have the line breaks to be part of the string's value, just put a backslash (\) at the end of those lines:

```
let softWrappedQuotation = """
The Red goat wears his spectacles. "Where shall I start,
Allow me your Majesty?" he begged.

"Start at the beginning," the Queen said gravely, "and go on
till you reach the end; then stop."
"""
```

To have a multiline string literal that starts or terminates with a line feed, put a blank line as your first or last line. For instance:

```
let lineBreaks = """

This string starts with a line break.
It also ends with a line break.

"""
```

Special Characters in String Literals

The following special characters are often included in string literals;

- The escaped special characters \0 (null character), \t (horizontal tab), \\ (backslash), \n (line feed), \" (double quotation mark), \r (carriage return) and \' (single quotation mark)
- An arbitrary Unicode scalar value, compiled as \u{*n*}, where *n* is actually a 1–8 digit hexadecimal number.

The below code shows four cases of these special characters. The wiseWords constant has two escaped double quotation marks. The dollarSign, blackHeart, and sparklingHeart constants show the Unicode scalar format:

```
1.let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
2.// "Imagination is more important than knowledge" - Einstein
3.let dollarSign = "\u{24}" // $, Unicode scalar U+0024
4.let blackHeart = "\u{2665}" // ♥Unicode scalar U+2665
5.let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
```

Because multiline string literals deploy three double quotation marks instead of one, you can put a double quotation mark (") inside a multiline string literal without necessarily escaping it. You will be able to put the text """ in a multiline string by escaping at least one of the quotation marks. For instance:

```
1.let threeDoubleQuotationMarks = """
2.Escaping the first quotation mark \"""
3.Escaping all three quotation marks \"\"\"
4."""
```

Initializing an Empty String

An empty String value can be created to be the starting point for creating a longer string by either assigning an empty string literal to a particular variable, or starting a new String case with initializer syntax:

```
1  var emptyString = ""               // empty string literal
2  var anotherEmptyString = String()  // initializer syntax
3  // these two strings are both empty, and are equivalent to each other
```

You can know whether a String value is actually empty by checking the string's Boolean isEmpty property:

```
1. if emptyString.isEmpty {
2. print("Nothing is here to display")
3. }
4. // Prints "Nothing is here to display"
```

String Mutability

You can indicate if a particular String can be mutated or modified by

assigning the string to a variable (where you will be able to modify it), or assigning it to a constant (where modification is not possible):

```
var variableString = "Dog"
variableString += " and cat"
// variableString is now "Dog and cat"

let constantString = "favorite"
constantString += " and another favorite"
// this reports a compile-time error - a constant string cannot be modified
```

Working with Characters

You can use the for-in loop while iterating over a string to get the individual Character values for the String.

```
for character in "Dog! 🐶 " {
print(character)
}
// D
// o
// g
// !
// 🐶
```

Alternatively, a stand-alone Character variable or constant can be created from one character string literal when you provide a Character type annotation:

```
let exclamationMark: Character = "!"
```

String values can be built when you pass an array of Character values as an argument to the string's initializer:

```
let catCharacters: [Character] = ["C", "a", "t", "!", " 🐶 "]
let catString = String(catCharacters)
```

```
3. print(catString)
4. // Prints "Cat!"
```

Concatenating Strings and Characters

String values can be concatenated (added together) by using the addition operator (+) to form a new String value:

```
1. let string1 = "hi"
2. let string2 = " John"
3. var welcome = string1 + string2
4. // welcome is now "hi John"
```

You can equally append another String value to a String variable that is already existing by using the addition assignment operator (+=):

```
1. var instruction = "Watch over"
2. instruction += string2
3. // instruction now equals "Watch over John"
```

String Interpolation

String interpolation is a method that can be used to create a new String value from a mix of variables, constants, expressions and literals by putting their respective values inside a string literal. String interpolation can be deployed in both the single-line and multiline string literals. Each object inserted into the string literal will be wrapped inside a pair of parentheses and then a backlash (\) is inserted before the object:

```
1. let multiplier = 4
2. let message = "\(multiplier) times 2.0 is \(Double(multiplier) * 2.0)"
3. // message is "4 times 2.0 is 8.0"
```

In the case above, the content of multiplier is put into a string literal as \(multiplier). This placeholder was then substituted with the exact value of multiplier when the string interpolation is solved to make an actual string.

The value of multiplier is equally part of a larger statement later in the string. This statement calculates the value of Double(multiplier) * 2.0 and puts the result (8.0) inside the string. In this instance, the statement is written as \

(Double(multiplier) * 2.0) when it is added inside the string literal.

You can use string interpolation inside a string that deploys extended delimiters by matching the number of number signs immediately after the backslash to the number of number signs just at the start and end of the string. For instance:

```
1. print(#"6 times 7 is \#(6 * 7)."#)
2. // Prints "6 times 7 is 42."
```

The expressions inside the parentheses within an interpolated string cannot have an unescaped backslash (\), a line feed or a carriage return; but they can have other string literals.

Unicode

Unicode is an approved standard for processing, representing and encoding text in different writing systems. It allows users to process and represent, in a standardized form, about almost any character from any language, and to also read and write those characters to and from an external source like the web page or a text file. The String and Character types used in Swift are essentially Unicode-compliant.

Unicode Scalar Values

Looking at what is happening behind the scene; Unicode scalar values are used to build Swift's native String type. A Unicode scalar value is actually a special 21-bit number for a modifier or character, such as U+1F425 for FRONT-FACING BABY CHICK or U+0061 for LATIN SMALL LETTER A ("a").

It is not all of the 21-bit Unicode scalar values that are designated to a particular character—some scalars are actually reserved for future use in the UTF-16 encoding. A scalar value that has been assigned to a character will also have a name, just like the LATIN SMALL LETTER A and FRONT-FACING BABY CHICK in the above example.

Counting Characters

The count property of a string can be used to recover a count of the Character values in the string.

```
1. let unusualMenagerie = "Koala, Snail, Penguin, Dromedary"
2. print("unusualMenagerie has \(unusualMenagerie.count) characters")
3. // Prints "unusualMenagerie has 40 characters"
```

String's modification and concatenation may not necessarily affect the character count of a string due to the Swift's usage of extended grapheme clusters for Character values.

For instance, if you prompt a new string with a particular four-character word like cafe, and then insert a COMBINING ACUTE ACCENT (U+0301) at the end of the string, the string's result will still have 4 as the character count, but the fourth character will be é and not e:

```
1. var word = "cafe"
2. print("the number of characters in \(word) is \(word.count)")
3. // Prints "the number of characters in cafe is 4"


4.
5. word += "\u{301}" // COMBINING ACUTE ACCENT, U+0301


6.
7. print("the number of characters in \(word) is \(word.count)")
8. // Prints "the number of characters in café is 4"
```

Accessing and Modifying a String

A string can be accessed or modified through the string's properties and methods or even by deploying subscript syntax.

String Indices

Each String value possesses an associated *index type*, String.Index, which actually corresponds to the location of each Character in the string.

As discussed above, different characters can demand different levels of memory to store, so if you want to know which Character is located at a particular position, you need to iterate over each Unicode scalar from the beginning or from the end of that String. For this exact reason, you cannot index Swift strings by integer values.

The startIndex property can be used to access the exact position of the first Character of a String. The endIndex property is the place just after the last character in a particular String. Hence, the endIndex property is really not a valid argument to the subscript of a string. The startIndex and endIndex will be equal when a String is empty

You will be able to access the indices just before and just after a specific index by deploying the index(before:) and index(after:) methods of String. If you plan to access an index that is farther away from the specific index, you can deploy the index(_:offsetBy:) method rather than calling one of these methods many times.

The subscript syntax can be used to access the Character at a specific String index.

```
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
// G
greeting[greeting.index(before: greeting.endIndex)]
// !
greeting[greeting.index(after: greeting.startIndex)]
// u
let index = greeting.index(greeting.startIndex, offsetBy: 7)
greeting[index]
// a
```

You will receive a runtime error if you try to access an index outside of a specific string’s range or to just access Character at an index outside of a string’s range.

```
for index in greeting.indices {
    print("\(greeting[index]) ", terminator: "")
}
// Prints "G u t e n   T a g ! "
```

You can also use the indices property to view all the indices of individual characters in a string.

```
1.for index in greeting.indices {
2.print("\(greeting[index]) ", terminator: "")
3.}
4.// Prints "G o o d m o r n I n g ! "
```

Inserting and Removing

To put a single character into a string at a specific index, you can deploy the insert (:at:) method, and you can also insert the contents of another string at a specified index by using the insert(contentsOf:at:) method.

```
var welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
// welcome now equals "hello!"

welcome.insert(contentsOf: " there", at: welcome.index(before:
  welcome.endIndex))
// welcome now equals "hello there!"
```

You can remove a single character from a string at a specific index by using the remove(at:) method, and you can also remove a substring at a specific range by using the removeSubrange(_:) method:

1. welcome.remove(at: welcome.index(before: welcome.endIndex))
2. // welcome now equals "hello there"

3.

4. let range = welcome.index(welcome.endIndex, offsetBy: -6)..

```
<welcome.endIndex
5. welcome.removeSubrange(range)
6. // welcome is now "hello"
```

CHAPTER THREE

Classes and Objects in Swift

Just like you see it in languages like C++, Objective-C, Java and few other languages, you will be able to define templates for your objects by using **Classes.** In Swift, Classes take the format

```
class Vehicle {
}
```

Classes feature both methods and properties. Variables that are core part of a class are called **properties** while functions that are essential part of a class are called **methods.**

In the example below, the Vehicle class has two properties: **Color** which is an **optional**
String, and **maxSpeed** which is an **Int**. The way Property is declared is just about the same as the way variables are declared in other codes

```
class Vehicle {
var color: String?
var maxSpeed = 80
}
```

Methods in a particular class look exactly the same as functions just anywhere else. Code that resides in a method can explore the properties of a class by using the keyword **self**, which talks about the object that is running the code currently:

```
class Vehicle {

    var color: String?
    var maxSpeed = 80

    func description() -> String {
        return "A \(self.color) vehicle"
    }

    func travel() {
        print("Traveling at \(maxSpeed) kph")
    }
}
```

The **self** keyword can be omitted if it is visible that the property is actually part of the current object. In the example above, you can see that **description** deploys the keyword **self**, while travel does not. When a class has been defined, instances of the class (called an object) can be created which you can work with.

For instance, if you want to define an instance of the Vehicle class, you will define a variable and call the initializer of the class. Once you have done that, it becomes very easy to work with the class's properties and functions;

```
var redVehicle = Vehicle()
redVehicle.color = "Red"
redVehicle.maxSpeed = 90
redVehicle.travel() // prints "Traveling at 90 kph"
redVehicle.description() // = "A Red vehicle"
```

Initialization and Deinitialization

A distinct method called **initializer** is called whenever an object is created in the Swift language. The method that you deployed to set up the object's initial state is the initializer and it is usually named **init.**

There are two types of initializers in Swift; *designated initializers* and *convenience initializers*. A
designated initializer helps you set up all the things required to use an object; default settings are often used where necessary. A convenience initializer, as the name implies, allows convenient setting up of the instances by allowing more details in the initialization process. As part of its setup, a convenience initializer needs to call the designated initializer.
In addition to initializers, codes can be run when removing an object using a method called *deinitializer*, named **deinit**. This runs just when the object's retain count has dropped to zero and it is always called before removing the object from memory. This is the final chance for your object to carry out any necessary cleanup before going away permanently.

```
class InitAndDeinitExample {
    // Designated (i.e., main) initializer
    init () {
        print("I've been created!")
    }
    // Convenience initializer, required to call the
    // designated initializer (above)
    convenience init (text: String) {
        self.init() // this is mandatory
        print("I was called with the convenience initializer!")
    }
    // Deinitializer
    deinit {
        print("I'm going away!")
    }
}

var example : InitAndDeinitExample?

// using the designated initializer
example = InitAndDeinitExample() // prints "I've been created!"
example = nil // prints "I'm going away"

// using the convenience initializer
example = InitAndDeinitExample(text: "Hello")
// prints "I've been created!" and then
//  "I was called with the convenience initializer"
```

Nil can also be returned by an initializer. This can be found applicable when the initializer is not able to construct an object. For instance, the NSURL class has an initializer that takes a string and converts such string into a URL; if the string is not a valid URL, the initializer will return nil. If you want to create an initializer that will be able to return nil—also known as a *failable initializer*— you need to put a question mark right after the **init** keyword, and nil will be returned if the initializer decides that it cannot successfully construct the object:

```
// This is a convenience initializer that can sometimes fail, returning nil
// Note the ? after the word 'init'
convenience init? (value: Int) {
    self.init()

    if value > 5 {
        // We can't initialize this object; return nil to indicate failure
        return nil
    }

}
```

When a failable initializer is used, an optional will always be returned;

```
var failableExample = InitAndDeinitExample(value: 6)
// = nil
```

Properties

Data in Classes are stored in *properties*. Properties, as discussed before, are constants or variables that are attached to instances of classes. The code snippet below shows how you can access properties that you have added to class;

```
class Counter {
    var number: Int = 0
}
let myCounter = Counter()
myCounter.number = 2
```

However, as objects become more complex, there can be a problem in the system. If you wanted to use engines to represent vehicles, you would need to add a property to the Vehicle class; however, this would mean that all Vehicle instances would have this property, even if they never need one. To make things better organized, it is actually better to move properties that are specific to a subset of your objects to a new class that *inherits* properties from another.

Inheritance

When you define a class, you can create one that *inherits* from another. When a class inherits from another (called the *parent* class), it incorporates all of its parent's functions and properties. In Swift, classes are allowed to have only a

single parent class. This is the same as Objective-C, but differs from C++, which allows classes to have multiple parents (known as *multiple inheritance*). If you plan to create a class that inherits from another class, you will need to put the name of the class (the one you are inheriting from) right after the name of the class you are creating, like the example below:

```
class Car: Vehicle {

    var engineType : String = "V8"

}
```

Classes that inherit from other classes can usually *override* functions in their parent class. This depicts that you can create subclasses that will inherit most of their functionality, but can specialize in certain areas. For instance, the Car class features an **engineType** property; this property will only be featured by only Car instances. To override a function, you will need to re-declare the function in your subclass and then add the override keyword to let the compiler know that you are not creating a method accidentally with the same name as the one in the parent class. In an overridden function, it is usually very useful to call back to the parent class's version of that function. You can do this through the **super** keyword, which lets you get access to the superclass functions:

```
class Car: Vehicle {

    var engineType : String = "V8"

    // Inherited classes can override functions
    override func description() -> String  {
        let description = super.description()
        return description + ", which is a car"
    }

}
```

Observers

When you are working with properties, you may normally want to run some code whenever there is a change in property. To support this, Swift will actually let you add *observers* to your properties. Observers are small chunks of code that can run just before or after the value of a property. To create a

property observer, simply add braces right after your property and add willSet and didSet blocks. These blocks each get passed a parameter—willSet, which is called before the property's value changes, is given the value that is about to be set, and didSet is given the old value:

```
class PropertyObserverExample {
    var number : Int = 0 {
        willSet(newNumber) {
            print("About to change to \(newNumber)")
        }
        didSet(oldNumber) {
            print("Just changed from \(oldNumber) to \(self.number)!")
        }
    }
}
```

Protocols

A *protocol* can be imagined as a list of requirements for a particular class. When you define a
protocol, you are creating a list of properties and methods that can be declared by classes. A protocol seems very much like a class, except that you don't need to provide any actual code—you just define what kinds of properties and functions exist and how they can be accessed. For instance, if you plan to have a protocol that describes any object that can blink on and off, you could use this:

```
protocol Blinking {

    // This property must be (at least) gettable
    var isBlinking : Bool { get }

    // This property must be gettable and settable
    var blinkSpeed: Double { get set }

    // This function must exist, but what it does is up to the implementor
    func startBlinking(blinkSpeed: Double) -> Void
}
```

Once a protocol has been created, you can then create classes that *conform* to a protocol. When a class conforms to a protocol, it is effectively promising to the compiler that it implements all of the properties and methods listed in that protocol. It is also allowed to have more than one protocol. To proceed with this example, you can create a particular class called Light that uses the

Blinking protocol. Remember, the only job of a protocol is specifying *what* a class can do—the class itself is the one that is actually responsible for determining *how* it does it:

```
class TrafficLight : Blinking {
    var isBlinking: Bool = false

    var blinkSpeed : Double = 0.0

    func startBlinking(blinkSpeed : Double) {
        print("I am a traffic light, and I am now blinking")
        isBlinking = true

        // We say "self.blinkSpeed" here, as opposed to "blinkSpeed",
        // to help the compiler tell the difference between the
        // parameter 'blinkSpeed' and the property


            self.blinkSpeed = blinkSpeed
    }
}

class Lighthouse : Blinking {
    var isBlinking: Bool = false

    var blinkSpeed : Double = 0.0

    func startBlinking(blinkSpeed : Double) {
        print("I am a lighthouse, and I am now blinking")
        isBlinking = true

        self.blinkSpeed = blinkSpeed
    }
}
```

One advantage of using protocols is that Swift's type system can be used to refer to any object that conforms to a given protocol. This is good because you have the chance to specify that you only care about whether an object conforms to the protocol—the specific type of the class doesn't matter since we are using the protocol as a type:

```
var aBlinkingThing : Blinking
// can be ANY object that has the Blinking protocol

aBlinkingThing = TrafficLight()

aBlinkingThing.startBlinking(4.0) // prints "I am now blinking"
aBlinkingThing.blinkSpeed // = 4.0

aBlinkingThing = Lighthouse()
```

Extensions

In Swift, you can *extend* existing types and add further methods and computed properties. This is very useful in two situations:

• You are working with a type written by another person, and you plan to add functionality to it but either you don't have access to its source code or you just don't want to mess around with it.

• You are working with a type that you wrote, and you want to improve its readability by dividing up its functionality into different sections.

With Extensions, you can carry out both of these options easily. In Swift, you can extend *any* type—that is, you can extend both classes that you write, as well as built-in types like Int and String. You will be able to create an extension by using the extension keyword and then follow it by the name of the type you want to extend. For instance, to add methods and properties to the built-in Int type, you can do this:

```
extension Int {
    var doubled : Int {
        return self * 2
    }
    func multiplyWith(anotherNumber: Int) -> Int {
        return self * anotherNumber
    }
}
```

Extension can also be used to make a type conform to a protocol. For instance, you can make the Int type conform to the Blinking protocol described earlier:

```
extension Int : Blinking {
    var isBlinking : Bool {
        return false;
    }

    var blinkSpeed : Double {
        get {
            return 0.0;
        }
        set {
            // Do nothing
        }
    }

    func startBlinking(blinkSpeed : Double) {
        print("I am the integer \(self). I do not blink.")
    }
}
2.isBlinking // = false
2.startBlinking(2.0) // prints "I am the integer 2. I do not blink."
```

Access Control

There are three categories of access control recognized by Swift and they all determine the kind of information that will be accessible to different part of the application;

- *Public:* Any part of the app can access public classes, properties and methods. For instance, all of the classes in the UIKit that you use to build iOS apps are actually public.
- *Internal:* Internal entities (data and methods) will only be accessible to the *module* in which they are only defined. A module is just like an application, library, or framework. This is the reason you cannot access the inner workings of UIKit—it is defined as internal to the UIKit framework. Internal access control is actually the default level of access control: meaning that if you fail to specify the access control level, it will be assumed to be internal.
- *Private:* Private entities are only accessible to the file in which it is declared. This means that you can actually create classes that hide their inner workings from some other classes that are in the same module, which helps to keep the amount of surface area that those classes expose to each other to a minimum.

The kind of access control that a method or property can have depends on the access level of the class that it is contained in. You cannot make a method more accessible than the class in which it is contained. For instance, you can't define a private class that has a public method:

```
public class AccessControl {

}
```

All methods and properties are essentially internal by default. You can explicitly define a member as internal if you want, but it isn't necessary:

```
// Accessible to this module only
// 'internal' here is the default and can be omitted
internal var internalProperty = 123
```

The exception is for classes defined as private—if you don't declare an access control level for a member, it will be set as private, not internal. It is *impossible* to specify an access level for a member of an entity that is more open than the entity itself.
When you declare a method or property as public, it becomes visible to everyone in your app:

```
// Accessible to everyone
public var publicProperty = 123
```

If you declare a method or property as private, it is only accessible from within the source file in which it is declared:

```
// Only accessible in this source file
private var privateProperty = 123
```

Operator Overloading

An operator is actually a function that takes one or two values and then returns a value.
Operators can be overloaded just like any other functions. For instance, you can represent
the + function like this:

```
func + (left: Int, right: Int) -> Int {
    return left + right
}
```

Swift allows users to define new operators and overload existing ones for their new types, which implies that if users have a new type of data, they can operate on that data using both existing operators, as well as new ones which they invent by themselves. For instance, imagine you have an object called Vector2D, which stores two floating point numbers:

```
class Vector2D {
    var x : Float = 0.0
    var y : Float = 0.0

    init (x : Float, y: Float) {
        self.x = x
        self.y = y
    }
}
```

Generics

Swift is a pure statically typed language. This implies that the Swift compiler will need to definitively comprehend what type of information your code is actually dealing with. This means that you cannot pass a string to code that expects to deal with a date (which is something that can happen in Objective-C!). However, this rigidity robes users of some flexibility. It is actually annoying to have to write a chunk of code that does some work with strings, and another that works with dates. This is where *generics* are actually applicable. Generics enable you to write code that does not really need to know precisely *what* information it is exactly dealing with. An example of this kind of use is in arrays: they don't actually do any work with the data they store, but instead just store it in an ordered collection. Arrays are, in fact, generics. To make a generic type, you will name your object as usual, and then specify any generic types between angle brackets. T is the traditional term used, but you can put anything you like. For instance, to create a generic Tree object, which has a value and any number of child Tree objects, you would carry out the following;

```
class Tree <T> {

    // 'T' can now be used as a type inside this class

    // 'value' is of type T
    var value : T

    // 'children' is an array of Tree objects that have
    // the same type as this one
    private (set) var children : [Tree <T>] = []

    // We can initialize this object with a value of type T
    init(value : T) {
        self.value = value
    }
```

CHAPTER FOUR

Introduction to Swift's Functions

Functions are defined as self-contained pieces of code that carry out a certain task. You assign a function a particular name that shows what it does, and the name will be used to "call" the function to carry out its assigned task when needed.

Swift's unified function syntax is actually flexible enough to convey anything from a very simple C-style function (without any names of parameters) to a rather difficult Objective-C-style method (that has arguments and names labels for each parameter).

Every function in the Swift language is associated with a type, which is made up of the function's return types and parameter types. This type can be used just like any other type in Swift, making it especially very easy to pass one function to another as parameters, and to also return functions from functions. You can also write a function inside another function to be able to capture

important functionality within a particular nested function scope.

Defining and Calling Functions

By defining a function, you will be able to optionally define one or more typed, named values taken by the function as input (also known as parameters). It is also possible to optionally define the type of value that will be passed back as the output by the function when it is completed; this is known as return type.

Every function has a *function name* describing the tasks that the function is performing. To use a particular function, you will call the function with the name and pass it arguments (input value) that correspond to the type of the function's parameter list.

The function in the case below is called **greet(person:)**, since that is the action it is performing – it inputs the name of a person and returns a greeting for the person. To get around this, you will define one input parameter – a string value called **person –** and a return string called that will contain the greetings for the person;

```
1. func greet(person: String) -> String {
2. let greeting = "Hello," + person + "!"
3. return greeting
4. }
```

All of the information is rolled up into the definition of the function, which is normally started (prefixed) with **func** as the keyword. The *return arrow* -> (a hyphen with a right angle bracket) is used to indicate the return type of the function.

The definition explains what the function is doing, what the function should receive, and what the function will return when it is completed. The definition makes it especially very easy to call the function ambiguously from anywhere in the code;

```
1. print(greet(person: "Anna"))
2. // Prints "Hello, Anna!"
3. print(greet(person: "Brian"))
4. // Prints "Hello, Brian!"
```

The **greet(person**:) function is called by passing the function a String value right after the person argument label, such as **greet(person: "Anna")**. Since the function actually returns a String value, you can wrap the **greet(person:)** in a call to the **print(_:separator:terminator:)** function to be able to print that string and examine its return value, as displayed above.

The body of the **greet(person:)** function begins by defining a particular new String constant named **greeting** and setting the constant to a simple greeting message. The **return** keyword is then used to pass the greetings out the function. When you check the line that says return greeting, you will see that the function ended its execution and the current value of **greetings** was returned.

The **greet(person:)** function can be called multiple times, and each time with different input values. The above case examines what happens if it is called with an input value of "Anna", and then an input value of "Brian". A tailored greeting was returned by the function in each of the cases.

The return message and the message creation can be combined into a single line to make the body of the function shorter;

```
1. func greetAgain(person: String) -> String {
2. return "Hello again, " + person + "!"
3. }
4. print(greetAgain(person: "Anna"))
5. // Prints "Hello again, Anna!"
```

Function Parameters and Return Values

Return values and Function parameters are very flexible in the Swift programming language. Anything can be defined right from a very simple utility function that has a single unnamed parameter to a complex function with parameter names that are expressive and different parameter options.

Functions without Parameters

You don’t need functions to define input parameters. The case below is a function without any input parameter which always output the same message anytime it is called;

```
1. func sayHelloWorld() -> String {
2. return "hello, world"
3. }
4. print(sayHelloWorld())
5. // Prints "hello, world"
```

The function definition still requires parentheses right after the name of the function, even though it does not take parameters. The name of the function is equally followed by an empty pair of parentheses when it is called.

Functions with Multiple Parameters

Functions with multiple input parameters are written inside the parentheses of the function and then separated by commas.

The name of a particular person is taken by the function and determines whether the person has been greeted (input) and then returns a greeting for the person;

```
1. func greet(person: String, alreadyGreeted: Bool) -> String {
2. if alreadyGreeted {
3. return greetAgain(person: person)
4. } else {
5. return greet(person: person)
6. }
7. }
8. print(greet(person: "Tim", alreadyGreeted: true))
9. // Prints "Hello again, Tim!"
```

The **greet(person:alreadyGreeted**:) function can be called by passing it both a String argument value tagged person and also a **Bool** argument value tagged **alreadyGreeted** in parentheses, separated by commas. Note that this function is different from the **greet(person:)** function you saw earlier. Although both of the functions contain names that start with **greet**, the **greet(person:alreadyGreeted:)** function actually takes two arguments while the **greet(person:)** function takes just one.

Function Argument Labels and Names of

Parameter .

A parameter name and label for an argument are associated with each function's parameter. You use the argument label when you want to call a function with each argument put in the call function together with its argument label before it. The function's implementation is done with the name of the parameter. Parameter, by default, uses their parameter name as the argument label.

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

All parameters should be represented with unique names. Although, multiple parameters can have the same argument label, but a unique argument label will affect your code's readability.

Specifying Argument Labels

An argument label is often expressed before the name of the parameter, separated by a space.

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

The example below is another variant of the greet(person:) function that takes the name and hometown of a person and then outputs a greeting;

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

When you use an argument label in your function, your function can be called in a sentence-like and expressive manner, while still giving a body of function that is readable and clear in purpose.

Omitting Argument Labels

In case you do not plan to have an argument label for a particular parameter, simply put an underscore (_) in place of an argument label for the particular parameter.

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(1, secondParameterName: 2)
```

If there is an argument label for a parameter, you must label the argument when you call the function.

Default Parameter Values

You will be able to define a *default value* for a particular parameter in a function when you assign a value to that parameter after the parameter's type. If you have defined a default value, the parameter can be omitted when you are calling the function.

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int =
  12) {
    // If you omit the second argument when calling this function, then
    // the value of parameterWithDefault is 12 inside the function body.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) //
  parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault is 12
```

Parameters that don't have default values should be placed at the start of a function's parameter list, just before those parameters with default values. The parameters without default values are often more important to the meaning of the function—it becomes easier to notice that the function is being called when you write them first and it doesn't matter whether any

parameter has been omitted or not.

Variadic Parameters

These are parameters that take 0 or more values of a specific type. The variadic parameter is used to indicate that you can pass a varying number of input values to the parameter when the function is called. The variadic parameters can be written by putting three period characters (...) just after the type name of the parameter.

The values you passed to a variadic parameter will be made available as an array of the appropriate type within the function's body. For instance, a variadic parameter with numbers as a name and Double... as the type will be made available within the function's body as a constant array named numbers of type [Double].

The case below finds the *arithmetic mean* (otherwise called the *average*) for a list of numbers of any length:

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

Note that it is possible for a function to have at most one variadic parameter.

In-Out Parameters

By default, function parameters are always constants. You will often get a compile-time error when you try to change a function's value from within the function's body. What this means is that it is not possible to change the values for parameters mistakenly as you will always get an error. If you want to modify a parameter's value with a function, and you want to let the changes persist even after the function call has finished, you just define the

parameter as an **in-out** parameter instead.

An in-out parameter can be written when you place the inout keyword just before a parameter's type. An in-out parameter usually has a value that is passed *in* to the function, the value is modified by the function, and it is then passed back *out* of the function to substitute the original value.

Only a variable can be passed as the argument for an in-out parameter. A literal value or a constant value cannot be passed as the argument since it will be difficult to modify literals and constants. While passing a variable as an argument to an in-out parameter, you will put an ampersand (&) right before the name of the variable. The ampersand will show that it can be modified by the function. It is not possible for in-out parameters to have default values and you cannot also mark a variadic parameter as an in-out.

The case below is a case of a function called swapTwoInts(_:_:), with two in-out integer parameters named a and b:

```
1  func swapTwoInts(_ a: inout Int, _ b: inout Int) {
2      let temporaryA = a
3      a = b
4      b = temporaryA
5  }
```

The swapTwoInts(_:_:) function is there to swap the value of b into a, and that of a into b. The function is able to perform this swap by storing the actual value of a in a temporary constant named temporaryA, assign the value of b to a, and then assign temporaryA to b.

The swapTwoInts(_:_:) function can be called with two variables of type Int in order to swap their values. Note that there is an ampersand before the names of someInt and anotherInt while passing them to the swapTwoInts(_:_:) function:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The example above depict that the original values of someInt and anotherInt have been modified by the swapTwoInts(_:_:) function, even though they were initially defined outside of the function.

Function Types

Every function is associated with a specific *function type*, consisting of the return type and the parameter types of the function.

For instance:

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

Two simple mathematical functions named addTwoInts and multiplyTwoInts were defined by the above example. Each of these functions take two Int values, and then return an Int value, which is actually the yield of carrying out an appropriate mathematical operation.

The type of these two functions is (Int, Int) -> Int. read as:

“A function with two parameters, both of type Int, and that returns a value of type Int.”

The case below is another example, for a function with no parameters or return value:

```
1. func printHelloWorld() {
2. print("hello, world")
```

```
3. }
```

The type of this function is actually () -> Void, or you can call it “a function with no parameters, and returns Void.”

Using Function Types

Function types are used just like most other types in Swift. For instance, you can define your variable or constant to be a part of the function type and then assign a suitable function to the variable;

```
1. var mathFunction: (Int, Int) -> Int = addTwoInts
```

You can read the above code as;

“Let a variable called mathFunction be defined with a type of ‘a function that accepts two Int values, and returns an Int value.’ Set this new variable to refer to the function called addTwoInts.”

This kind of assignment is allowed by the Swift’s type-checker since the addTwoInts(_:_:) function actually has the same type as that of the mathFunction variable.

The assigned function can now be called with the name **mathFunction:**

```
1. print("Result: \(mathFunction(2, 3))")
2. // Prints "Result: 5"
```

You can even assign a different function that has the same matching type to the same variable, in the same way you do for nonfunction types.

```
1. mathFunction = multiplyTwoInts
2. print("Result: \(mathFunction(2, 3))")
3. // Prints "Result: 6"
```

As it is with many other types, you can let Swift infer the function type on its own anytime you assign a function to a variable or constant;

```
1. let anotherMathFunction = addTwoInts
2. // anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

Function Types as Parameter Types

Function type like (Int, Int) -> Int can be used as a parameter type for another function. This allows you to enable the function's caller to provide some aspects of a function's implementation when the function is called.

The below case is an example that prints the result of the math function above;

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8"
```

The above case defines a function called **printMathResult(_:_:_:),** with three parameters. The first parameter is named **mathFunction**, and has type **(Int, Int) -> Int**. Any function of that type can be passed as an argument for the first parameter. The 2nd and 3rd parameters are named **a** and **b**, and both are of the **Int** type. These two parameters will be used as the two input values for the math function.

The **printMathResult(_:_:_:)** will be passed the **addTwoInts(_:_:)** function when it is called, followed by the integer values **3** and **5**. The provided function is called with the values 5 and 3 and 8 is printed as the result.

The function of **printMathResult(_:_:_:)** is to print a call's result to a math function of a suitable type. It doesn't even matter what that function's implementation is actually doing—what matters is that the function should be of the right type. This lets **printMathResult(_:_:_:)** to hand off some of its functionality to the function's caller in a type-safe way.

Closures

Another useful feature of the Swift language is that of *closures.* A closure is a small, anonymous piece of code that can be used like functions. Closures are good for passing to other functions in order to tell them how they should perform a particular task. To give you an overview of how closures work, consider the built-in sort function. This function takes an array and a closure, and deploys that closure to see how two individual elements of that array should be ordered (i.e., which one should go first in the array):

```
var sortingInline = [2, 5, 98, 2, 13]
sortingInline.sort() // = [2, 2, 5, 13, 98]
```

If you want to sort an array so that small numbers will go before big numbers, a closure can be provided detailing how that can be done like the one below;

```
var numbers = [2, 1, 56,32,120,13]
// Sort so that small numbers go before large numbers

var numbersSorted = numbers.sort({
(n1: Int, n2: Int) -> Bool in return n2 > n1
})
// = [1, 2, 13, 32, 56, 120]
```

Closures have a special keyword, **in**. The **in** keyword enables Swift to know where to break up the closure from its definition and its implementation. So in the previous example, the definition was (n1: Int, n2: Int)->Bool, and the implementation of that closure came after the **in** keyword: return n2 > n1.
A closure, just like a function, takes parameters. In the preceding example, the closure clarifies the type and name of the parameters that it works with. However, you don't need to be quite so verbose—the compiler can infer the type of the parameters for you, much like it can with variables. Notice how types are obviously absent in the parameters for the following closure:

```
var numbersSortedReverse = numbers.sort({n1, n2 in return n1 > n2})
// = [120, 56, 32, 13, 2, 1]
```

You can make it even terser, if you don't really care what names the parameters should have. If you omit the parameter names, you can just refer to each parameter by number (the first parameter is called $0, the second is called $1, etc.). Additionally, the **return** keyword can be omitted if your closure only contains a single line of code:

```
var numbersSortedAgain = numbers.sort({
    $1 > $0
}) // = [1, 2, 13, 32, 56, 120]
```

Lastly, if the last parameter in a function call is a closure, it can be placed just outside the parentheses. However, this is just a way to improve code readability and does not necessarily change how the closure works.

```
var numbersSortedReversedAgain = numbers.sort {
    $0 > $1
} // = [120, 56, 32, 13, 2, 1]
```

The defer Keyword

Sometimes, you may be planning to execute some codes but at a later time or schedule. For instance, if you are writing code that opens a file and carries out some changes, you will also need to make sure that the file is closed when you are done. This is important, and it is easy to forget when you start writing your method. The defer keyword allows you to write code that will run at a later time. Specifically, code you put in a defer block will run when the current flow of execution leaves the current scope—that is, the current function, loop body, and so on:

```
func doSomeWork() {
    print("Getting started!")
    defer {
        print("All done!")
    }
    print("Getting to work!")
}

doSomeWork()
// Prints "Getting started!", "Getting to work!", and "All done!", in that order
```

CHAPTER FIVE

GETTING READY ON THE ROUTE TO DEVELOPING iOS 14 BASED APPS

The Apple Developer program

One of the first things you must understand on your way to learning the

essentials of iOS 14 based applications is deciding whether it becomes essential for you to join the **Apple Developer program**. There are many benefits you stand to get when you join the Apple Developer program as a paid membership. There are two ways of enrolling in the Apple Developer program; **individual membership** and the **organizational** or **company membership.** The **individual membership** is when you paid the membership to join the program by yourself while the **organizational membership** is when your company has a paid membership already and you are only expected to join through your company's link. Today, membership into the Developer program for Apple as an individual costs 99 USD per year.

Prior to the on-boarding of the iOS 9 in 2015 and the Xcode 7, one of the key benefits of the Apple Developer program was that you can test your iOS based apps on physical iOS devices with the creation of provisioning profiles and certificates once you joined the program. However, this seemed too tedious and demanding at that time. Fortunately today, the only requirement you need to have at your disposal if you plan to join the Apple Developer program is your Apple ID.

Of course there are things you can actually get done on the platform without paying the membership fee, but there exists some app development stuff that you won't be able to access without your paid membership. Features like the Apple Pay, Game center, In-App purchasing and access to iCloud are only possible with the paid membership. This is why it is advisable to endure and pay the 99 USD membership fee to enjoy premium packages from the Developer program.

The paid **Apple Developer Program** gives you, as an engineer, access to some beta development tools, distribution ability through Apple's App Stores and beta operating system releases. It also lets you use some of the cloud-dependent features of the developer platforms, such as iCloud, CloudKit, Maps, In-App Purchase and App Groups. A lot of the cloud-dependent features, like the iCloud will be used more often in this guide. You will not be able to run these apps if you have not subscribed to the paid membership yet.

The following are the benefits of a paid membership;

- You will have unrestricted access to the Apple Developer Forums. The Developer forum is often frequented by engineers from Apple itself and you are allowed to ask any question bothering you. The forum also has the presence of developers like you who are there to learn, attempt to ask questions and rub minds together. Although, the forum can still be accessed with your Apple ID.
- You will also have unrestricted access to beta versions of any OS before they are released into the market for public use. This way, you can easily test your applications on the next iOS version, iPadOS, tvOS and WatchOS so that you will be able to actually know if there is any editing that you can do to your app to make the app compatible with the latest OS. You will also have access to the beta versions of the development tools.
- You will not be able to submit or publish your apps to the App store for sale without a paid Developer membership. What this means is that membership will be needed at some point when you want to publish your apps to the App store for download by users. Even you cannot release your app for free on the App store without a paid membership.

That said, registering for the Developer Program is not necessary if you only plan to view the documentation or you just want to download the current version of the developer tools, so you can play around with writing apps without opening your wallet to pay.

The next thing, then, is how to get registered on the Apple Developer program.

Enrolling in the Apple Developer Program

To enroll in the Apple developer, you will need the following;

- **As an individual**
    - Apple ID with the two factor authentication enabled
    - Basic information about you, including your address and legal name.
    - Your credit card details.
- **As an organization**

    - Apple ID with the two factor authentication enabled
    - A **D-U-N-S number.** The D-U-N-S number is used by Apple to confirm your company's legal entity status and identity. It is a unique 9 digits obtained from Dun & Bradstreet used widely as a business identifier for companies. If your company does not have a D-U-N-S number, you can request one for your company. Visit https://developer.apple.com/support/D-U-N-S/ for how to check if your company already has a D-U-N-S number, otherwise you can register by providing your details.
    - Legal entity status. Apple will not accept a fictitious business name. Your business must be registered as a legal entity.
    - Legal binding authority: The person registering for the Apple Developer program on behalf of the company must have legal authority to bind the company with Apple's legal agreements.
    - Your organization must have a working website.

- Once all the requirements have been met, you can proceed by visiting https://developer.apple.com/programs/enroll/, scroll down the page and tap on **"Start your enrolment."**
- You will be taken to a page where you are expected to input your Apple ID. Enter your Apple ID and tap on the arrow to proceed.

Follow all the instructions that will be prompted on the subsequent page to register as an individual or to register for your organization.
Note: If your plan is to build iOS applications for your company, then you need to check, first, whether your company already has a paid membership. To do this, kindly confirm from your program admin and ask him/her to invite you into the Developer program from the member center of the Company's Developer program. Once the admin has done this, an email will be forwarded to you from Apple titled "*You Have Been Invited to Join an Apple Developer Program.*" The email will contain the link you will follow to activate your membership.

It won't take up to 24 hours before you are finally accepted into the Apple developer program as a solo member. You will receive the notification from Apple in the form of an activation mail. If it is your company you are enrolling for, acceptance can take weeks or months to be activated because of the extra verification requirements involved.
While you are awaiting activation email from Apple, you can log into the Member Center, albeit with a restricted access, using your Apple ID and password at the URL below;
https://developer.apple.com/membercenter. Once you have logged in, tap on the "*your Account*" tab located at the top part of the screen to see the active status of your enrollment. Once you received the activation email, simply log in to the member center and you will be able to see that you now have access

to a wide range of resources.

CHAPTER SIX

THE Xcode 12 and the iOS 14 SDK

Installing Xcode 12 and the iOS 14 SDK

You cannot build an iOS app without an iOS SDK and Apple's Xcode Development environment. You will be wondering, at this time, about what Xcode is. Xcode is Apple's integrated Development environment where you will have access to code, build, test and debug your iOS apps. The Swift language is used to code iOS applications and the basics have been discussed in the earlier chapters. The Xcode provides an IDE where you will get to use the Swift language to compile and run codes for your iOS applications. Xcode actually contains tools for developers to manage their whole development workflows, till the point where they can submit such apps to the App store.

This section will talk about how you can install the latest Xcode 12 and the iOS 14 SDK, and also delve into their basic features.

You can download and install the Xcode 12 by following the steps below;

- Navigate to the Apple Developer page with the link https://idmsa.apple.com/IDMSWebAuth/signin?appIdKey=891bd3417a7776362562d2197f89480a8547b108fd934911b and then sign in with your Apple ID and Password.
- Upon successful login, you will be taken to a page where you can download the new Xcode 12. The Xcode file size is usually heavy, so you must ensure that you have enough space on your Mac computer.
- Tap on the blue **“Download”** button.
- Upon successful download, double click on the file to start the “Un-zipping” process.
- After unzipping, you can install the Xcode 12 on your MacOS by dragging the file from the “downloads” folder and then inside “Application” folder on your MacOS.
- To open the app, scroll to the “Application” folder on your computer

and tap twice on the app icon. You can even launch the app from your Mac launch pad.

- Once you have successfully launched the app, you will be greeted with the Xcode 12 welcome page.

Note: The iOS 14 SDK is downloaded from the Apple Developer page.

Creating a project with Xcode 12

Once you receive the Welcome page, simply tap on the "**Create a new Xcode project"** to start creating a fresh Xcode project. A fresh Xcode project will enable you to develop apps for iOS, watchOS, tvOS and iPadOS. When you initiate a project, all the files and resources to be deployed to develop your Apple apps will be organized and you can quickly and conveniently use them wherever you want. There are available templates for each OS that you want to develop, ranging from iOS, tvOS, iPadOS etc. If you wish to see the interactive preview of what you are working on, select Swift as the programming language and Swift UI as the programming interface. Once a project has been created, you will be able to access the main Xcode window. It is from the window that you will be able to access, edit and manage all of your projects.

Follow the steps to create;

1. Click twice on your Xcode icon to launch Xcode, and from the "Welcome to Xcode" window, tap on the "Create a new Xcode project." Alternatively, select **File**, click on "**New"** and tap on "Project."

2. You will be prompted with a window where you can choose the target operating system.

3. Choose a template under Application and click on Next.

For instance, if you plan to build an iOS app that has a single empty view, choose Single View App. If you want to create macOS apps, choose App. To create a watchOS app that can run without a companion iOS app, choose Watch App.

4. Again, you will be prompted with another window where you are expected to input some details in the text field and then configure

your project by choosing options from the pop-up menu.

You must input a name for your product (you can let the product name be **Hello Swift)** and the organization identifier (a DNS string that can identify your organization. In case your organization does not have, simply use com.example and then follow this by your organization's name) and the name of the product you are building. You will also input the name of your organization (fill this with your name in case you are not working for an organization). You will see a bundle identifier (which is obtained from your product's name and your organization identifier) right below the organization identifier box.

5. You will be prompted with a team pop-up menu where you are expected to select your team.

You can add an account if the "Add account" button appears. This step can be skipped since you can actually add a team to your project at a later time.

6. Select your programming language from the language pop-up menu. Here, you will select **Swift** as the programming language.
7. Choose an interface from the User Interface pop-up menu. You will select Swift UI here.
8. Tap on the "Include Unit Tests" and "Include UI Tests boxes" to add a test target to your project.
9. Tap Next, and you will get a window asking you where you want to save your work.
10. Choose a location for your project, optionally choose "Create Git repository on my Mac" to deploy source control (recommended) and then tap on "**Create."**

The Xcode 12 Main Window (The Xcode 12 interface)

The Xcode 12 main window is actually an interface where you get to see, edit

and manage all of the essential parts of your project. As discussed, Xcode displays your entire project in a single window, which is divided into a number of sections. You can open and close each section at will, depending on what you wish to see.

Let us take a look at each of these sections and examine what they do.

The editor

The Xcode editor is the place where you are going to spend a lot of time since most of the coding is done there. The editor is where all source code editing, project configuration and interface design are carried out. The editor often changes depending on which file you are working on. If you are editing your source code, the editor will be a text editor, with code completion, syntax highlighting, and all the usual features that developers have come to expect from an integrated development environment. If you are modifying a user interface, your editor will be a visual editor, which will enable you to drag around the major part of your interface. Each file you opened has their special editor. When you initiate your first project, the editor will begin by displaying the project settings. You can divide the editor into an assistant editor and a main editor. The assistant editor displays files that are related to the file you are working on in the main editor. It will continue to show files that have a relationship to whatever is open, even if you open different files. For instance, if you open an interface file and then open the assistant, the assistant will, by default, show related code for the interface you are editing. If you launch another interface file, the assistant will display the code for the newly opened files.

At the top of your editor, you will see the *jump bar*. The jump bar allows you to quickly jump from the content that you are editing to another piece of related content, such as a file in the same folder. The jump bar is a fast way to navigate your project.

The toolbar

The Xcode toolbar serves as mission control for your entire interface. The Xcode is the only part of your Xcode that will not change significantly as you develop your applications, and it also functions as the place where you will get to control what your code is actually doing.

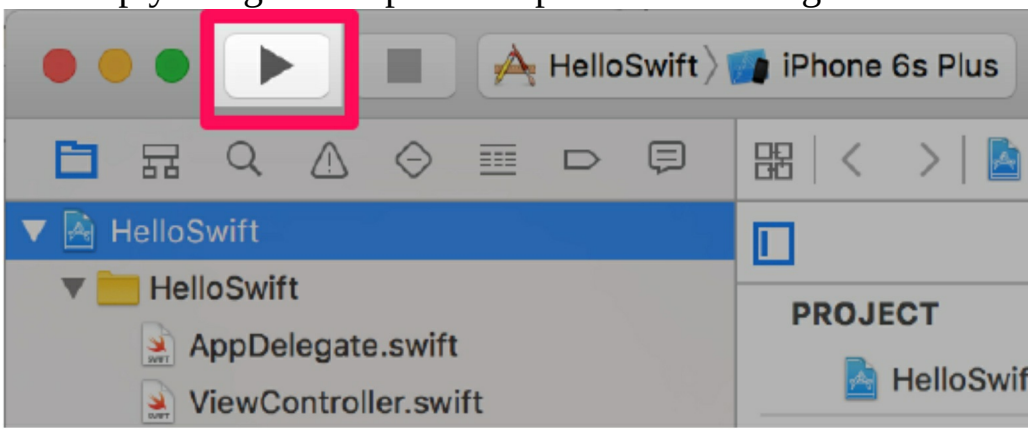
The Xcode Toolbar contains the following items;

Run button

When you tap on the run button, the Xcode will be instructed to compile and start your application. Depending on which application you are running and the settings you selected, the **run button** will have different actions;

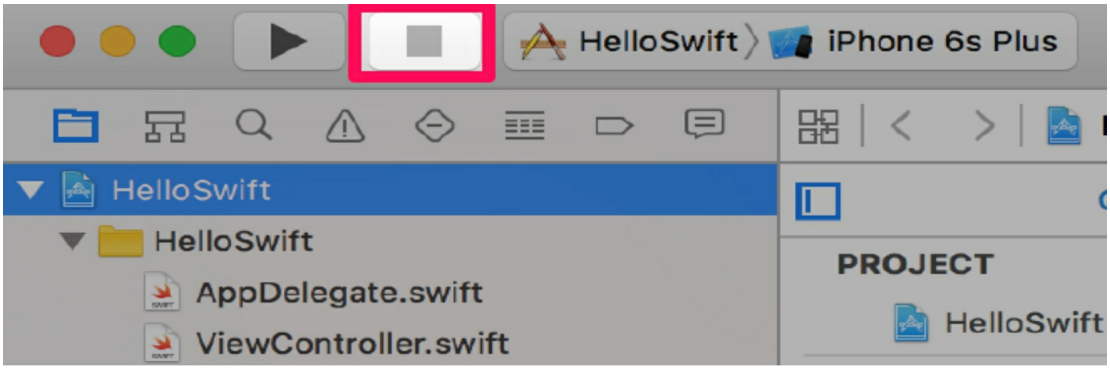
- If you are building a Mac application, the new app will show in the Dock and will run on your machine.
- If you are building an iOS application, the new app will open either on a connected device (like iPhone or iPad) or on the iOS simulator.

Additionally, if you tap and then hold on the **run button**, you can change it from **run** to some other actions like **Test, Analyze** or **profile.** The Test action will run any unit tests that you have initiated; the Profile action will run the application Instruments, while the Analyze action will examine your code and help you figure out potential problems and bugs.



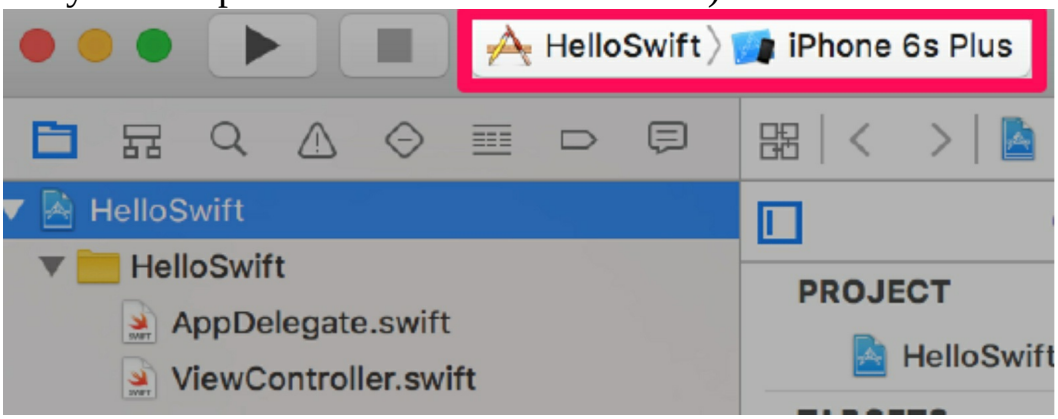
Stop button

Simply tap on the **Stop button** if you plan to terminate any task that is currently being run by the Xcode - if it is creating your application, it terminates; and if your application is running in the debugger, it stops it.



Scheme selector

In Xcode, *Schemes* are called build configurations—that is, what you want to build, how the product will be run on a device, and where it will run (will it be run on your computer or on a connected device).



Your projects can feature multiple apps inside them. Using the Scheme selector, you can choose which target or app that you plan to create. To select a target, tap on the left hand side of the scheme selector. You can equally choose where your application will run. If you are creating a Mac application, you might likely be planning to run the application on your Mac. If you are creating an iOS application, you, however, have the option of running the application on an iPad simulator or an iPhone simulator. (These are actually the same application; it simply changes shape depending on the selected scheme.) You can also decide to run the application on a connected iOS device if the device has been set up for development.

Status display

The status display displays what the Xcode is doing—creating your application, installing an application on an iOS device etc.

If you have more than one task in progress, you will see a small button on the left side which cycles through your active task when you tap it.

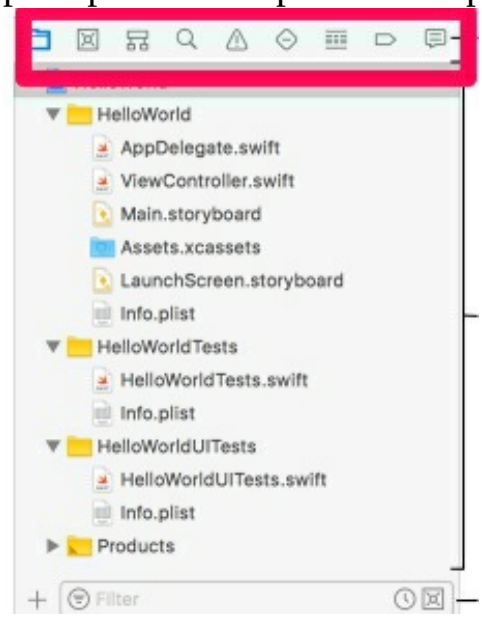
Editor selector

The editor selector is the one that will determine how the editor will be laid out. You can choose to show either a single editor, the version editor or the editor with the assistant, so that you can always compare different files' versions if you are deploying a revision control system such as Subversion or Git.

The navigator

The lefthand area of your Xcode window is the *navigator*, which presents information

about your project. Tapping on a specific button in the navigation bar will prompt different parts of the project inside the content area.



From left to right, the navigation section has nine (9) tabs which include;

Project navigator

The project navigator will list various files in your project. This navigator is actually the most common as it actually determines what will be displayed inside the editor. Whatever you choose in your project navigator will be what you will see in the editor area.

Source control navigator

The source control navigator helps to access your source control working copies, commits, branches, tags and remote repositories.

Symbol navigator

All of the symbols in your project can be accessed with the symbol navigator. It helps list all the functions and classes existing in your project. If you want the quick summary of a class or you just want to jump directly to a method in that class, you will find the symbol navigator handy.

Search navigator

With the Search navigator, you can always search across your project while looking for specific texts. (Simply use the ⌘ -Shift-F shortcut. Press ⌘ -F if you plan to search the current open document.)

Issue navigator

The issue navigator lists all the problems encountered by Xcode in your code. This can include compilation error, warnings, and issues spotted by the built-

in code analyzer.

Test navigator

The test navigator displays all the unit tests that are associated with a project. Unit tests used to be an optional component of Xcode but are now built into Xcode directly.

Debug navigator

The Debug navigator will become active during program debugging. You will be able to check the active state of the threads that constitute your program.

Breakpoint navigator

With this navigator, adding, editing and deleting breakpoints that you set while debugging your program becomes easier.

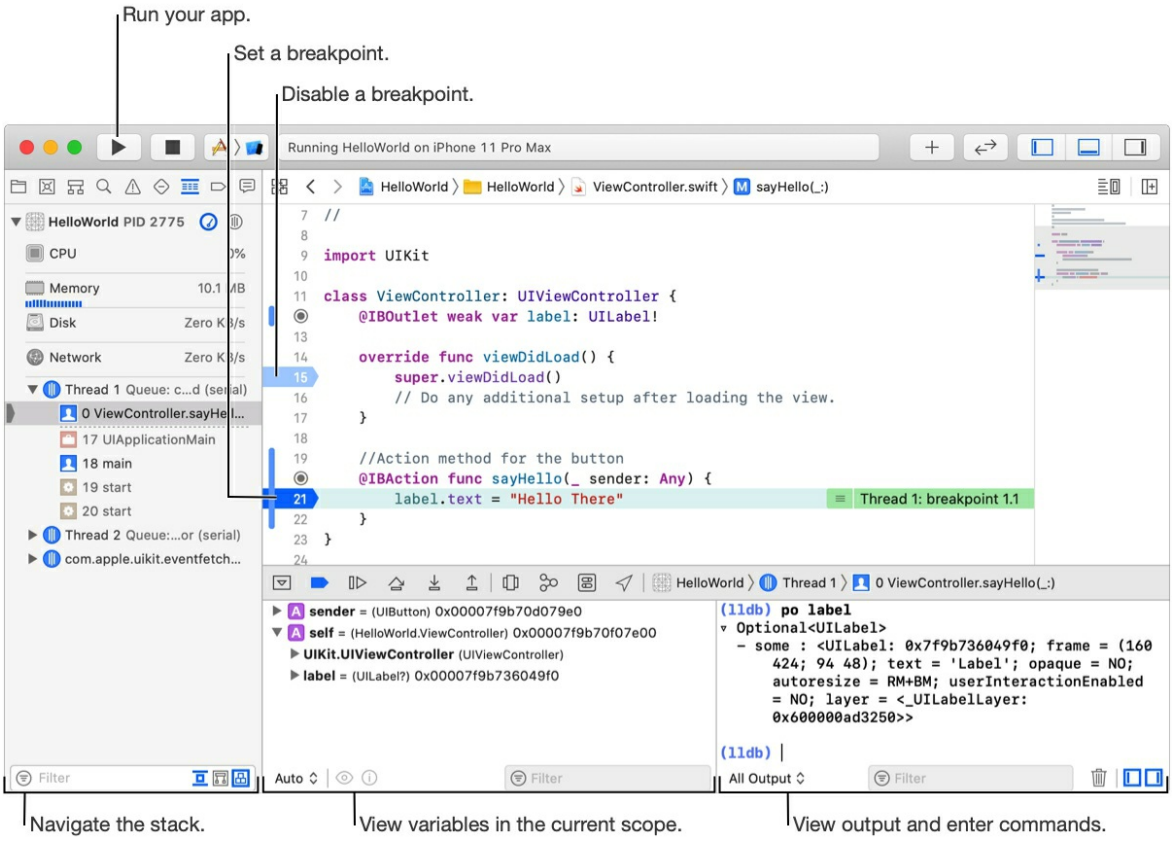
Report navigator

The Report navigator lists all the activity that Xcode has carried out with a project (like building, analyzing and debugging). It is even easier to navigate back and check previous build reports in your Xcode session.

The Debug area

About the debug area

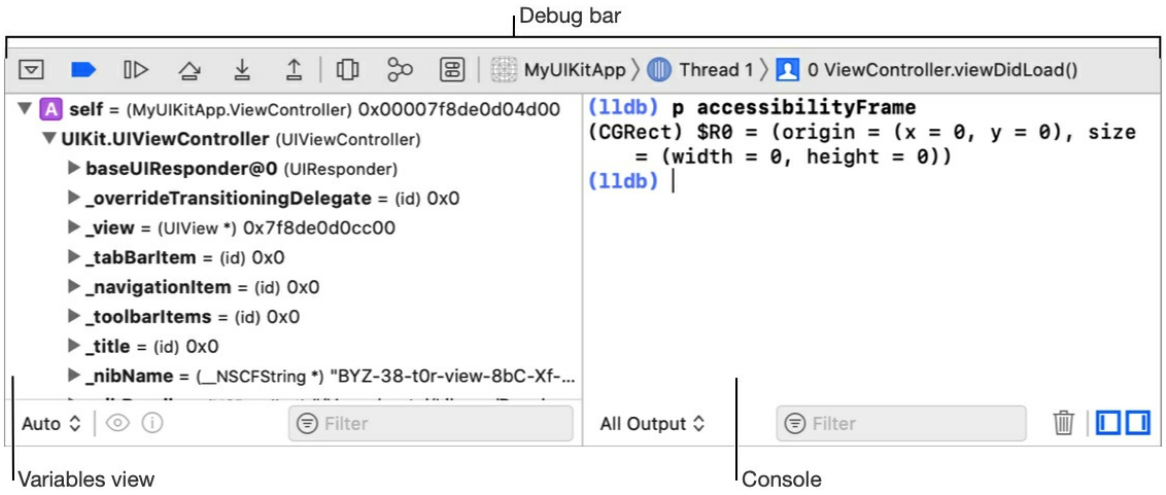
You can inspect your code while running your application by using the Xcode debugger. Automatically, you will see the debug area and the Debug navigator when you create and start your application. If necessary, you can display the navigator area by tapping on the left button () and display the debug area by tapping on the middle button () on the right of your toolbar.



The following are the three main parts of the debug area:

- The *debug bar* has buttons that you can use to enable or disable all breakpoints, enable graphical debugging of memory state and view, control the execution of your app, simulate location, jump to stack frames and override environment settings.
- The *variables view* brings the list of variables that are available to inspect within the scope of your present location in the code. This list is actually a disclosable hierarchy, showing the values of all parts of the structure of a variable as you continually tap the disclosure triangles.
- The *console* contains a text area like an interactive Terminal. The console can be used to interact directly with the LLDB (The LLDB command-line debugger gives underlying debugging services for app development on all of Apple platforms. You can prompt LLDB debugging commands from your Xcode debugging console while debugging an app or from the Terminal window.), view result from use of Print Description, and also work with standard input and output from your app. For instance, enter po [expression] and the debugger will execute the expression. As you are

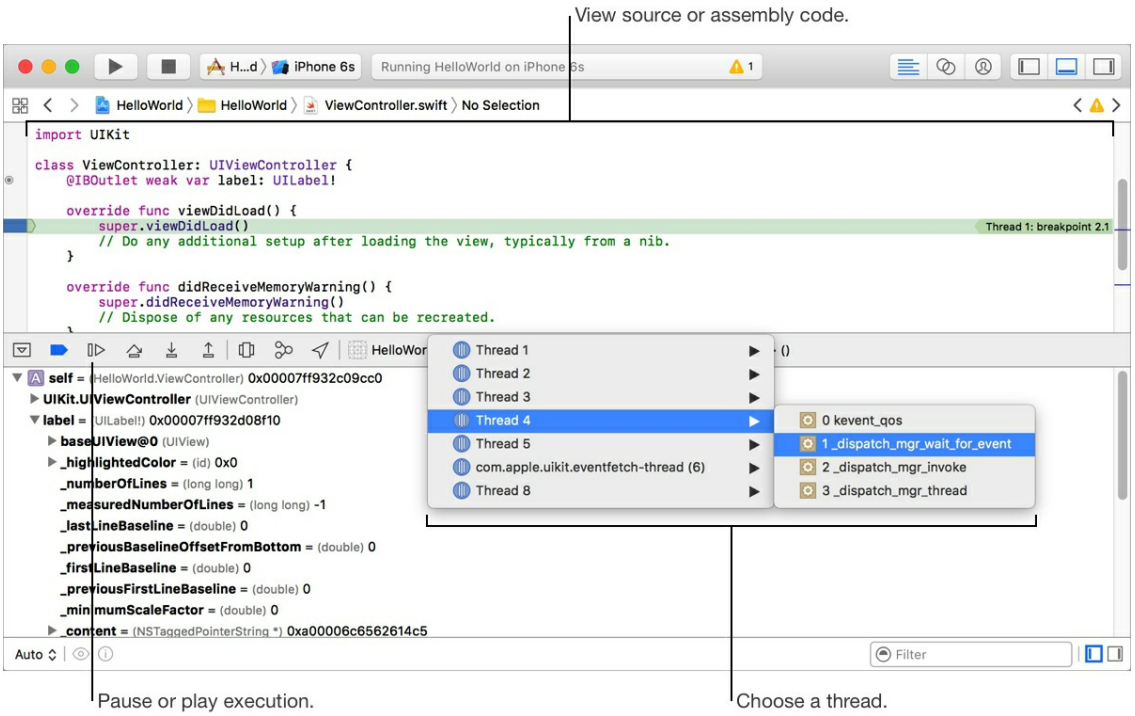
typing your expression, the debugger will continually offer you useful suggestions for completing what you are typing.



Multiple applications and processes can be debugged at the same time. You will need to create a separate project window for each session.

View threads in the debug area

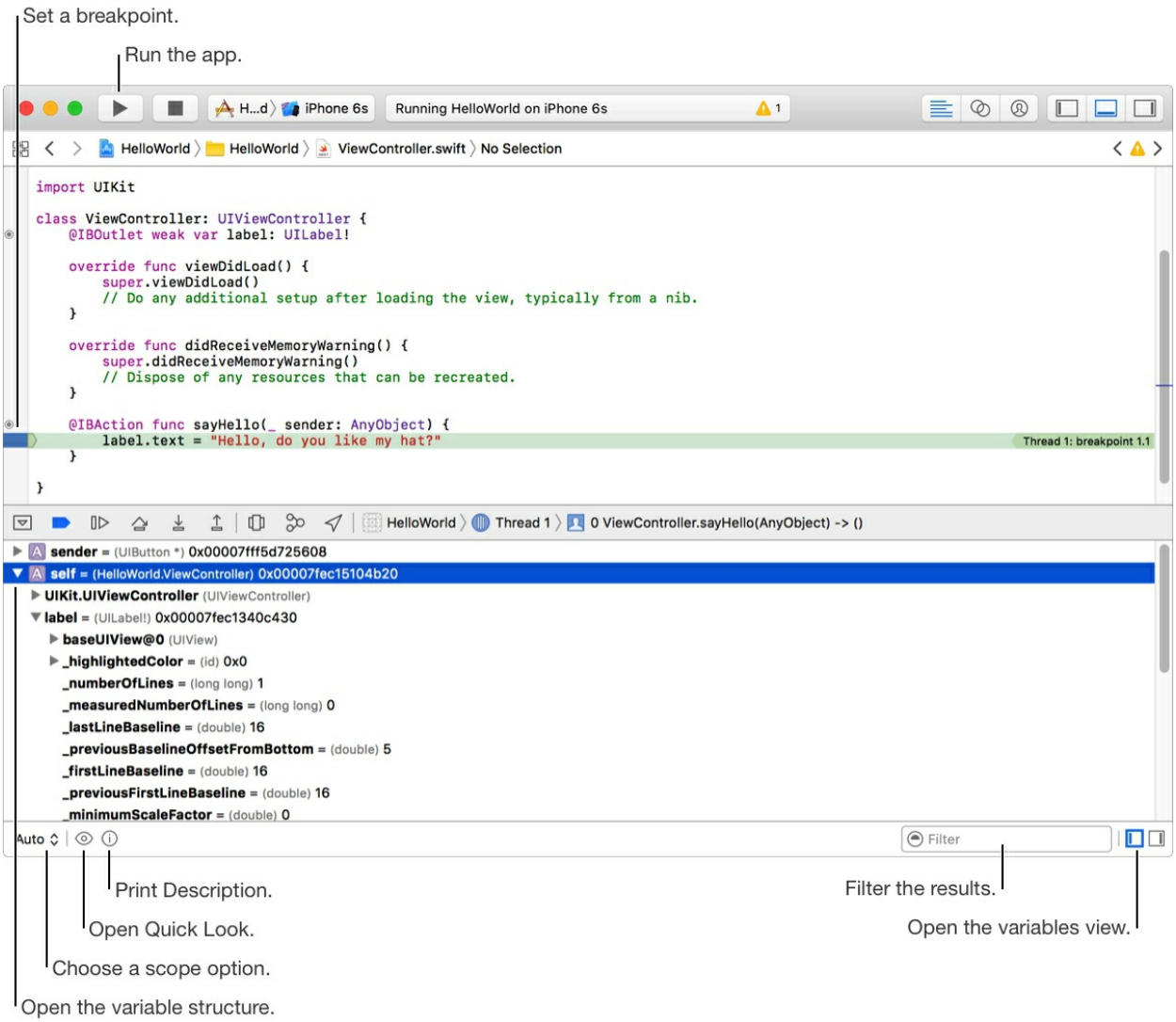
You can leverage the debug area to examine the stacks and threads of your running app. When you choose a thread, or a stack within a thread, in the debug bar, the Xcode will show you the corresponding assembly code or source file in the source editor. The debug area opens automatically when you create and run your app.



1. In the debug area, tap on the Pause button or exercise a little patience for the application to terminate at a breakpoint.
2. In the debug bar, select a thread from the displayed pop-up menu.
3. In the source editor, check the corresponding assembly code or source code.

View variables in the debug area

In the debug area, you can uncover a problem in your source code by inspecting the value of a variable. The debug area opens automatically when you create and run your app.



Each item inside the variables view list displays; the name of the variable as it appears in the code, the current value of the variable, the runtime type of the variable, and a summary for the variable, if available. The icon in the name of the variable indicates the kind of the variable.

View variable

1. In the debug area, tap on the Pause button or wait for your app to terminate at a breakpoint you had set previously.

The variables that are in scope will be shown in the variables view of the debug area. The variables that have been deallocated appear dimmed and cannot be inspected.

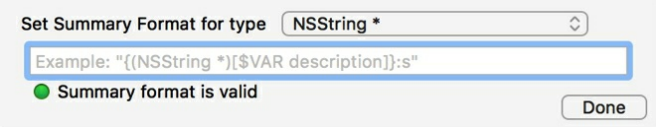
2. Select a scope option from the pop-up menu at the lower-left corner of the variable view.

Display recently accessed variables: Select Auto.

Display only local variables: Select Local.

Display all variables, registers, globals, and statics: Select All.

3. You can filter your results by entering texts inside the search field located at the bottom-right corner.
4. Tap the disclosure triangle located to the left side of the variable to access the structure of a variable.
5. You can edit the summary format of a variable by Control-click on the variable and then select Edit Summary Format… from the shortcut menu.

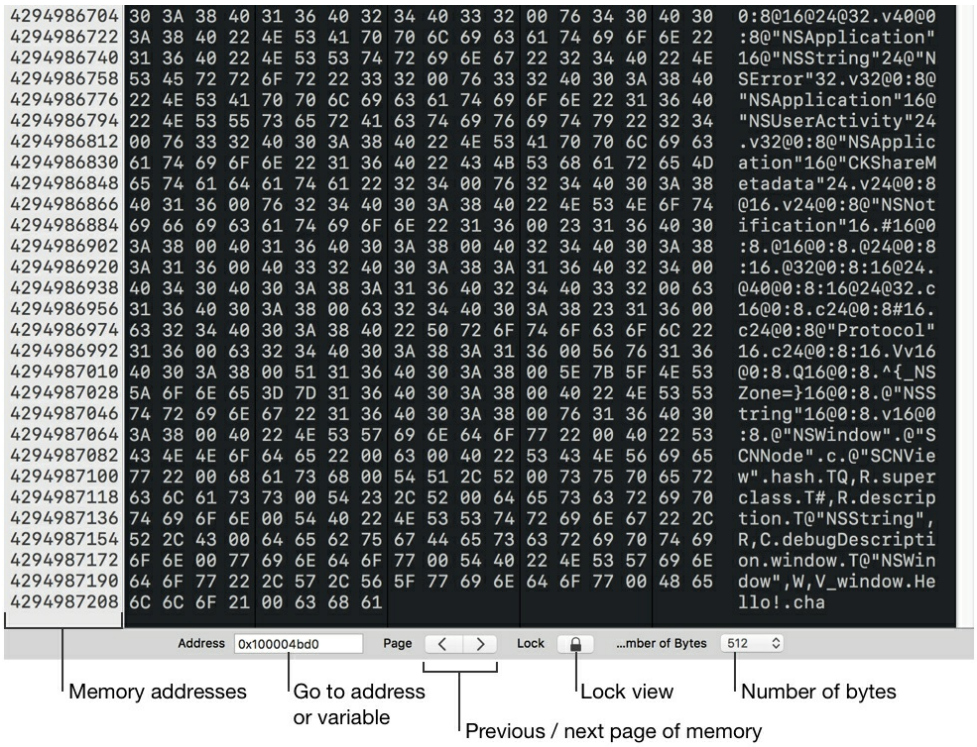


In the popover, input a valid expression and tap “Done.” The default formatter will be overridden by this expression and is used to create a summary for all variables of this type.

6. The value of a variable can be edited by Control-click the variable, select Edit Value… from the shortcut menu, and input a new value.

View memory for a running app

1. In the debug area, tap on the Pause button or exercise a little patience for the application to terminate at a breakpoint you had set previously.
2. Command-click on a variable and select View memory of “*variableName*” or choose the Debug > Debug Workflow > View Memory (Shift- ⌘ -M) menu item to launch the *memory console*.



The memory console displays the ASCII contents and hexadecimal of particular memory address ranges.

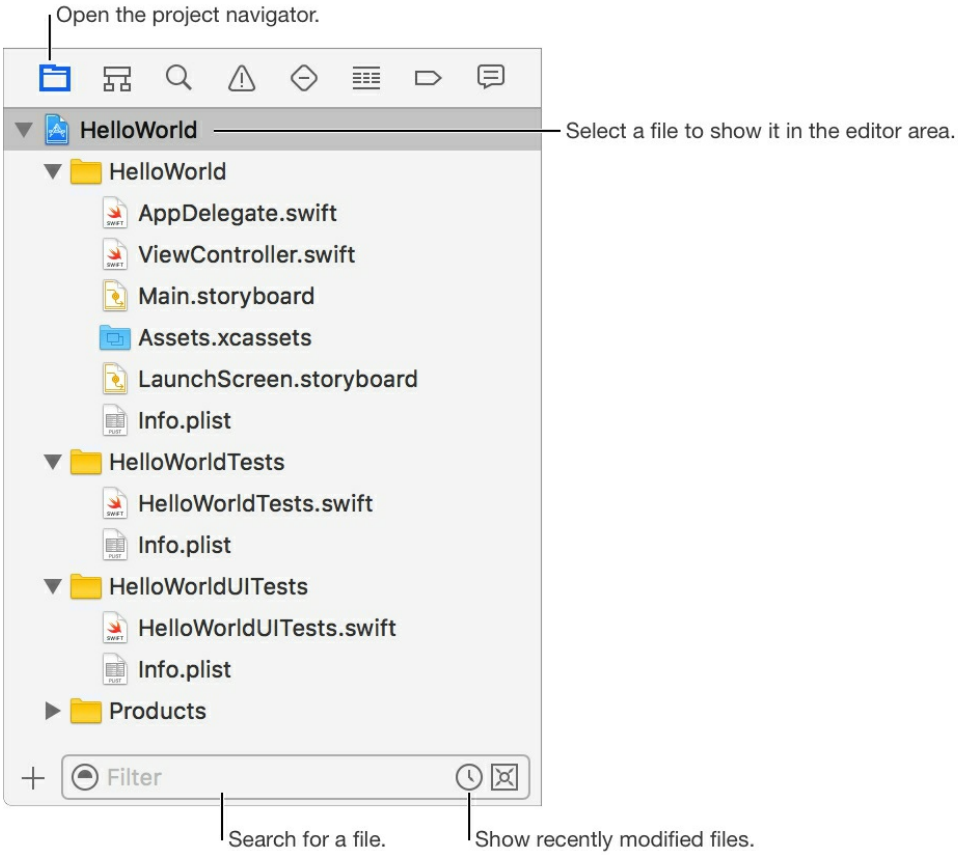
3. Tap on the memory addresses column to pick between hexadecimal and decimal representations.
4. Input a memory address or the variable’s name in scope to navigate to that region of memory.
5. Click the previous and next page controls to navigate between pages of memory.
6. Click the lock view button to prevent updates to the contents of memory in the current view.
7. Choose a different number of bytes to show on a single page.

Managing project file

About the Project navigator

The project navigator, like it has been said before, can be used to launch, add, delete, and organize files in a project. Tap on the Project navigator button (▭) located at the top section of the navigator area and your project files will be

shown in the content area below.



Open a file: Tap the file to quickly open the file in the editor area.

View properties of a file: Select the file by clicking on the file, then tap View > Inspectors > Show File Inspector. The properties of the file will be displayed in the File inspector.

Search for a file: Write anything you want to search inside the filter field below the content area.

Show recently modified files: Tap on the Recent Files icon (🕓) inside the filter field.

Show files with source control status: Click on the Source Control icon (⊠) inside the filter field.

Unlock a file: Choose the file, then select File > Unlock. (If the file is already unlocked or you don't have permissions to unlock the file, the menu item is disabled.)

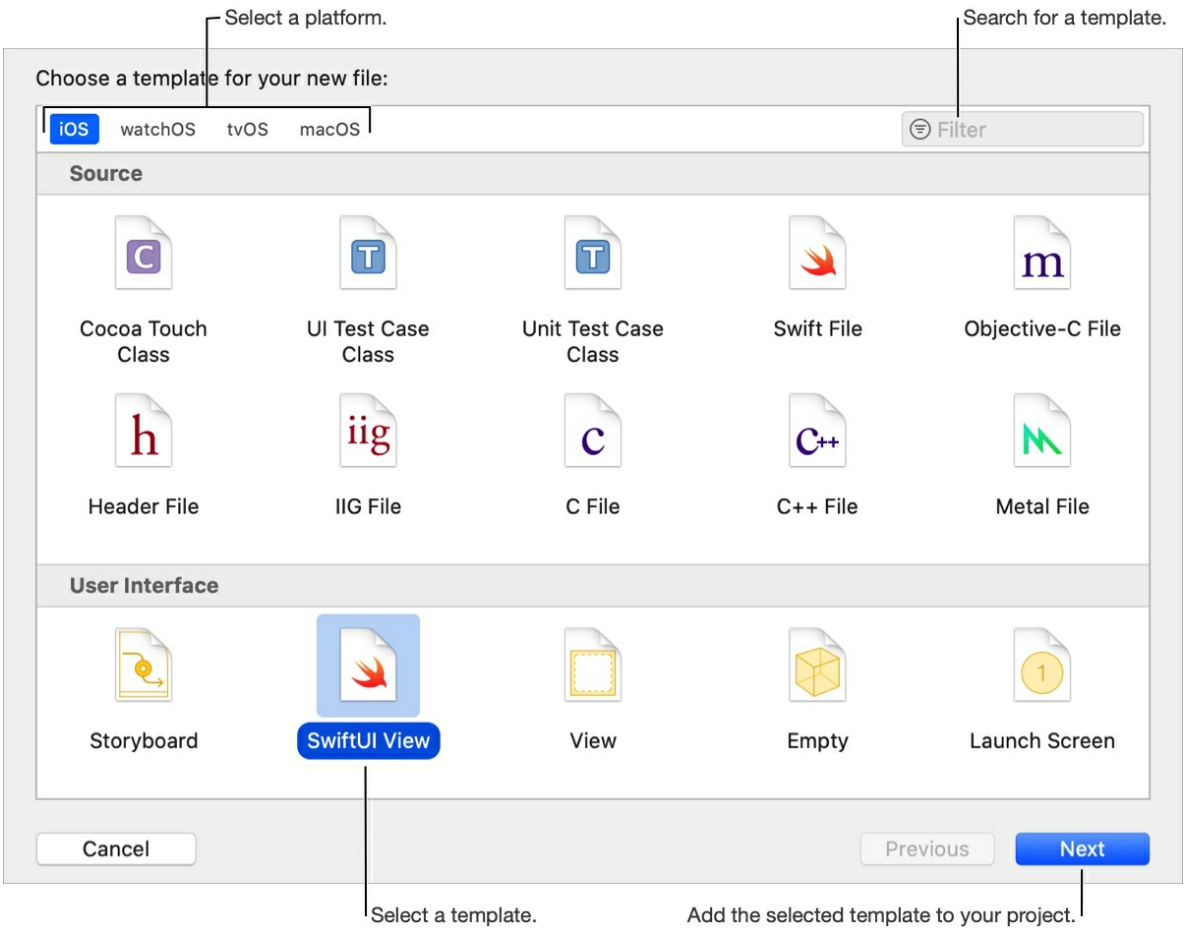
Choose a relative or absolute location: In the File Inspector, choose a

location from the Location pop-up menu. For example, choose Relative to Group (recommended) to preserve references when you move your project folder.

Modify the default move and copy behavior: To force a move operation, press and hold Command (⌘) while dragging files. To force a copy operation, press and hold Option (⌥). To force a reference operation, press and hold Command-Option (⌘ - ⌥).

Add files and folders to a project

Xcode has templates for the common kind of files you may wish to add to your project, such as Swift files or Playgrounds. You can as well add copies of, or references to, existing files and folders on your computer.



Adding existing files and folders

1. In the Project navigator (), choose the precise location where you

want the file to be added.

2. Tap on the Add button (+) in the filter bar and select File from the pop-up menu (or choose File > New > File).
3. Tap iOS, tvOS, watchOS or macOS located at the top of the page to bring the templates for that platform. The templates are often organized into groups like Source, Resources and User Interface.
4. Choose a template for the file type, and tap “Next.”
5. In the following window, input the required information and tap “Next.” For instance, fill in a class name for a class implementation file.
6. In the last page that will be prompted, select a location and input a filename (if applicable).
7. From the Group pop-up menu, select a group. If the group is associated with a folder (the default when you create a group), the project structure is the same as the file system structure.
8. Choose the targets that you plan to add the file to.
9. Tap on “Create.” The new file is selected in the Project navigator and opens in the appropriate editor.

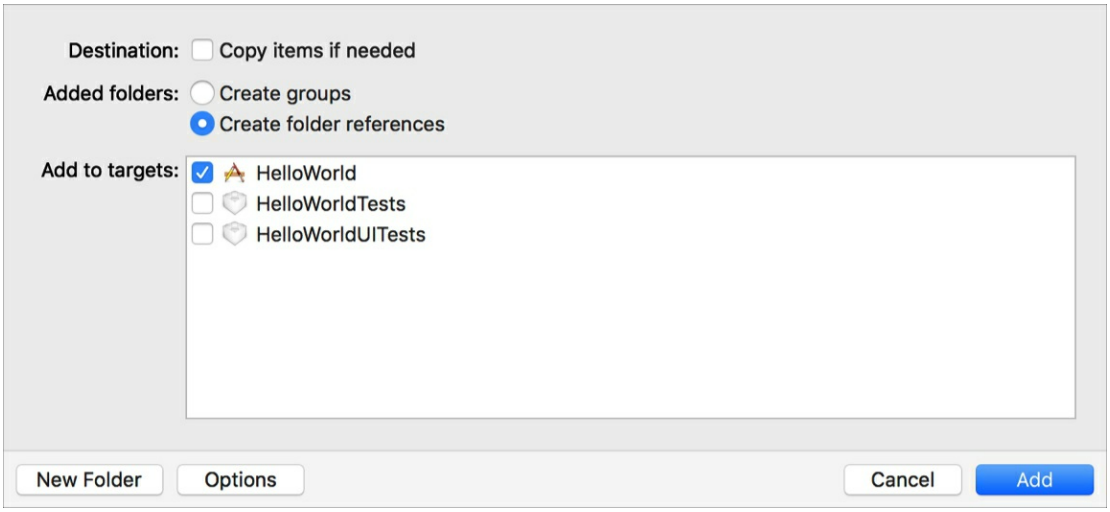
Adding existing files and folders

1. In the Project navigator (), choose the destination group or project for the item you want to add.
2. Tap on the Add button (+) inside the filter bar, select Add Files to “[Project Name]” from the pop-up menu (or choose File > Add Files to “[Project Name]”), and choose the files or folders.
3. Tap on **options** at the bottom of the page.

The listed are the options for how your folders and files can be added; Choose

- *Copy items if needed:* Copies the files and folders to the project folder.
- *Create groups:* Keeps the group structure the same as the file structure.

- *Create folder references:* Shows the folders in the Project navigator but doesn't copy them to the project. A *folder reference* is a reference in the Project navigator to a folder in the file system.



In Add to targets, select the targets (target is the product that you want to build) that you wish to add the file to.

Optionally, tap on New Folder if you want to add a folder for your files.

Click Add.

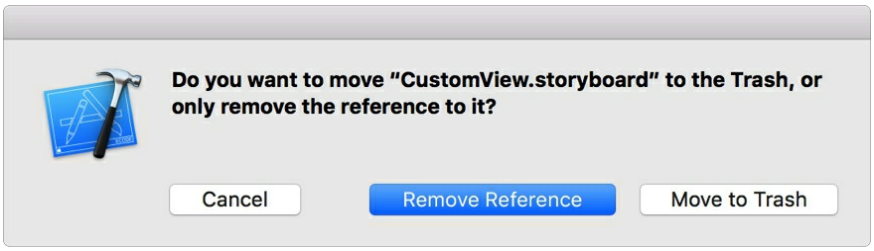
Note: You can also drag the files from the Finder to a location in the Project navigator to add them.

Deleting files and folders

1. In the Project navigator (), choose the files and folders.
2. Choose "Edit" and then tap "Delete."
3. Choose a delete option from the displayed dialog.

Remove the files and folders from the project and the file system: Click Move to Trash.

Remove the files and folder references from the project only: Click Remove References.



Organize files in groups

Organize the files that you have added to your project with Groups. For instance, the project you created from a template will contain a group containing the files for each target. A group is associated with an underlying folder that has the same name by default.

Add a new group: In the Project navigator, tap where you plan to add the group, then select File > New > Group. To create a group without an associated folder, click File, select New and choose "Group without Folder."

Add files to a new group: Choose the files, then tap File > New > "Group from Selection". The files will be transferred to the associated folder in the file system.

Add files to an existing group: Choose the files and then drag the files to the group. If the group is associated with a folder, the files will be moved to the associated folder.

Rename a file or group: Click twice on the group or file and then input the new name for the group or file.

Change a group's associated folder: Tap on the group, then select View > Inspectors > Show File Inspector.

CHAPTER SEVEN

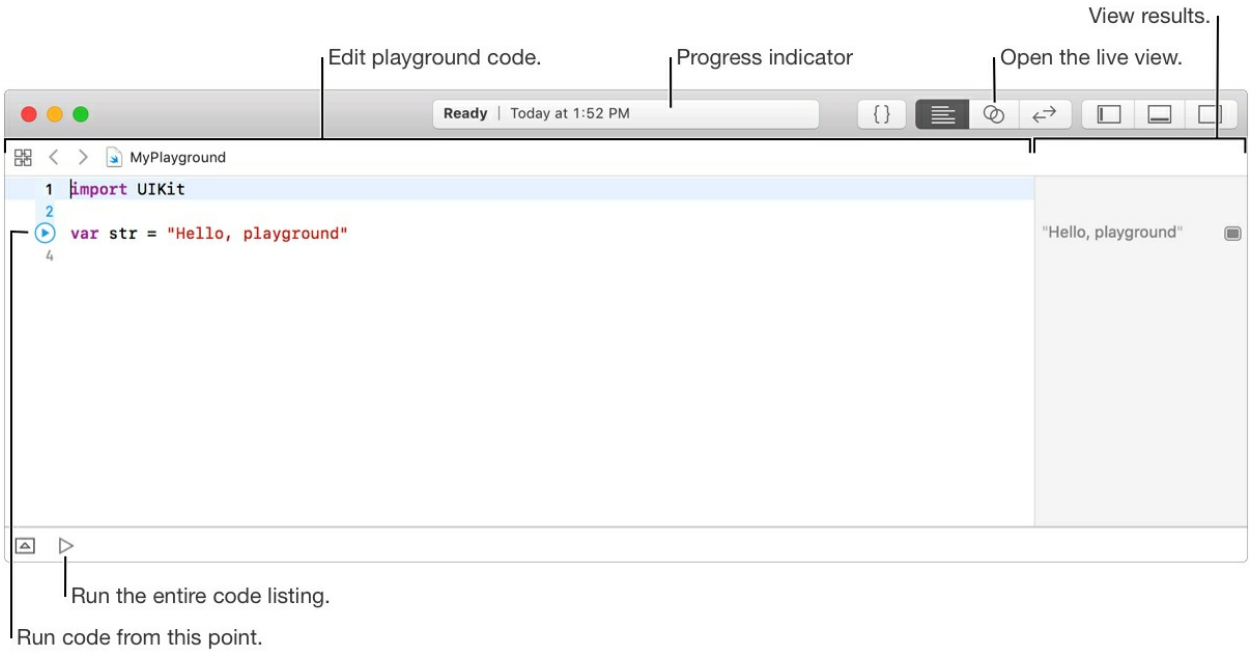
AN INTRODUCTION TO Xcode 12 PLAYGROUND

With the Playground feature in the Xcode 12, learning and programming with

Swift has never been that easier. Playground offers an interactive environment where users (developers) can enter executable Swift codes with the result showing in real time.

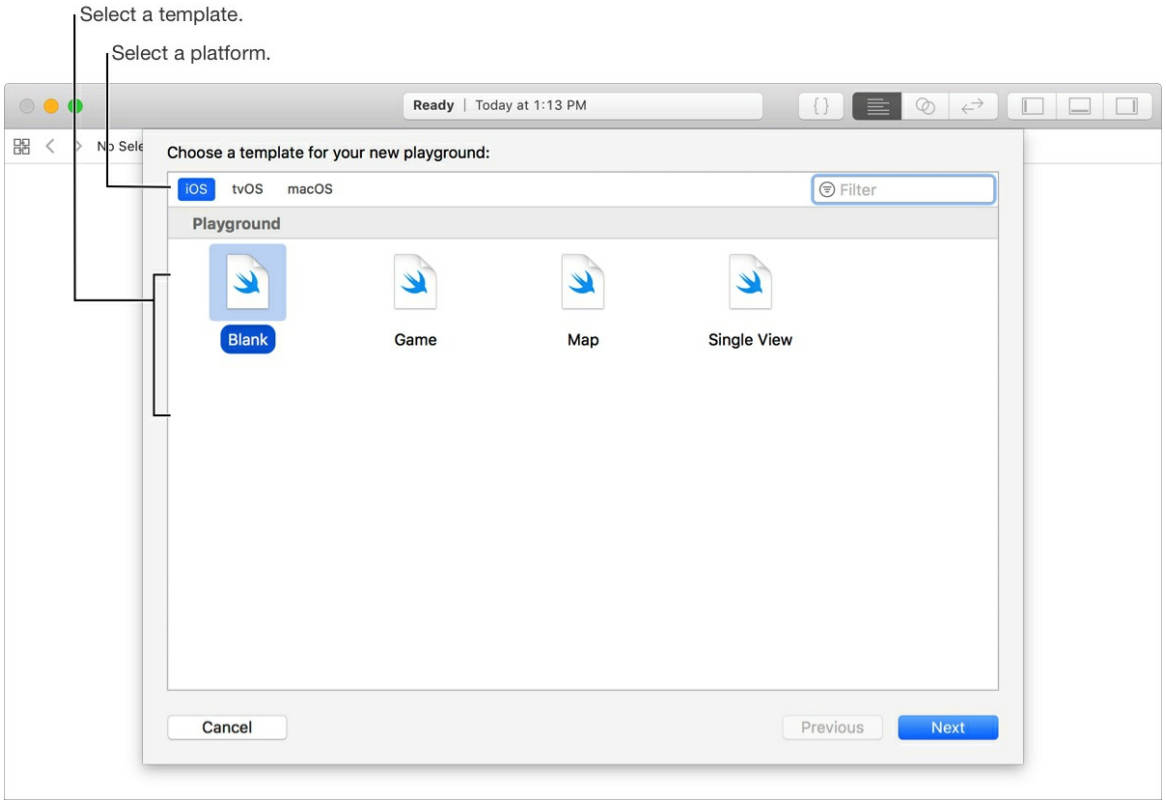
Create, edit, and execute playgrounds

You can use the playgrounds feature in Swift to learn and navigate the world of Swift, prototype parts of your app, and also create learning environments for others. The interactive Swift environment allows users to experiment with code, create custom views and even explore system APIs. Notes and guides can be added in your codes for other users by using the rich comments. Also, you can add navigation and assemble concepts that are similar into pages. In playgrounds, you can run code from the insertion point supporting an incremental development style. When every part of your codes has been perfected in the playgrounds, you can then transfer your codes successfully into your project.



How to create a playground

1. Tap “File,” select “New” and then choose “Playground.”
2. From the prompted page, choose the platform you want your playground to run on.



3. Under Playground, choose a template out of the available templates, then tap Next.

The available templates are:

- *Blank:* A generic playground.
- *Game:* A playground developed based on SpriteKit.
- *Map:* A playground that deploys MapKit.
- *Single View:* A playground with a single view.

Tap on the **"Next"** button at the bottom and you will be directed to a page where you can enter the file's name and choose a location. Once you have done these, tap on **"Create."**

Edit a playground

It is worth telling that the source editor in a playground has the exact features as the source editor inside the project editor.

- Input Swift code into the playground source editor: Xcode parses your codes as you enter them in the source editor. If there exists a syntax error

in your code, you will receive a prompt (message) beside the line of code that contains the syntax error. To see the full message that talks about the issue and suggests fixes, tap the error or warning icon. Then tap on the Fix button next to a suggestion to update your code.

- Deploy the code completion if you want to avoid syntax error henceforth.

Xcode gives you inline suggestions for completing the name of a symbol. Tap on an item in the suggestion list or deploy the Up Arrow or Down Arrow keys to choose it. Then hit Return to accept the suggestion.

For a method or function containing parameters, code completion places a placeholder for each parameter. To scroll to the next placeholder, press Tab; to move to the previous placeholder, press Shift-Tab.

- To replace texts or find specific texts, select Find followed by an option.

For example, click Find > Find and Replace, then enter text in the Replace and with fields and press Return.

Run a playground

You can run your code automatically by stopping entering code or rather manually when you are ready. While your code executes, you will see a progress indicator in the toolbar showing on the right, and when the code ends, you will obtain the results in the same location.

- To have a view of the playground live view while you are still running the code, launch the assistant editor.
- To switch between run modes, tap and hold the Run button, then select:
- *Automatically Run:* Choosing this, you will be able to run your code each time you enter a statement or pause typing.
- *Manually Run:* Choosing this, you will be able to run the code only when you click a Run button.

Note: In most of the available templates, the manual mode is the default mode.

To run code from the insertion point in manual mode, hover the pointer over the line number in the gutter, then click the Run button that appears.

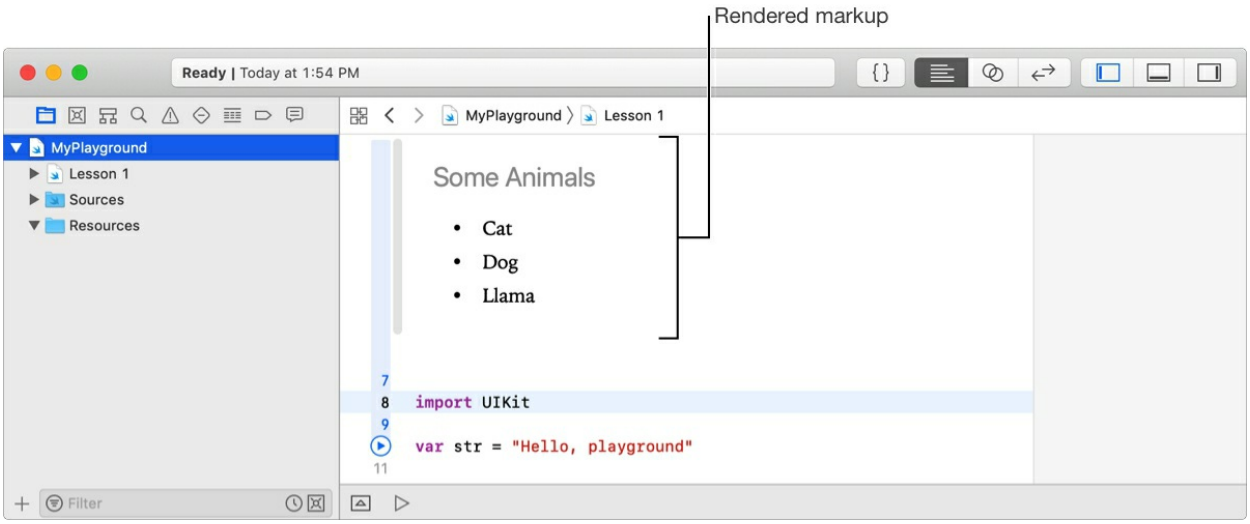
To run the entire code listing in manual mode, click the Run button at the bottom of the window.

Add, edit, and view rich comments

You can add formatted text to your playground by using the playground markup format. The markup format supports rulers, headings, lists, emphasis, code voice, bold, assets, links, and more. The rendered text can be accessed by switching to rendered markup mode, and you can switch to raw text mode by editing the markup.

The below markup renders as a title followed by a bulleted list.

/*:### Some Animals* Cat* Dog* Llama*/



View the rendered markup: To view the rendered markup, select “Editor” and click on “Show Rendered Markup.”

Edit the raw text: To edit the raw text, select “Editor” and click on “Show Raw Markup.”

Add auxiliary code to a playground

If you plan to add code that you don’t want to recompile each time you run the playground or you don’t want the user to see the codes in the playground, simply add the source files to the playground or a page Sources folder. Xcode will compile files in Sources folders only when you add the files or save edits to the files.

The public symbols in files you added to the playground Sources folder will

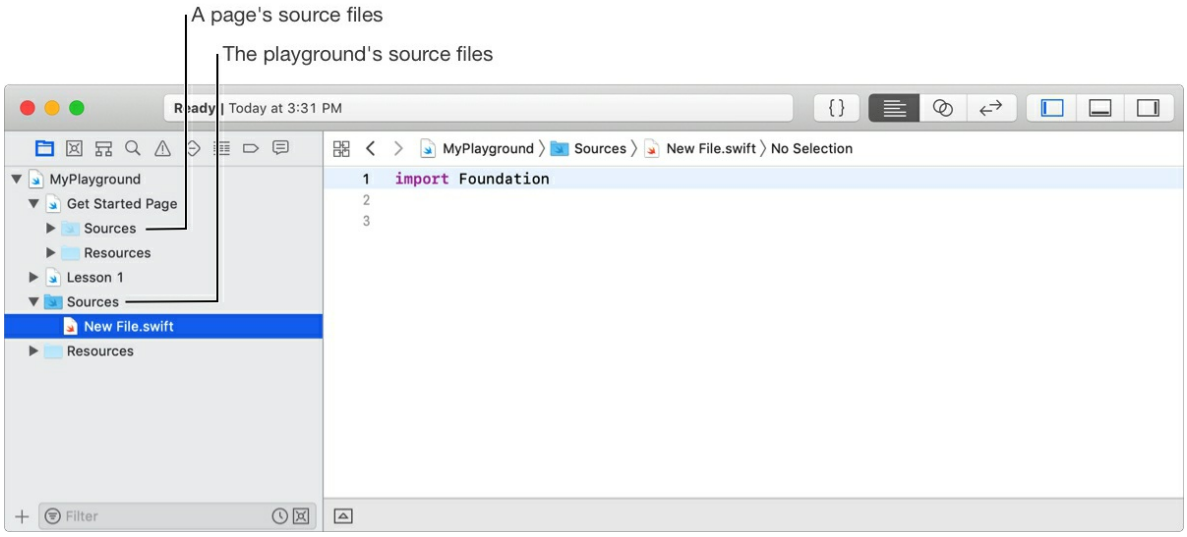
be made available to all pages in the playground. On the contrary, the public symbols in files you added to the page Sources folder will only be made available to the page containing the Sources folder.

Important: The compiled code imports into a playground as a module. To view a symbol (class, method, function, variable, or protocol) in the playground, the auxiliary Swift source file must use the public keyword to export the symbol.

Add a new source file

1. Launch the Project navigator (click on View, select “Navigators” and then click on “Show Project Navigator”).
2. In your Project navigator, choose the Sources folder for the page or playground, tap on the Add button (+) located in the filter bar, then select Add Files to "Sources" from the pop-up menu (or tap on File, select “New” and click on “Add Files to "Sources"”).

A new Swift file will appear in the Sources folder and will be opened in the source editor.



3. Edit your code in the file, then select File > Save (Command-S) to compile the code.

Add an existing source file

1. In your Project navigator, choose the Sources folder for the page or playground, tap on the Add button (+) located in the filter bar, then select Add Files to "Sources" from the pop-up menu (or tap on File,

select “New” and click on “Add Files to "Sources"”).

2. In the dialog that is displayed, choose the source file and then click Add.

The file appears in the Sources folder.

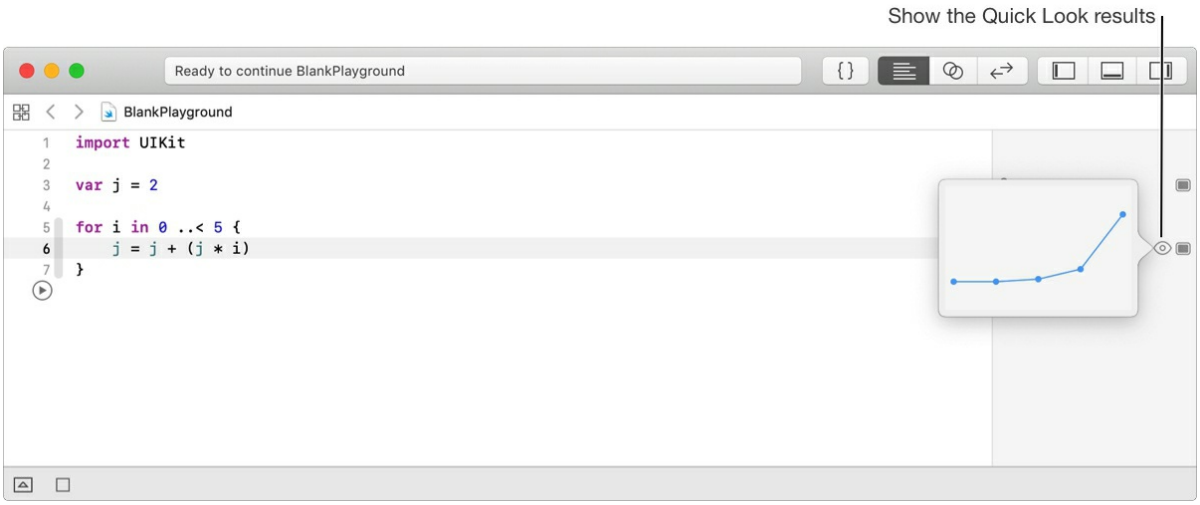
View results of an executed statement

See the detailed result of an executed statement in a Quick Look preview or simply by adding a results view to your playground. Some results can be viewed in different ways. For instance, the set of results for a number variable in a loop can be viewed as a set of values for each iteration, a graph of the values for each iteration of the loop or the value of the last iteration.

View a Quick Look preview of the results

- In the sidebar, hover the pointer adjacent to the Show Result button for the line of code, then tap on the Quick Look button that appears.

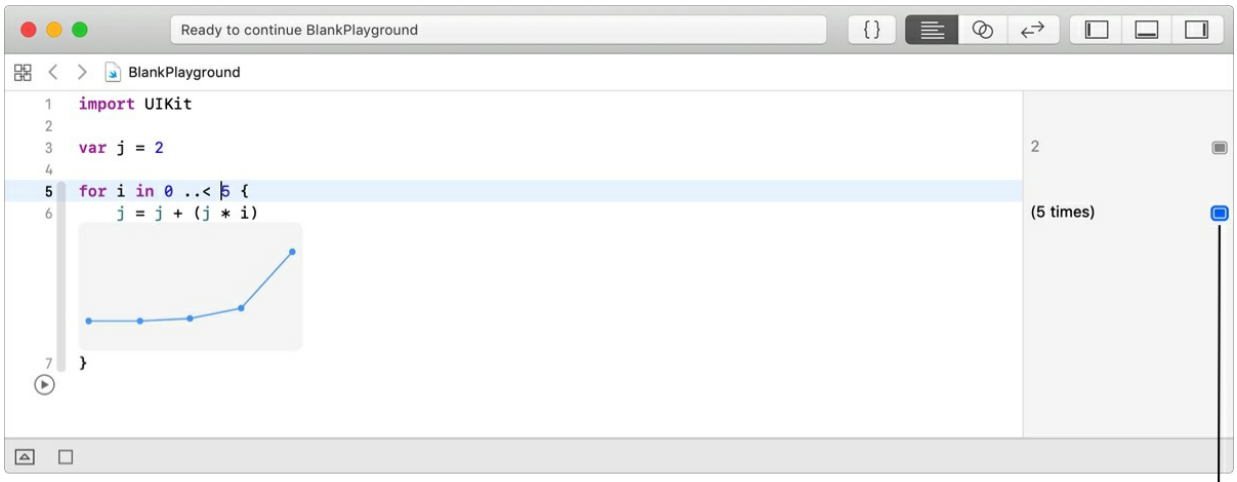
The results view appears in a popover.



Add a results view to the playground

1. In the sidebar, tap on the Show Results button for the line of code (or position the insertion point in the line of code, then select Editor and tap on “Show Result for Current Line).

A results view will show in the editor below the line of code.



Hide a results view in the playground

- In the sidebar, tap on the Show Results button for the line of code (or position the insertion point in the line of code, then select Editor and tap on “Hide Result for Current Line”).

Resize a results view

1. Hover the pointer over the corner or edge of the results view.

The pointer will change to a double-headed arrow.

2. Resize the edge of the result view by dragging the edge.

Change the display of a results view

You can access some results in different ways. For instance, you can view the value of a number in a loop as a graph, as the last value or as a series of values.

- Control-click the results view in either the Quick Loop popover or the source editor and then select an option from the popup menu:

Latest value: Shows the current value.

Value history: Shows all the values.

Graph: Shows the values as a graph.

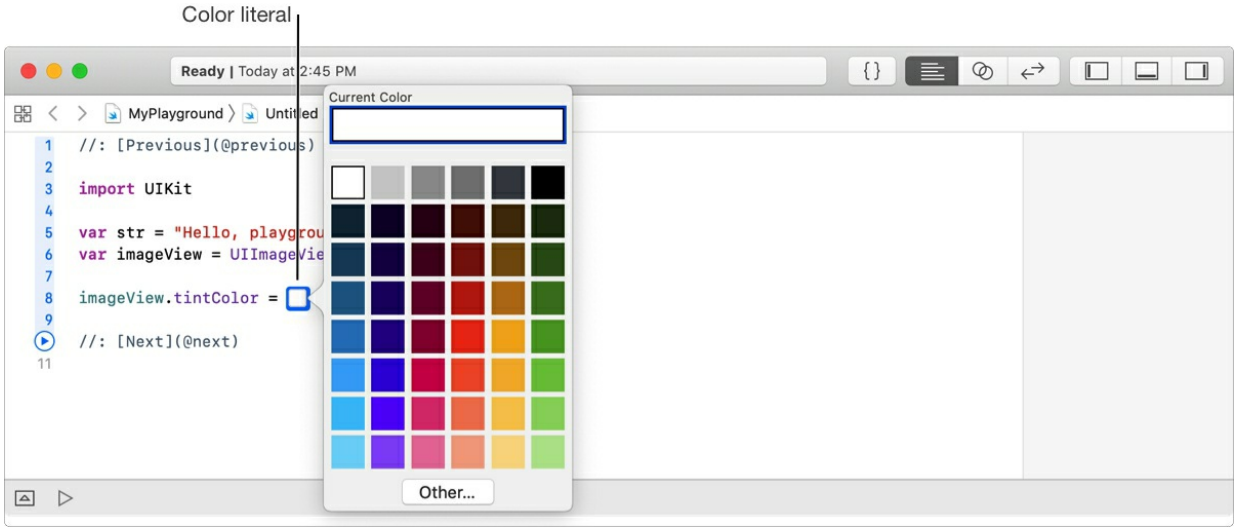
You can alternatively select the results view in the source editor, then click on “Editor > Result Display Mode” and any of the above options.

Add a color, file, or image literal to a playground

You can create literals in your playground code when the value of a color, file, or image does not need to change. The same literal can be used in many places in your playground. The types of literals are: **Color, Files** and **images.**

Adding a color literal

1. Position the insertion point inside the code where you plan to add the color literal.
2. Choose "**Editor**" and select "**Insert Color Literal**."



3. Tap twice on the color wheel that shows and then select a color from the color picker.

Add a file literal

- Position the insertion point inside the code where you plan to add the file literal.
- Choose "**Editor**" and select "**Insert File Literal**."
- In the page that shows, choose a file, then click Open.

Adding a file literal to a playground page will automatically add it to the page's Resources folder; otherwise, it will be added to the playground Resources folder.

Add an image literal

- Position the insertion point inside the code where you plan to add the file literal.
- Choose "**Editor**" and select "**Insert File Literal**."
- In the page that shows, choose a file, then click Open.

Adding a file literal to a playground page will automatically add it to the page's Resources folder; otherwise, it will be added to the playground Resources folder.

Alternatively, you can drag an image file to the source editor.

Copying and moving literals

- Use the Edit menu to cut, copy, and paste literals.

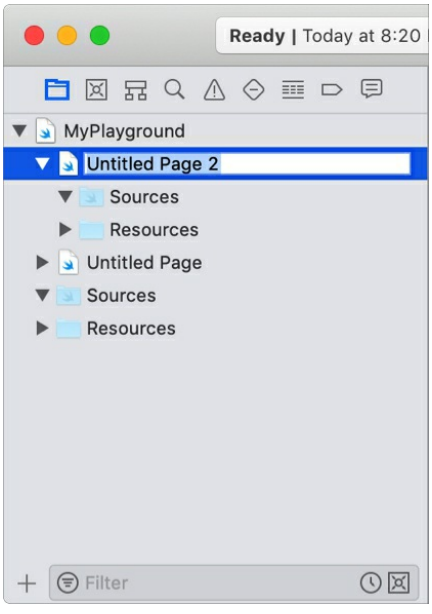
Add, move, and rename playground pages

Add pages to your playground to create separate lessons. Then you can rename the playground pages and change the order.

Add a new page

1. Launch the Project navigator (select View, tap on Navigators and choose "Show Project Navigator").
2. Choose File, select New and choose Playground Page.

Two new pages will show in the playground group the very first time you add a page to your playground. One will contain the current content of the playground editor and the other will be the new page.



Rename a page

1. In the Project navigator, choose the page and tap Return. The page's name will become editable.

2. Input the new name of the playground page inside the text field and click on Return.

Change the order of pages

- Change the order of pages by dragging a page to another location in the list of pages.

Copy a page from one playground to another

1. Launch the Project navigator (choose View > Navigators > Show Project Navigator) in each playground.
2. Drag that page from the source playground to the destination playground.

The page and any associated resources and sources will be copied to the destination playground.

Add an interactive live view

The live views can be deployed to make your playground to be interactive as much as possible, create your own custom elements and experiment with various user interface elements. Add an interactive live view to your playground by importing PlaygroundSupport and setting the live view of the current page to your custom view or view controller.

Note: If you choose the Single View template when you create a playground, your code already sets the live view. Simply launch the assistant editor and run the code.

Add a live view

1. In your code, import PlaygroundSupport.

Add this code snippet;

```
import PlaygroundSupport
```

2. Set the page's live to an instance of your view or view controller.

For instance, the Single View template will set your page's live view to an instance of MyViewController:

```
PlaygroundPage.current.liveView = MyViewController()
```

3. Open the assistant editor in the playground (select View, tap Assistant Editor and click on Show Assistant Editor), then run the code.

The live view will render inside the assistant editor and will not stop running until you stop it or an error occurs.

CHAPTER EIGHT

USING Xcode in SwiftUI Mode

CREATING A SWIFT UI INTERFACE

Show a preview of your user interface

You can display an interactive preview of the SwiftUI code that you edit in the source editor.

As you make edits in your code, Xcode builds and runs the code, and displays the results in the canvas. The codes you enter in the source editor and the user interface layout in the canvas are kept in sync by the Xcode. If a *user interface element* is added to the canvas from the library, Xcode will add the corresponding code to the source file. If you choose an element on the

canvas, the corresponding code is selected in the source file. For every element's properties you change in the inspector, Xcode will add code to the source file.

To get started using SwiftUI, either choose SwiftUI as the user interface when you create an Xcode project, or add a file that uses SwiftUI to an existing project.

Add a file that uses SwiftUI

1. Select "**File,"** tap on "**New**" and click on "**File**."
2. You will be taken to another page, choose the platform, choose SwiftUI View under User Interface and then tap "**Next."**
3. Another page will be prompted where you will input the name of the structure. Once you are done, tap on "**Create**."

Optionally, select a group from the pop-up menu and choose an alternate target.

The canvas will appear automatically to the right of the source editor.

Show a preview

1. In the Project navigator, choose a file that uses SwiftUI, then select "**Editor**" and tap on "**Canvas**."

Alternatively, you can select Canvas from the Editor Options pop-up menu located on the right of the jump bar.

2. Tap on the Resume button located in the upper-right corner of the canvas to start the preview.

Run the app on a simulated device with or without a debug session

You can switch from the preview to a *live preview* where the app is run on a simulated device directly in the canvas, or switch to a *debug preview* that has a debug session. For macOS apps, the app will run on the desktop, not in the canvas. First display a preview, then deploy the controls in the lower-right corner of the canvas to switch between modes.

- Tap on the **live preview** button inside the canvas to start running the app.

iOS,watchOS and tvOS apps will run on a simulated device in the canvas.

macOS apps run on the desktop. Click the Bring Forward button in the canvas if the window is not showing.

- If you plan to run the app with a debug session, simply Control-click the Live Preview button in the canvas, then select Debug Preview from the pop-up menu.

When the app opens, you will see a debug session in the debug area.

- To terminate the debug preview or the live preview, click on the Live Preview button in the canvas. Xcode will go back to the preview mode.

Run on a connected device

For iOS, tvOS, or watchOS apps, your app can be opened on a device from the preview but the app must be code-sighted before launching.

Before you start, you will need to add your Apple ID account and also assign the target to a team, then show the preview.

1. Connect the device to your Mac.

   For watchOS apps that depend on an iOS app, connect an iPhone that has been paired with an Apple Watch.

2. Select the Preview on Device button (located below the Live Preview button) in the canvas.

If your app refuses to run, go through the error message above the canvas and click on Diagnostics for more details.

Add views and modifiers from the library

You can lay out much of your SwiftUI user interface with standard views and modifiers that you drag from the library to either the canvas or your source code.

1. In the Project navigator, choose a file that uses SwiftUI, then tap on the Library button (+) located in the toolbar.

The library will open in a separate window. You can alternatively option-click the button to launch the library in a persistent window.

2. Click on the buttons inside the toolbar of the library to switch between the different libraries.
3. Drag an element from your library to the source editor, canvas or inspector.

If you drag an element to the canvas, you will see a valid destination showing in blue and a popover with information will also appear below. If you drag an element to the source editor, the code will shift to show where the code will be inserted.

Regardless of where the changes have been made, Xcode keeps your source code and user interface layout in sync.

Edit user interface element attributes

There are many ways you can use to edit user interface element attributes.

Tip: Learning SwiftUI becomes easy when you edit attributes in the inspector and view the resulting changes to the source code.

- Command-click the structure in the code or element in the canvas, select "**Show SwiftUI Inspector**" from the Action menu and then change the attributes in the next pane.

Xcode helps to keep the layout and source code in sync.

- Select "View," tap on "Inspectors" and choose "**Show Attributes Inspector**," then change the attributes in the Attributes inspector that appears on the right. Xcode will update the source code.
- Input the code in the source editor. Xcode updates the preview.

CHAPTER NINE

Connecting codes with the user interface using the Interface Builder workflow

The **Interface builder** can be used to connect your codes with your app's user interface.

Note: For Swift apps, the SwiftUI can be optionally used to lay out your app interface and see an interactive preview. iOS apps using SwiftUI also have a LaunchScreen.storyboard file that you edit using Interface Builder.

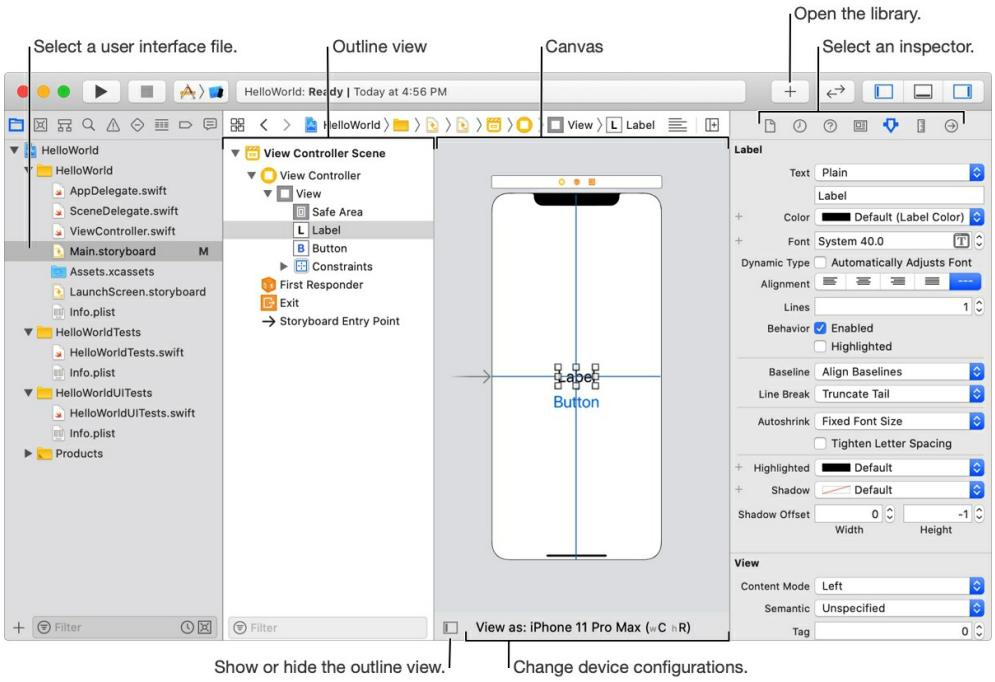
Step 1: Create a storyboard or XIB project

To deploy Interface Builder, select XIB (macOS) or Storyboard as your user interface when you create your project from a template. The project then has a main user interface file (a MainMenu.xib file or Main.storyboard, Interface.storyboard) that features the view controllers and views that appear when your app first launches. For iOS apps, there is also a LaunchScreen.storyboard file for the view that is shown while the app is launching.

Step 2: Open a user interface file

In the Project navigator, choose a user interface file and the file will be opened in Interface Builder in the editor area. The views will appear in the canvas area and the structure of the underlying objects will appear in the outline view. Simply tap on the toggle (▭) below the canvas to expand the outline view if it collapses. The first time you launch a storyboard file, you will see the layout appearing in a default device configuration that you can change later.

To launch the Interface Builder in a separate window, simply control-click the user interface file, then select "Open in New Window" from the pop-up menu.

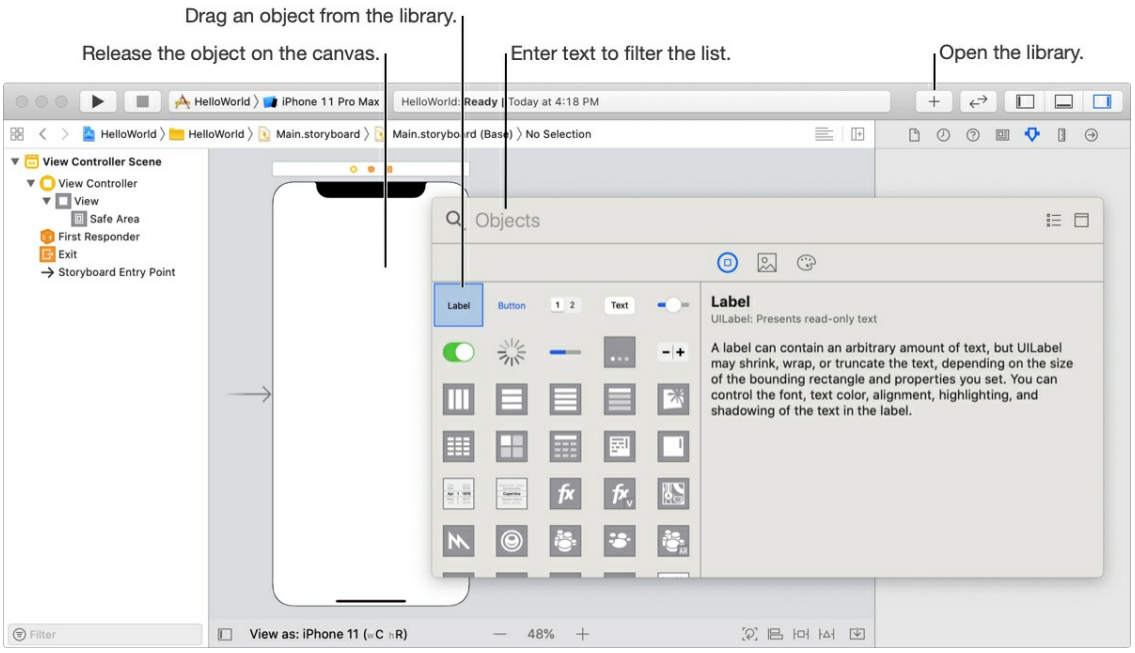


Step 3: Add controls, images, and other UI objects

Drag all of the objects you need in your user interface to the canvas from the library. To open the library, tap on the Library button (+) in the toolbar, then click the Image (), Objects (), or Color library () to find the objects that you want to drag.

In the canvas, the objects can be repositioned by dragging them to anywhere you want them using the gridlines to help center and align the objects. To change text (for instance, to edit the title of a button), tap twice on the object and then write the text.

Inspectors can also be used to edit objects. Choose the object on the canvas and tap on the Attributes inspector (). You can also tap on the Size inspector () to access information about the size and position of a view.



For macOS apps, add items to the Touch Bar. Drag any NSTouchBar object from your Object library to a window or custom view. Drag NSTouchBarItem objects to the Touch Bar and connect the items to your code. Then you can preview the items in your NSTouchBar object, test the items using a real Touch Bar on a Mac, or if you don't have one, using the Touch Bar simulator.

Step 4: Connect views and controls to your code

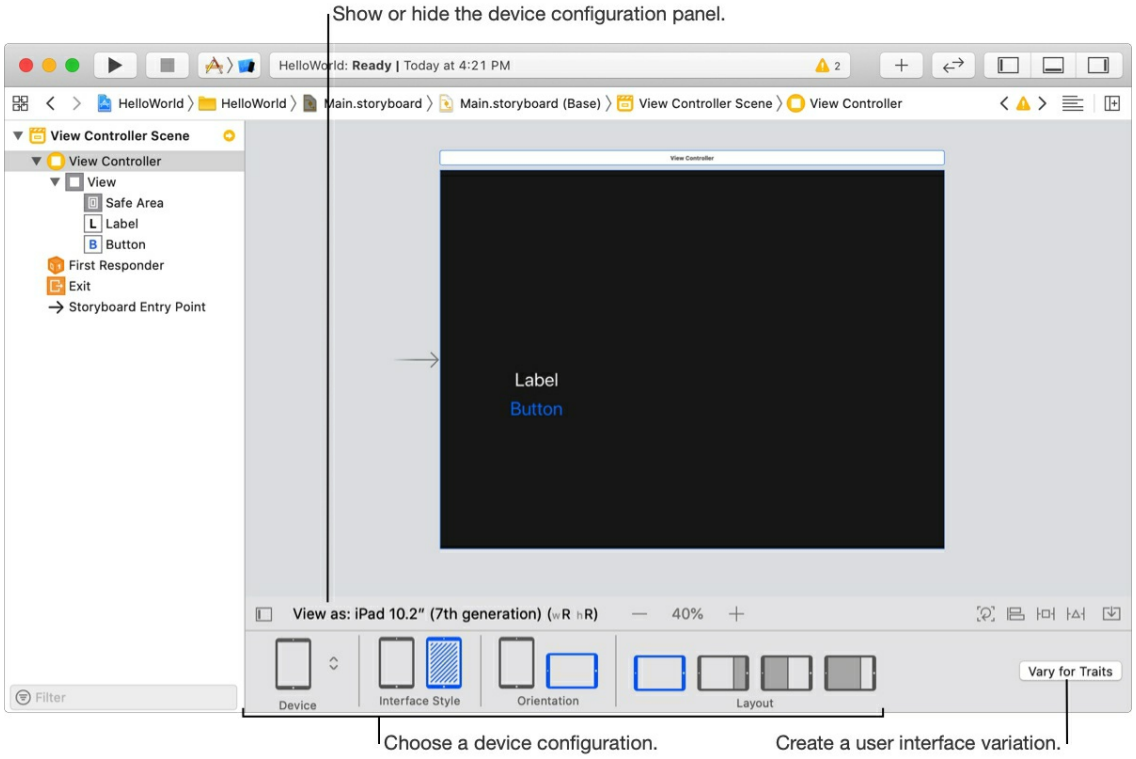
It is very easy to write the code that implements objects' behavior as you add them to the user interface. To access the implementation file for the view controller, select Automatic followed by the filename of the class implementation in the jump bar. Then visually connect your code to the user interface object using the Interface Builder.

If you wish to reference an interface object in your code, simply add an outlet connection by control-dragging from the object on the canvas to the code in the source editor where property declarations are allowed. To add an action method that's called when the user interacts with a control, Control-drag from the control to the method implementation section of the implementation file.

You can remove and modify connections by selecting and either selecting View > Inspectors > Show Connections Inspectors to open the Connections inspector (), or control-click to launch the connections panel.

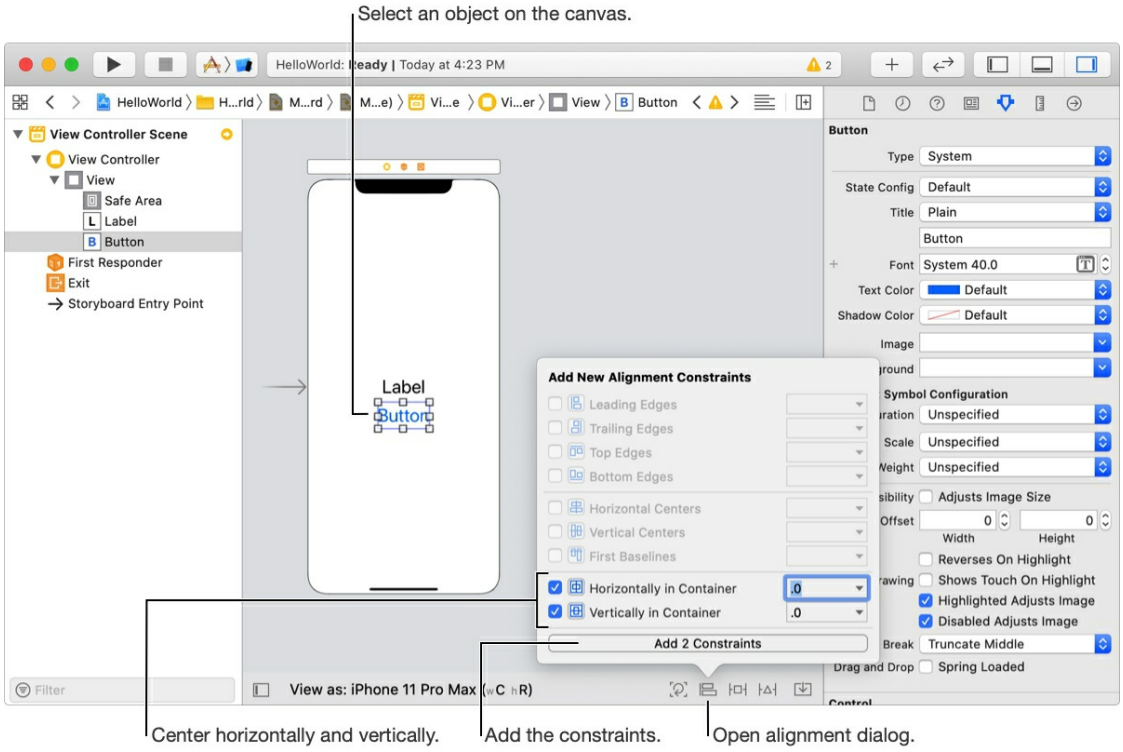
Step 5: View the UI using different device configurations and create variations as needed

First utilize the “View as” button below the canvas to view the user interface using different device configurations that you think most of the end-users of your app will be using. Then create variations of the user interface by tapping on the “vary for trait” at the bottom right.



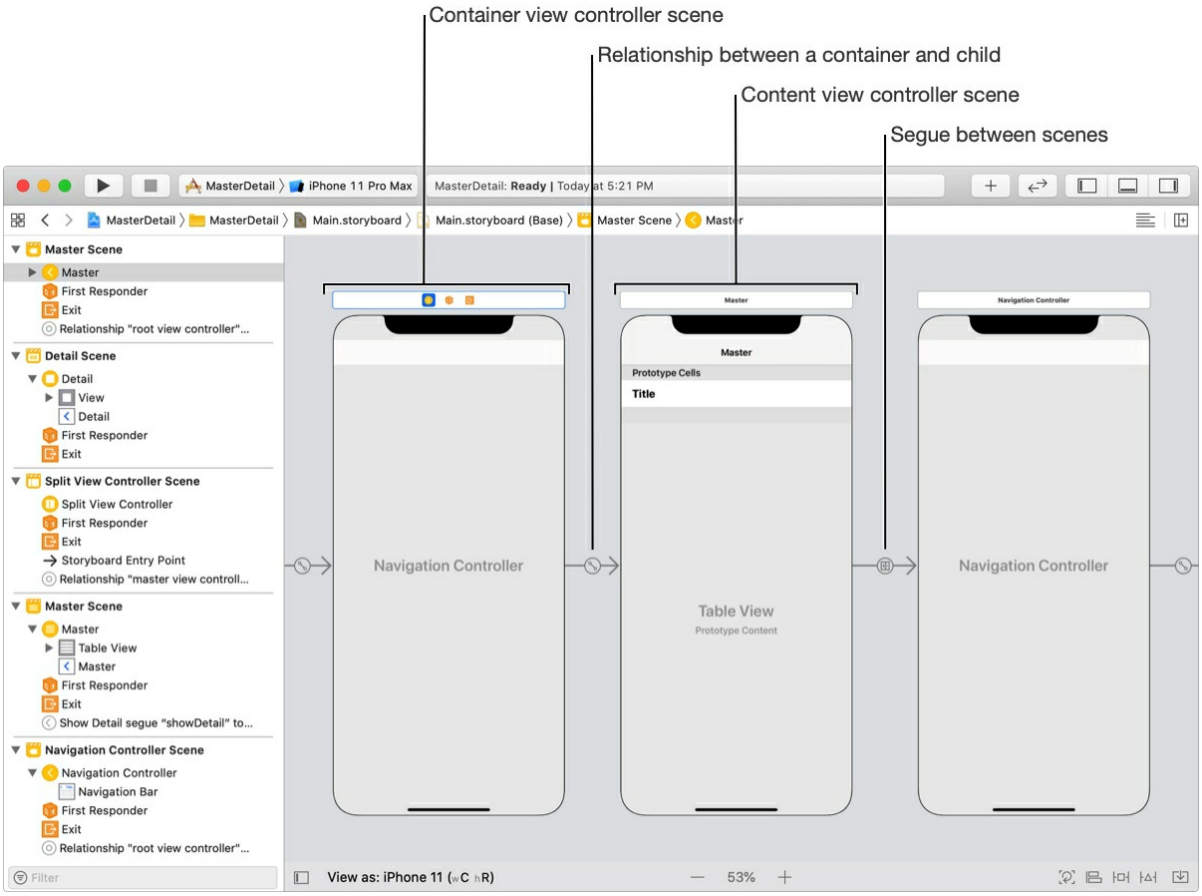
Step 6: Define layout constraints for your app’s user interface

Use Auto Layout constraints to set rules for how the objects should scale and reposition if you observed that the objects in the canvas don’t appear in the location you expect when you change the device configurations. You can add distance and alignment, and size and position constraints. Then use the tools to find and resolve Auto Layout errors and warnings.



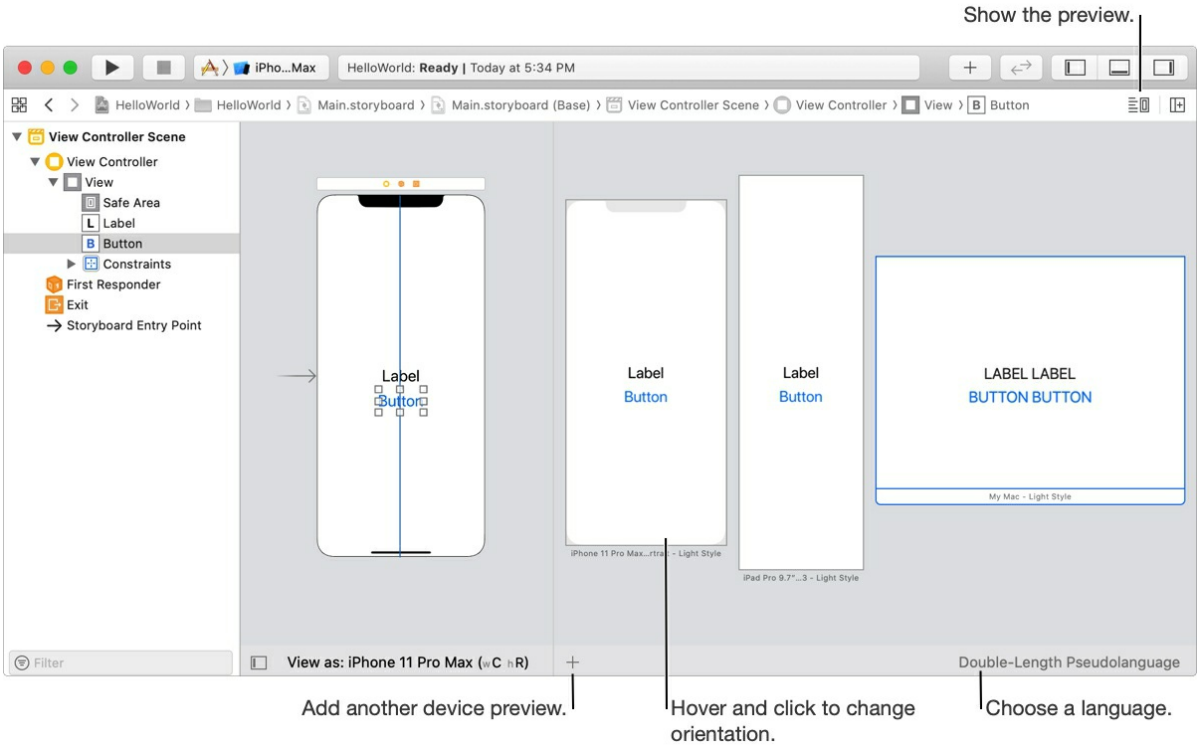
Step 7: Design the user interface of your app with storyboards

Graphically lay out the user's path through your app in a storyboard consisting of scenes, segues that connect the scenes, and controls to trigger the segues. First add scenes and views, then add segues between them.



Step 8: Preview your layout

The assistant editor can be used to preview your layout in different device configurations. For iOS apps, switch between portrait mode and landscape mode, then select different device families. For macOS apps, you can choose either the Dark Appearance or Light Appearance to preview the layout. If you add localization to your app, you can choose a language from a pop-up menu. If you don’t have localization yet but want to see how your layout handles different string lengths, choose Double-Length Pseudo Language from the menu.



Add user interface objects to the canvas

Much of your user interface can be laid out with Views, standard windows and controls from the object library. The objects you found in the library have been tested to work properly and meet Apple's human interface specifications.

1. In the Project navigator, click on the user interface file and then open the library by clicking on the Library button (+) in the toolbar. The library will be launched in a separate window. Alternatively, you can option-click the button to open the library in a persistent window.

2. In the library toolbar, choose the Object library ( ) and then drag any object to the canvas from the library.

In the canvas, the Interface Builder will highlight a valid destination in blue. Properly align and reposition your object by using the guidelines.

You can alternatively drag the object to the outline view. (Show the outline view by clicking on the Show Document Outline button ( ) located at the lower-left corner of the canvas.)

3. In the inspector area, edit the attributes of the object.

Connecting objects to codes

Add an outlet connection to send a message to a UI object

To allow your code to be able to send messages to a user interface object, simply add a connection from the user interface object to a special property in your class called an *outlet*. For instance, if you want to set the text of a label programmatically, simply add a connection from the label in the user interface file to a label outlet in your code. The interface Builder both adds the declaration for the outlet to your class and connects the instance of your class to the outlet.

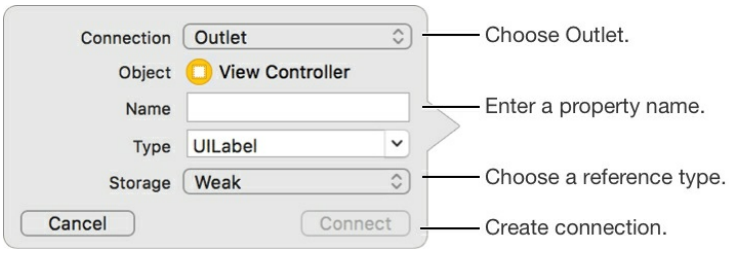
1. In Interface Builder, launch the assistant editor by choosing View, tap on Assistant Editor and select “Show Assistant Editor.”
2. Deploy the jump bar located at the top of the assistant editor to choose the implementation file of the object that will send the messages.
3. Control-drag from the object, in the outline view or in the canvas, to the code in the assistant editor.

Xcode will tell you where you can insert an outlet declaration in your code.

4. A popover will be displayed where you can choose Outlet from the Connection menu, fill in the property name, and select a storage reference type.

Use a reference type of strong for objects that are not implicitly retained such as gestures and array controllers.

The instance of the class that will be connected to the outlet appears in the Object field.



5. Click Connect.

Interface Builder will add the declaration for the outlet to the class. Outlets are defined as IBOutlet properties. The IBOutlet keyword informs Xcode that this particular property can be connected to your user interface file.

6. Utilize the outlet property in your code. Now, you can get and set the properties of the object in your code.

Add an action connection to receive messages from a UI object

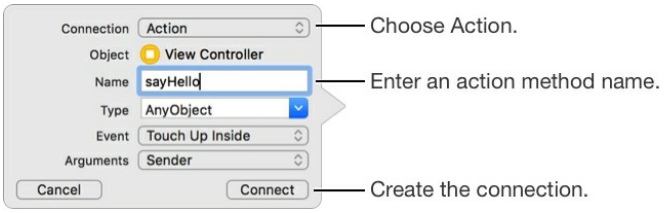
Your object will receive messages from a control if the object is the target of an action specified by the control. For instance, when the user clicks on a particular button, an action message is sent by the button to a target. When an action connection is added, the interface Builder will add an action method to your class and sets the target for the control to an instance of your class. Then you are responsible for implementing the action method.

1. In Interface Builder, launch the assistant editor (click on View, tap Assistant Editor and click on Show Assistant Editor).
2. Use the jump bar located at the top of the assistant editor to choose the implementation file of the object that you want to receive the action message.
3. Control-drag from the control, in the outline view or in the canvas, to the code in the assistant editor.

Xcode will always indicate the part where you can insert an action method in your code, or if you hover over a method, Xcode will tell whether it can be the action for the target.

4. In the displayed popover, select Action from the Connection menu,

input the name of the action method, and select an event from the Event pop-up menu.



5. Click Connect.

In the implementation file, Xcode will insert a code snippet for the action method. The IBAction return type is a special keyword showing that you can actually connect the instance method to your user interface file. Xcode will set the target of the control to an instance of your class (for instance, a view controller object), and sets the control's action to the action method selector. The action message will be set to your object at runtime when the specified event occurs.

6. Implement the action method. It is your duty to write the code for the action method.

Connect from an object to code

Connecting an object to source code requires you to add the declaration in the outlet and the class or action connection at the same time. An object can also be connected to an existing method in the source code.

1. In Interface Builder, open the connection panel by control-clicking an object.
2. In Interface Builder, launch the assistant editor (click on View, tap Assistant Editor and click on Show Assistant Editor).
3. Use the jump bar located at the top of the assistant editor to choose the implementation file for the outlet or action.
4. In the connections panel, drag from a connection well (open circle on the right side of an outlet or action) to the code in the assistant editor.

Xcode will always indicate the part where you can insert an action method in your code, or if you hover over a method, Xcode will tell whether it can

be the action for the target.

Connect from one object to another

1. In Interface Builder, open the connection panel by control-clicking an object.
2. In the connections panel, drag from a connection well to the other object in the outline view or canvas.

   Xcode will always indicate the part where you can insert an action method in your code, or if you hover over a method, Xcode will tell whether it can be the action for the target.

3. Select an event for the action if you drag from an action method in the displayed dialog.

CHAPTER TEN

Build and run your app

You can use real devices or simulated devices to build and run your applications. For macOS applications, your Mac will be the device for this

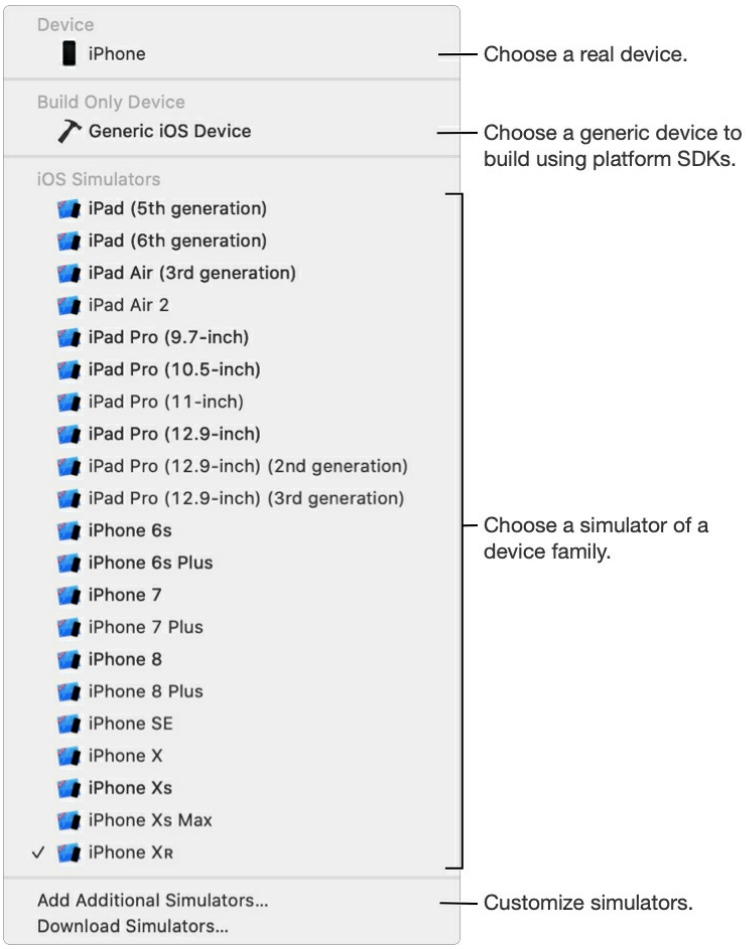
purpose. When you run your app through Xcode, you will see a debugging session opening automatically in the debug area.

1. From the scheme pop-up menu in the toolbar, select a scheme.

Xcode will build a scheme for each target in your project if you have created a project from a template.

2. Select a run destination from the scheme pop-up menu.

The *run destination* is the one that will determine where your app will run after it has been built. For macOS apps, one default destination called My Mac exists. For iOS, tvOS, and watchOS apps, you can choose a device connected to your Mac, a simulated device, or a wireless device paired to Xcode.



3. Build, run and debug your app by clicking on the Run button in the toolbar.

The “View activity” area in the toolbar displays the progress and result of your app building. If the app building has not been successful, you will see “Failed” bolded in the “View activity” area.

On the other hand, if your app building has been successful, your app will open on the simulator or on the device and a debugging session will be opened in the debug area.

4. If you get a warning message or an error, tap on the corresponding red or yellow icon in the “View activity” area.

The issue shows in the Issue navigator displaying the line of code where the error or warning occurred. (The Behavior preferences can be used to customize the behavior of some alerts.)

5. There is a stop button in the toolbar that you can use to stop a build process or to put a stop to the debugging session.
6. If the app building is successful and the app launches, the debug area can be used to control and inspect your running application.

You can only use a real device or a generic device as the run destination to create an archive for distribution. You will not be able to create an archive with simulator SDKs.

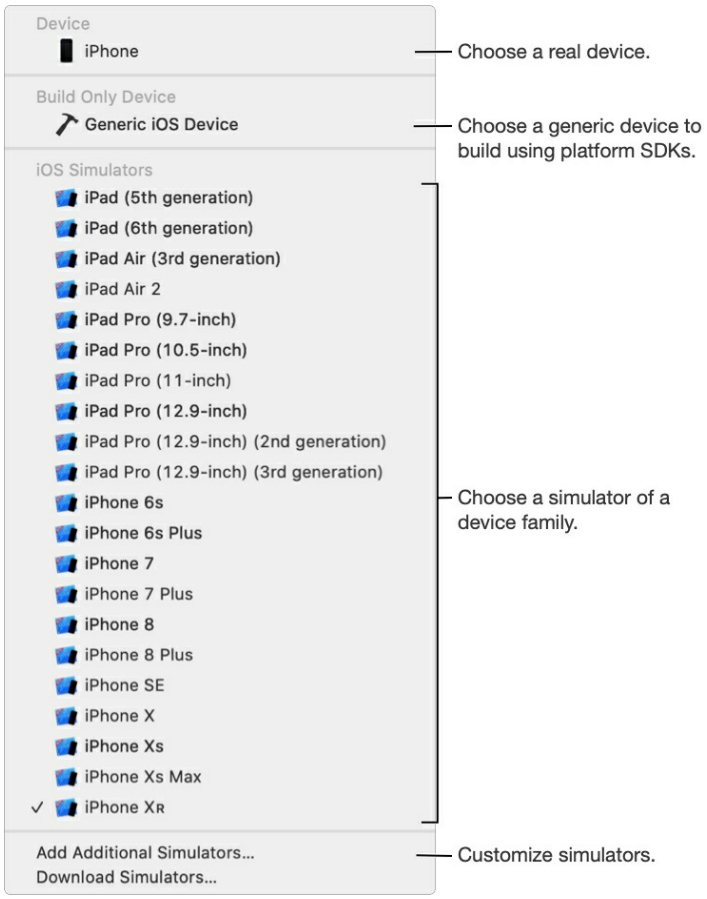
You can rebuild all the files in a project by choosing “Product,” select “Clean Build Folder,” and then choose Product > Run or Product > Build For > Running.

Run an app on a simulated device (iOS, tvOS, watchOS)

For iOS, tvOS, and watchOS apps, your app can be run and tested comfortably with a simulator—a simulator is a developer tool built with Xcode that simulates devices.

1. From the scheme pop-up menu in the toolbar, select a scheme and a device family under [Platform] Simulators.

Optionally, you can configure your desired device family by clicking on the “Add Additional Simulators.”



If you are building apps for watchOS, select the WatchKit App target as a scheme, and select an Apple Watch simulator.

2. Tap on the Run button in the toolbar.

If the app has been built successfully, the Simulator will open and run your app in the simulated device. For watchOS apps that need an iOS app to function, you will see the Apple Watch and the iOS simulator.

Alternatively, your apps can be run on real devices connected to your Mac or on a wireless device paired with Xcode.

Run an app on a device

All iOS apps, tvOS apps, and watchOS apps need to be code signed using a provisioning profile (A *provisioning profile* is a system profile that can be used to launch one or more apps on devices and use certain services.) to launch on a device. macOS apps that utilize certain app services need to be signed to launch on your Mac too.

If you activate automatic signing (recommended) – (*Automatic signing* is a target setting that enables Xcode to manage signing assets for you. The signing settings are found in the General pane under the heading signing in the project editor. To activate automatic signing, choose "Automatically manage signing."), Xcode will create the necessary signing assets for you in your developer account. If the team belongs to a developer program, you will be required to explicitly register the device before you will be able to run the app. For macOS apps, you register the Mac running Xcode.

Before you continue, add your Apple ID account and assign the target to a team. Multiple ID accounts can be added and the account can belong to multiple teams. If you have not added your Apple ID, follow the steps below to do so;

1. Tap on the Add button (+) located in the lower-left corner.
2. A page will be displayed where you can choose Apple ID. Click on "Continue."
3. You will get another page where you can input your Apple ID, tap "Next" and you will be able to enter your Apple ID password on the next page. When you are done, tap "Next."

If the registration is successful, your Apple ID will be shown in the column on the left and the teams that the Apple ID belongs to will appear on the right. The table displays the team name and your program role. If you have not yet joined the Apple Developer program, your team name will be your first name and last name and the name "Personal Team" will be enclosed in front of your name. Refer to the previous chapters on how to join and register for the Apple Developer program.

On macOS 10.11 and later, if you have previously enabled the two-step verification for your Apple ID, you may be required to enter an additional verification code. On earlier operating systems, you may be required to

input an app-specific password.

4. Tap on the “Create Apple ID” in the lower-left corner of the page if you don’t have an Apple ID. Once an ID has been created for you, repeat these steps to add your Apple ID.

Follow the steps below to run an app on a device;

1. For iOS, tvOS, and watchOS apps, connect the device to your Mac. For watchOS apps that are dependent on an iOS app, connect an iPhone that is paired with an Apple Watch.

You may have to actually wait for the Xcode to enable the device before it shows in the scheme menu for the next step. For iOS and watchOS apps, unlock your device’s screen and trust the computer.

2. Select a scheme and your device under the Device section from the scheme pop-up menu in the toolbar.

For watchOS apps, select the WatchKit App target as your scheme and the Apple Watch as your run destination. For watchOS apps that depend on an iOS app, both the name of the Apple Watch and iPhone will appear in the menu.

For macOS apps (including a Mac version of an iPad app), select My Mac as the run destination. For the iPad version, choose an iPad device under Device.

3. If the device shows under Unavailable Device in the scheme menu, move the mouse over the device, check the reason for this, and fix the problem.

For instance, if the operating system version is lower than the deployment target, improve the operating system version on the device by upgrading the OS, or better still, you can change the deployment target in your project.

4. In the project editor, tap on “Signing & Capabilities,” display the Signing settings and then tap “Register Device(s)” under Status.

The “Register device” button will not show if you had previously registered

your device.

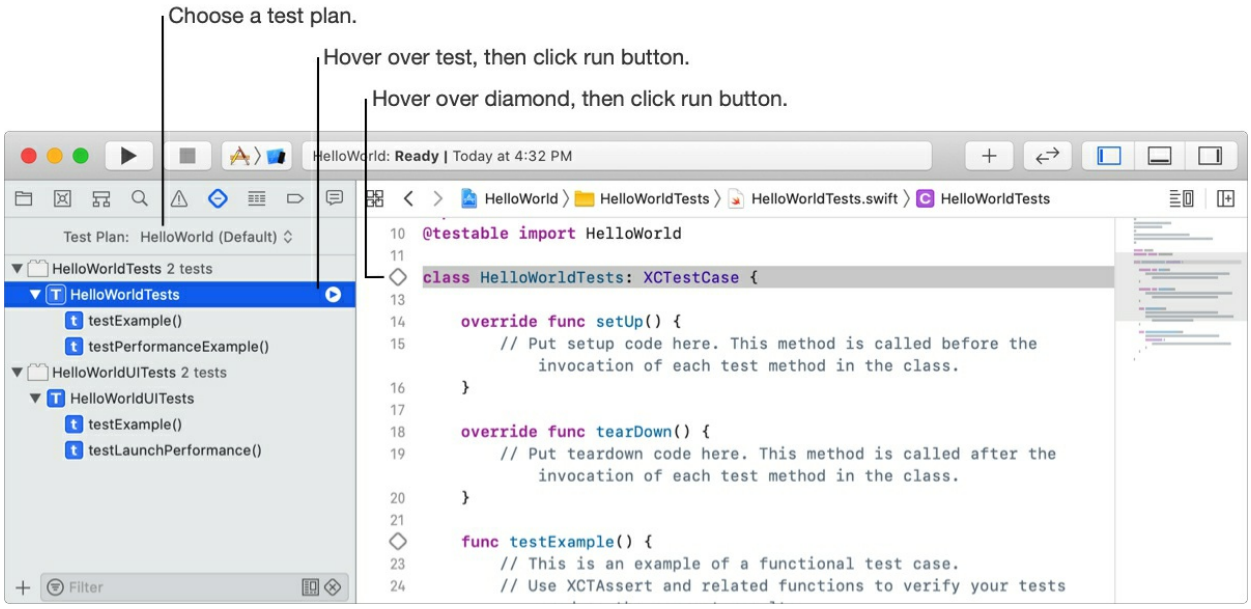
5. In the toolbar, tap on the Run button.

If the product builds successfully, Xcode will install and launch your app on the device.

Run UI tests and unit tests

You will be able to check the behavior and performance of your code by running UI test and Unit test using the Test navigator.

Note: If you had previously converted your scheme to use test plans, you need to first choose the test plan from the Test Plan pop-up menu and then follow the steps below;



Run all tests in a scheme or test plan: Click on “Product,” and then tap on “Test.” You can alternatively click and hold on the Run button in the toolbar and then choose Test.

Run all tests for a test target or test class: In the Test navigator, place your mouse pointer over any test class or test target and then tap on the run button that will show.

Run an individual test: Place your mouse pointer over any test icon or status icon and then tap on the run button that shows.

View the source code for a test: Choose a test in the Test navigator to

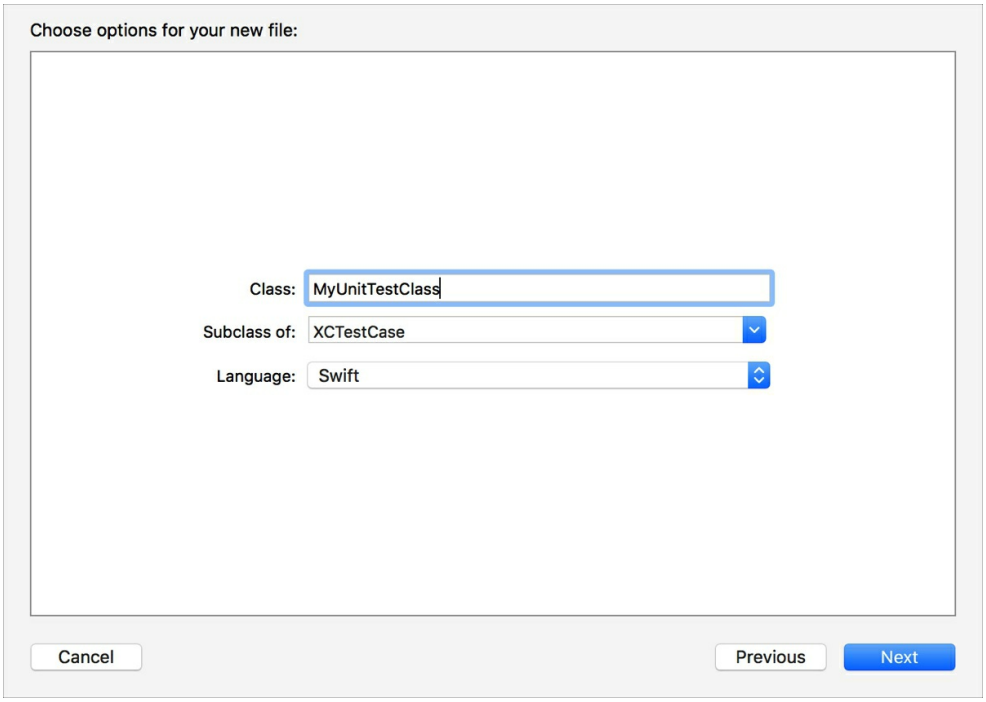
access its code in the source editor.

Alternatively, you can decide to run tests from the source editor. Move your mouse over a diamond icon that shows in the gutter next to a test class or a test method and then tap on the run button that appears.

Add a test class to a project

You can expand the scope of testing in a project with new methods by adding a test class.

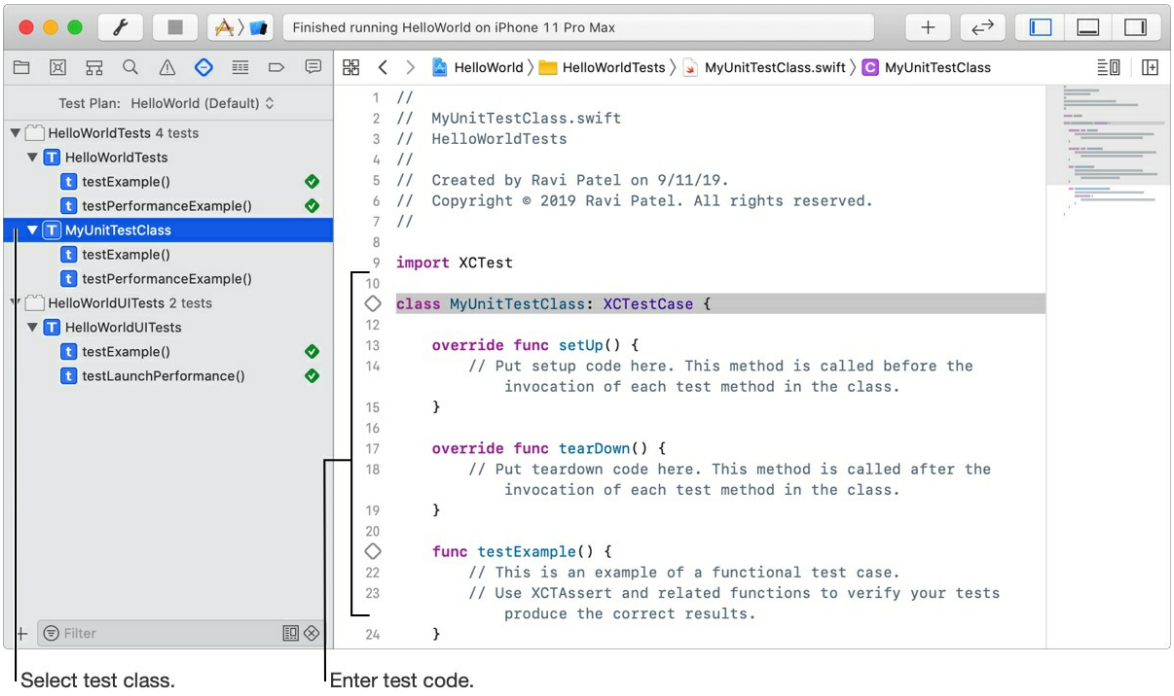
1. Click on the Add button (+) located at the bottom of the Test navigator.
2. Select “New UI Test Class” or “New Unit Test Class” from the pop-up menu.
3. Input the class’s name inside the class field.
4. Select a superclass from the “Subclass of” pop-up menu. Classes are, by default, subclasses of XCTestCase.
5. Tap on the language pop-up menu to select a programming language.



6. Tap on the Next button.
7. Choose a group, destination and test target.

8. Tap on the “Create button.”

A new class will be added to the target and will show in the Test navigator. The new class contains templates for teardown, setUp, testExample, and testPerformanceExample methods.



9. Select the new class in the navigator and enter the test code in the source editor.

Add a test target to a project

You can expand the scope of testing in a project with new methods by adding a test target.

1. Click on the Add button (+) located at the bottom of the Test navigator.
2. Select “New UI Test Target” or “New Unit Test Target” from the pop-up menu.
3. Write a name for the target inside the Project Name field.
4. Select an implementation language from the Language pop-up menu.
5. If you have many projects in your workspace, select a project from the Project pop-up menu. The test target will be created in the project you specify here.

6. From the “Target to be Tested” pop-up menu, select a target to run tests on. This can be any target contained by the project except a test target.
7. Activate any additional options, such as organization name, your team and bundle identifier.
8. Tap on the Finish button.

A new test target with a new class will be added to the project and will show in the Test navigator. The new class contains templates for teardown, setUp, testExample, and testPerformanceExample methods.

9. Choose the new class in the navigator and input the test code in the source editor.

View UI test and unit test reports

The report of your Unit test and UI test run directly in the main window can be accessed. You can add attachments to the report and control the structure of the reports by using the XCTest framework.

1. In the Project navigator, choose a test source code file to launch it in the source editor.
2. In the source editor, Control-Click on a test status icon in the gutter, then select “Jump to Report” from the pop-up menu.

The test report will be displayed in the editor area on the right.

3. You can reveal an activity or test by clicking on the disclosure triangle.

Utilize the XCTest activities APIs to group subtasks.

4. To view a screenshot, launch the assistant editor and choose the screenshot attachment in the report.

Utilize the **XCTest screenshot APIs** to add screenshots to the report.

Creating and distributing a watch-only app

Most Apple Watch’s apps need a companion iOS app to work; that is Watch users must find a way to connect their iPhone to their Watch for full

functioning. Nonetheless, you will still be able to create a watch-only app with no companion app and then offer such an app for sale on the App store on Apple Watch. Watch-only applications are also made available on the iOS App store.

Step 1: Create a watch-only app project

Choose the Watch App template under the watchOS platform when you are creating your Xcode project. You will get a page where you will have the option to include complication or a notification screen.

Your project will contain a [Project Name] target that has project settings but no files. The embedded WatchKit extension target and WatchKit extension apps must have the same bundle ID prefix as the [Project Name] target.

Step 2: Run the watch-only app from Xcode

From the scheme menu in the toolbar, select a real device or an Apple Watch simulator. Since you have created a watch-only app project, you won't see iOS simulators and devices in this menu. Tap the **run button** once you choose a simulator, the Apple Watch simulator will open without a companion iOS app.

Step 3: Distribute and test the watch-only app

You can use Testflight to distribute a beta build version of your watch app or distribute the app to registered devices. Once you create the archive, choose the archive in the Archives organizer and in the inspector, view details about the archive. Since you built a Watch-only app (with no iOS companion), the watchOS state will be "iOS app will be thinned." For a watchOS app that has a companion iOS app, the WatchOS state will either be "iOS app is required" or "iOS app is optional."

Step 4: Distribute the watch-only app through the App Store

Since what you have built is of great quality, the next thing (if you want) is to distribute your product through the App store. Go to "**Add WatchOS app information"** in the App Store Connect Help to create a watch-only app record. In the last page when you are uploading the product to App Store Connect, you will be able to review the targets. For watch-only apps, the targets are: [Project Name], [Project Name] WatchKit App, and [Project Name] WatchKit Extension.

If you select Ad Hoc, Enterprise, or Development as your **distribution method**, you can select an Apple Watch device variant as the **distribution option**.

Support running a watchOS app without an iOS app

For projects you created using the "iOS App with Watch App" template, support can be added for running the WatchOS app without the iOS companion app. To make sure that your app runs independently of the iOS companion app, go to **Creating Independent WatchOS apps.**

This feature has been enabled by default in Xcode 11 and Xcode 12.

1. In the project editor, select the [Project Name] WatchKit Extension target and then tap on General.
2. Under Deployment Info, check the "Supports Running Without iOS App Installation" box.

CHAPTER ELEVEN

LOCALIZING YOUR APPS

Localization is the steps involved in making your apps adaptable to many languages. In App Store Connect, you will be able to select **territories** where you want your apps to be available on the App store. To select **territories** where you want your apps to be available on the App store, follow the steps below;

Note: All countries of the world are selected here by default, but you can deselect countries where you don't want your apps for sale.

1. Select your app from "My Apps," A page will be launched under the **"App store"** tab as shown below.
2. Tap on **"Pricing and Availability"** from the sidebar. You need to set pricing for your app before you will be able to select countries where

you want your app to be available.

3. Tap “**Edit”** under Availability.
4. You will get a dialog where you can choose the regions or countries where you want your product to be available in;

- *Select all countries or regions:* Select All.
- *Choose specific countries or regions:* Click on the checkbox next to the regions or countries that you want to include and then un-select the checkbox next to the countries or regions you don’t plan to add.
- *Automatically add new App Store regions or countries:* Select the New Countries or Regions checkbox that you see in the upper-left corner.

Tap on “Done” located at the end of the dialog and then select “Save” from the upper-right end.

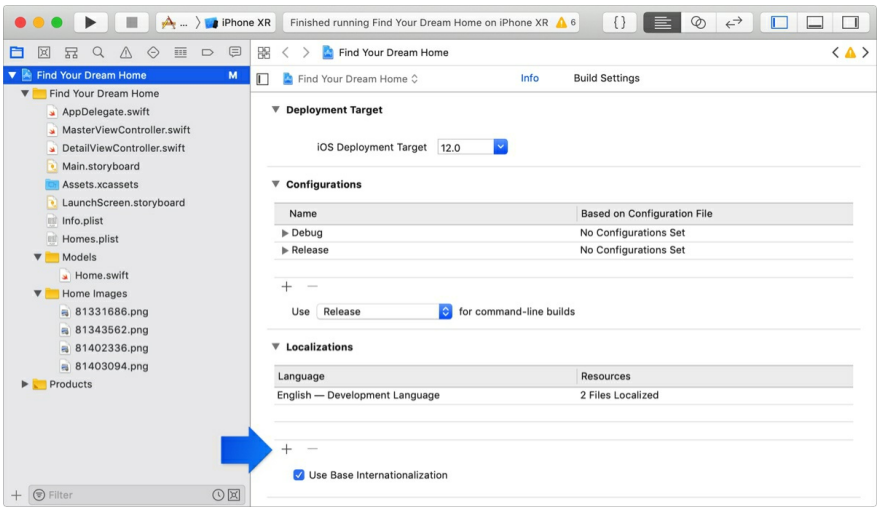
Once you have successfully added territories to your app, the app can also be localized in XCode to enable users utilize the app in their respective local language, culture and region.

To prepare the application for proper localization, you will have to add the languages that you intend to support first. Use the below steps as guides to add languages to your app;

Adding languages supported by apps in the App Store

1. Choose the project in the project navigator, and then tap on “**Info.”**
2. Under Localizations, click on the Add button (+) and then pick a language from the displayed pop-up menu.

The displayed pop-up menu features the language name with the language ID in parenthesis—for instance, Japanese (ja), German (de), and Arabic (ar). For dialects or scripts, the region will appear in parenthesis—for instance, German (Switzerland). The Other submenu (located at the lower end of the menu) has more regions and languages.



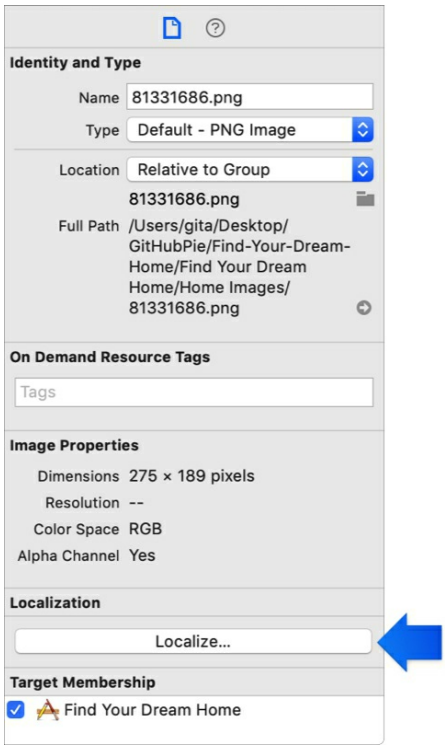
3. In the screen that shows, deselect the resource files that you don't plan to localize for this language, and then select Finish.

Once a language has been added successfully, you can proceed to **mark the resources** that you wish to vary for this language.

Make a resource localizable

Any type of resources can actually be localized, including audio files and images. For instance, an image that is culturally sensitive for a particular region or language can be added. Proceed with below steps;

1. Choose the resource in the Project navigator.
2. Select **"Localize"** in the inspector under Localization.



3. You will get a dialog box where the development language for the resources can be selected. Click **"Localize."**
4. In the inspector, under Localization, choose the languages that you desire for the resource to be localized in.

Once you have been able to localize your resources, you can then **export localizations** for those languages that you want to support and give the language-specific **Xcode Localization Catalog (xcloc) folders to localizers**.

Export localization

1. In the Project navigator, select the project and then select Editor and tap on Export For Localization.
2. You will be shown an export dialog where you will input a suitable name for the folder, choose a location, and select the languages you plan to add under Localizations and then tap "**Save."**

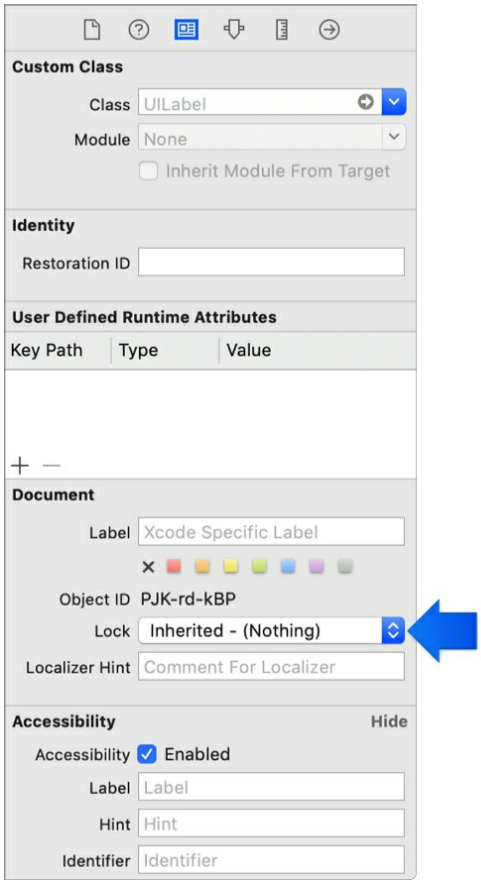
You can continue building your app in the base language and **lock views** optionally while you await localization, and you can then **import localization (**which is the xcloc folders that have actually been translated).

Proceed with the below steps to lock views;

Lock a single view

1. Choose a user interface file in the Project navigator.
2. Choose the view in the Interface Builder.
3. In the Identity inspector, choose a locking level from the pop-up menu under Document.

- *Inherited - [(locking level)]:* Utilize the locking level of the parent view.
- *Nothing:* Don't lock any property. All the properties will be editable.
- *All Properties:* Lock all properties.
- *Localizable Properties:* Lock localizable properties, like user-facing text and size.
- *Non-localizable Properties:* Lock non-localizable properties (make user-facing text and size properties editable).



Ensure that your app is duly tested in all of the language that you have approved. After you have successfully imported localizations, you can then test your app in the regions and the languages you have included.

CHAPTER TWELVE

GETTING YOUR APPS TO THE STORE

App thinning (For iOS, tvOS and watchOS)

All of the apps that are submitted to the app have been tailored – with minimal footprint - to the capability of the users' devices and their devices' operating system. This is the guiding principle with which the App Store and operating system optimize the installation of iOS, watchOS and tvOS apps. This selective optimization, called *app thinning*, gives users the chance of creating apps that use almost all of the features of the device hosting them, accommodate any future updates that can be done to the app and also occupy lower disc space on the device. Better user experience is achievable with faster app download and creation of more disc spaces for other contents and apps. It is not good if an iOS app takes all of the disc space on users' iPhone to install.

Slicing (for iOS, tvOS)

Slicing is the act of building and delivering different forms of the app bundle for different operating system versions and target devices. A *variant* of an app features only the resources and executable architecture that are essential for the target operating system version and device. As time goes on, you can

always update and provide a full model of your application on the App store. Different variants of the apps you sent to the Store will be created and delivered by the App Store based on the operating system version and the devices supported by your app. The assets catalogs can be used to allow the App store choose images, GPU resources and other data appropriate for each variant. When an app is installed by the user, the precise app variant that will be supported by the user's device and operating system will be downloaded and installed for the user without the user having to worry about anything.

During app building, Xcode simulates slicing for you so that you can locally create and test different app variants. Xcode helps to automatically slice your app when you make and run your app on a real device or in a simulator. When you create an archive, the full version of your app will be included in the Xcode, and various versions of your apps can be exported from the archive.

Note: Only devices running iOS and tvOS 9.0 and later will support sliced apps. For users whose operating system is not OS 9.0 and above, the App store will bring universal variants for them. Universal variants are also delivered through apps bought in volume through the Apple School Manager or Apple Business Manager, Mobile Device Management (MDM), or apps downloaded by using iTunes 12.6 or earlier.

Bitcode

Bitcode is just an intermediate rendition of a compiled program. Apps that you have put on App Store Connect that contain bitcode will be compiled and linked on the App Store. When you include bitcode, your app binary (a file that contains machine codes which are executable by the computer) can be re-optimized by Apple in the future without necessarily uploading a fresh version of that app to the App store.

Bitcode is the default for iOS apps, though optional. For apps that run on watchOS and tvOS, bitcode is very much required.

To prevent Apple from accessing your App's symbols, Xcode usually hides your app's symbols by default. You have the chance of including app symbols when you submit an app to the App Store Connect. Including symbols will enable Apple to give crash reports for your app when the app is distributed with TestFlight or with the App store. You don't have to upload

app symbols if you are planning to collect crash reports by yourself. All that you have to do is to download the bitcode compilation dSYM files once you have successfully distributed your app.

On-Demand Resources (iOS, tvOS)

On-demand resources are resources—like sounds and images—that you can tag with keywords and request in groups, by tag. These resources are hosted on Apple server by the App store which also manages the download for you. The App Store also slices on-demand resources, further optimizing variants of the app.

On-demand resources help provide a topnotch user experience:

- Apps downloading become especially faster since apps sizes are now smaller. This improves user experience.
- On-demand resources will be downloaded in the background, when needed, while the users still continue to explore your app.
- The operating system removes on-demand resources when they are no longer needed and when devices' disk space is low.

For instance, an app may split resources into levels and asks for the next level of resources only when it anticipates that the user will move to that level. Similarly, the app can request In-App Purchase resources only when the user makes the corresponding in-app purchase.

Note: You will need to host the on-demand resources by yourself if you distribute your app to registered devices (outside of the App Store).

Distribute an app through the App Store

The processes involved in getting your apps to the end-users are not exactly tedious and mind-boggling. Your apps must be tested and monitored on real and simulated devices in the Xcode before you ever think of distributing them through the App store. The apps that are worthy to be on the App store are apps of good quality and they must be user friendly.

If you distribute your app using TestFlight, you need to carry out some of the below steps before you can finally distribute the final product. Once the steps

have been completed, the app will be uploaded to App Store Connect.

Step 1: Prepare your app for submission

Navigate to “App Review” to review the guidelines concerning App Store and human interface. The procedures you need to follow to prepare your watchOS apps for submission have been discussed previously. For instance, an app store icon must be provided. In case you have not included an app store icon, you will be required to add an app icon to your iOS apps by dragging an app icon to the App store iOS well found in the AppIcon image set.

Step 2: Enter additional details in the App Store Connect

You may have to input additional detail in the App Store Connect before you can get to submit your app to App Review. Be careful with the settings you choose here because you won’t be able to edit them once your app has been submitted or released. For instance, you cannot edit the name of the app, subtitle of app, private policy URL etc once your app has been uploaded.

Step 3: Archive, validate, and upload your app

In case you could not distribute your app using TestFlight, prepare your app for distribution and create an archive of your app now, you can validate the archive and fix some validation errors – if any- before you continue. Then upload the app to the App Store Connect and exercise a little patience for the app to pass App Store Connect validation tests.

Step 4: Submit your app to App Review

To submit the build to App Review, see the “Publish your app in App Store Connect” below;

Publishing your app in the App store: Proceed with the guides below to see how you can publish your app in the App store;

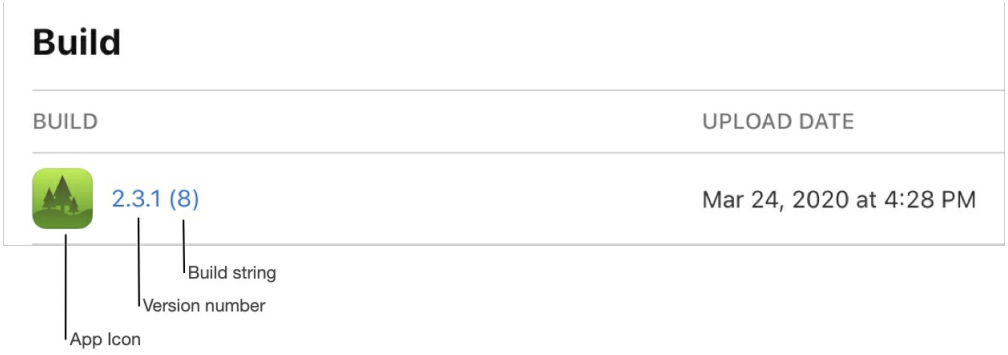
Choose your build

Each app has multiple models, and each app model may have more than one builds. Select which app build you wish to submit when publishing your app

on the App store. Proceed with the guides below to select a build;

1. Scroll to “My Apps,” and select your app. The page launches under the App Store tab.
2. In the sidebar, click on the app version under the platform you decided to select.
3. On the right, move down to the Build section and then tap on the (+) sign (add button) beside “Build.”
4. The Add Build dialog will appear where you can get to select the build that you wish to submit.
5. Tap Done.

You will see the app icon, date & time of upload and the build string in the Build section.



6. Tap “**Save”** in the upper-right end of your screen.

Set pricing and availability

Setting price and availability for your app is among the core things you will have to do once your app has been built successfully. You will need to set a price for your app and choose territories where the app should be available for people to buy (choosing app territories has been explained previously). If you fail to choose territories for the apps, the apps, by default, will be made available in all regions. Your app can also be published as a preorder. Proceed with guides below to fix an appropriate price for your product;

1. Scroll to “My Apps,” and choose your app. The page launches under the App Store tab.
2. Tap on **“Pricing and Availability”** in the sidebar. You will see the

price schedule (displaying the price) on the right.

3. From the price column, choose a price group from the pop-up.

Submit your app for review

To get your app available for review on the App store, you must submit the app for necessary review. The App review process entails reviewing an app submitted to the App store to ensure the app is reliable and is able to pass all required tests. Follow these steps to submit your app for review;

Submit the app

1. Scroll to “My Apps,” and choose your app. The page launches under the App Store tab. .
2. From the sidebar, choose the exact version of the app that you wish to submit for appropriate review.
3. On the right, move down to the Build section to verify that the correct build version for the app has been set.
4. You will receive a Version Release section where you will have the chance to select a release option. You can pick from any of these;

- *Release the app yourself:* Choose “Manually release this version.”
- *Automatically release the app after approval:* Tap “Automatically release this version.”
- *Automatic release the app but no earlier than a specified date:* Select “Automatically release this version after App Review, no earlier than” and input the appropriate date and time below the option.

While releasing your application as a first timer on the App store, you can decide to publish the app as a preorder. Publishing your app as a preorder will remove other release options automatically.

Under the “Phased Release for Automatic Updates” section, you can choose to release app updates in phases if you are submitting a version update.

Under the “Reset Summary Rating” section, you can opt to reset summary rating if you are releasing a version update.

Click on the “**Submit for Review”** in the upper-right end.

If required, attend to the export compliance questions and then upload encryption authorization documents. In the United States, all apps are loaded on Apple servers and the apps are subjected to the United States export laws.

Attend to all the questions on Content Rights. If your app has third-party content, you need to confirm that you have the permission to utilize the third-party content in each territory in which your app is made available. It is your duty to determine and abide by the applicable regulations in each territory.

Provide the details about advertising identifiers in the displayed dialog.

Click on “**Submit.”**

Monitor your app status and attend to review issues

Once the app has been successfully submitted for review, its status will change to “Waiting for review”. If your app has some issues, you will have to check and then give an appropriate reply to the communication. It can take your app about 24hours to be fully available on the App store after approval.

Request promo codes

Once your app has been approved successfully, you can request promo codes to give to users before you finally make your app available on the App Store. The users will be able to use the promo code when they want to buy your app. The promo code can usually allow them to buy your product at a discounted price. The promo code can be shared to the users by email.

ABOUT AUTHOR

Gary Elmer is a programming expert with passion for what he knows how to do best. He has written a lot of beginner and technical guides on programming languages, and will not stop teaching programming until the baby in the womb can have an understanding of it.

Gary read computer science and engineering from the University of Minnesota, USA. Currently, he is undergoing a Masters of Business administration program.