Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Getting Started

In order to get started with making iOS apps, you need to download a program called Xcode from the Mac App Store.

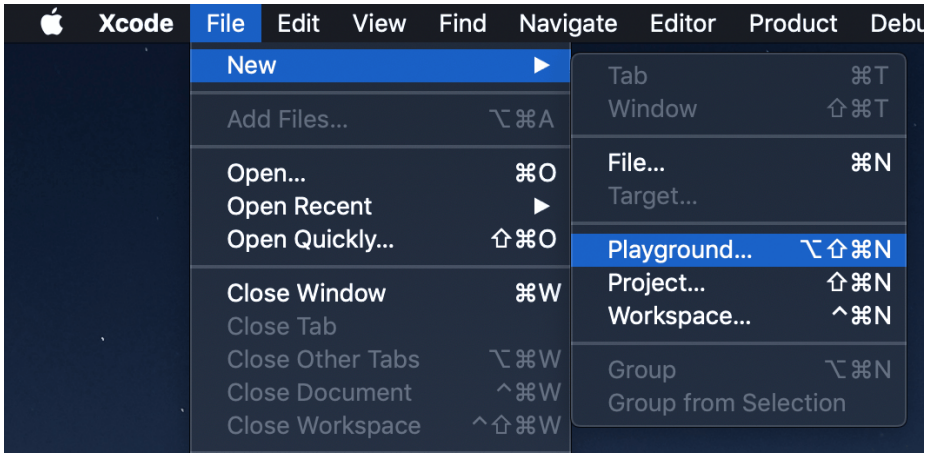
Xcode is what is known as an **integrated development environment** or **IDE** for short.

Start a New Playground

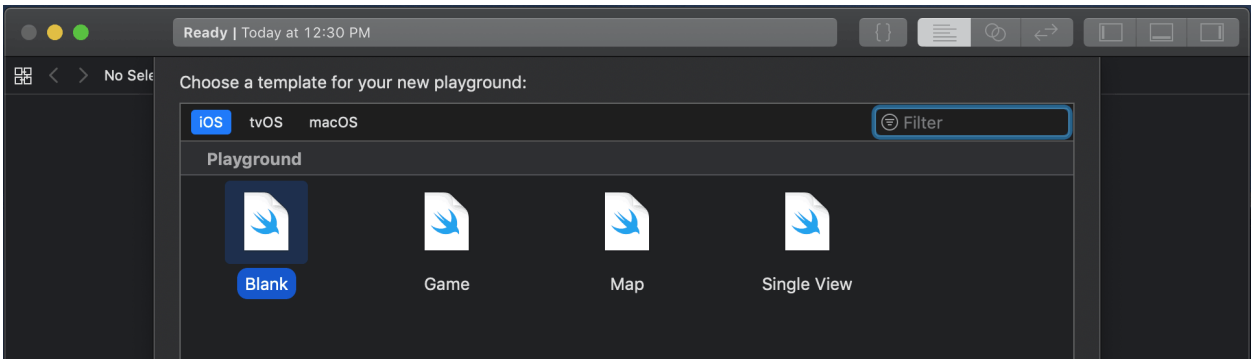
Once you have Xcode installed and launched, you should see the following welcome dialog:

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

If you don't get this welcome dialogue, then you can always go up to the "**File**" menu, go under "**New**," and then choose "**Playground**."



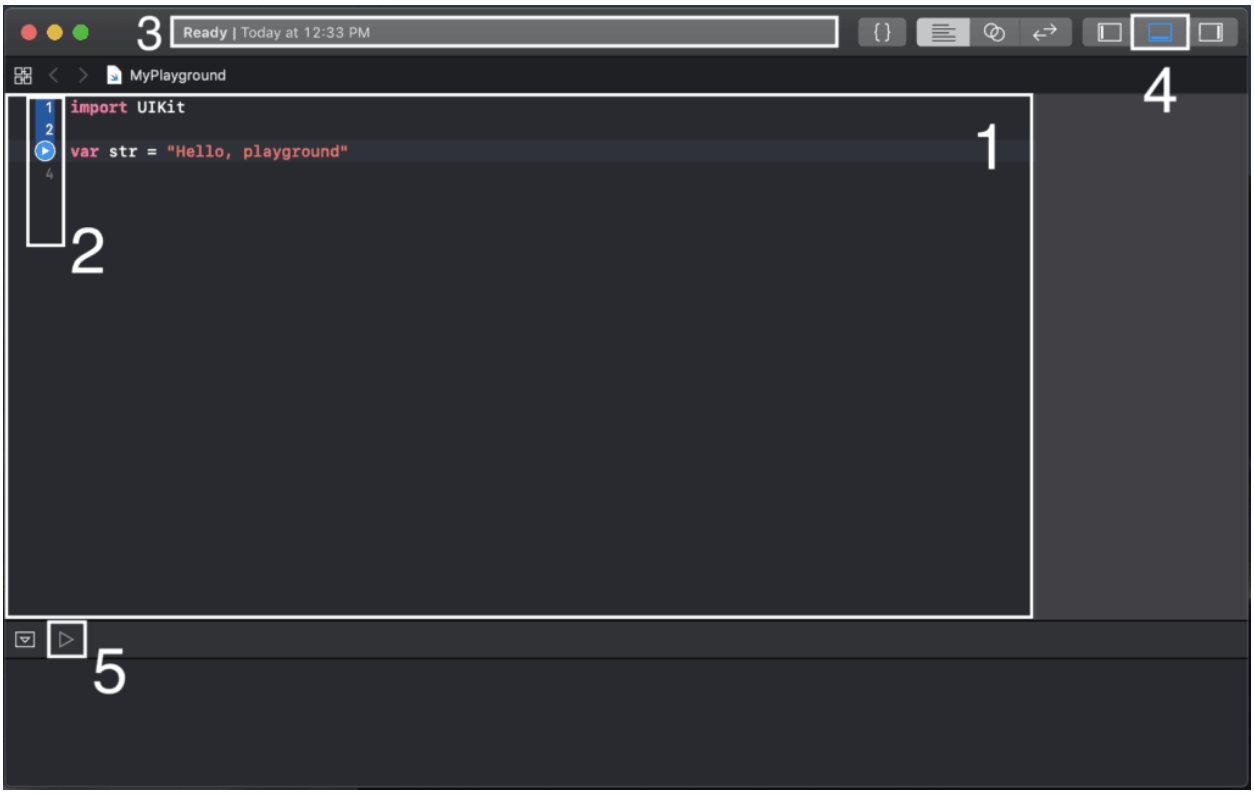
Click on "**Get started with a playground.**" The dialog that pops up allows you to choose what type of playground you want to create



Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Different Elements of the Playground

Here are the most important elements of the playground that you need to focus on for now:



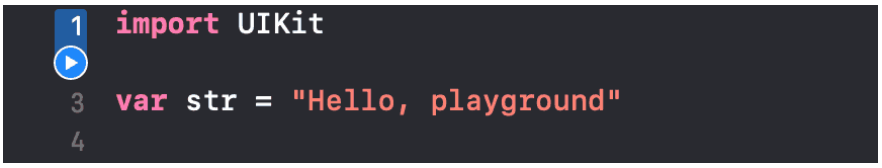
1. **Code editor** : this is where you're going to be typing your Swift code.
2. **Line numbers** : these will help you refer to different lines of code.

*If you don't have line numbers and you want to enable them, then just go to **Xcode** > **Preferences** > **Text Editing** > **Line Numbers**, and you can turn those line numbers on or off.*

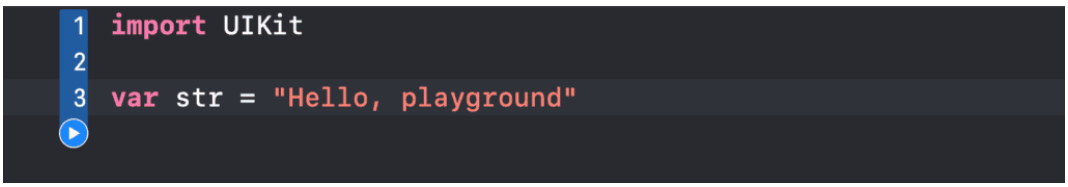
Eng: Khalid Mohamed Mesbah
Tel: 01061085541

You'll also notice that when you hover your mouse over the different lines, a blue play icon follows. Clicking on the play icon executes the code highlighted in blue on the left.

For example, if I hover over line 2 below and click play, Xcode will only run the first line of code.



However, if I hover over line 4 and run it, then Xcode will run all the code up to and including that point.



3. Status bar : tells you the current status of the playground.

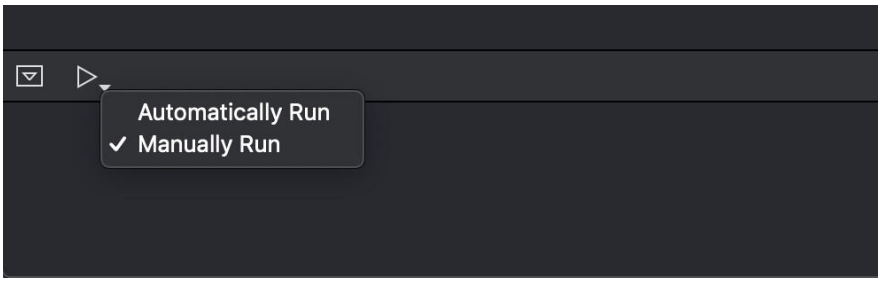
If the status says that it is ready for you, Xcode is ready to accept your code and run it.

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

4. Show/Hide Debug : allows you to hide or show the debug or console area: the place where we're going to be testing our Swift code.

5. Execute Playground : runs all the code in your playground

Holding down the play button gives you two options: "Automatically Run" and "Manually Run."



*The "**Manually Run**" mode means you need to click either click this play button or the blue play icon to run your code.*

*"**Automatically Run**" means Xcode will automatically execute your playground and update the results every time you edit the code.*

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

A Swift Tour

Swift is a new programming language for iOS, macOS, watchOS, and tvOS app development. Nonetheless, many parts of Swift will be familiar from your experience of developing in C and Objective-C.

Constants and Variables

Constants and variables must be declared before they're used.
You declare constants with the `let` keyword
You declare variables with the `var` keyword

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

> NOTE
> If a stored value in your code won't change, always declare it as a constant with the `let` keyword. Use variables only for storing values that need to be able to change.

Type Annotations

You can provide a type annotation when you declare a constant or variable to be clear about the kind of values the constant or variable can store.

Write a type annotation by placing a colon after the constant or variable name, followed by a space, followed by the name of the type to use.

```
var welcomeMessage: String
```

The colon in the declaration means "…**of type**…,"

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

You can define multiple related variables of the same type on a single line, separated by commas.

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

Naming Constants and Variables

Constant and variable names can contain almost any character, including Unicode characters:

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Constant and variable names **can't** contain whitespace characters, mathematical symbols, arrows, private-use Unicode scalar values, or line- and box-drawing characters.

Nor can they begin with a number, although numbers may be included elsewhere within the name.

Once you've declared a constant or variable of a certain type, you can't declare it again with the same name, or change it to store values of a different type.

Nor can you change a constant into a variable or a variable into a constant.

> NOTE
> If you need to give a constant or variable the same name as a reserved Swift keyword, surround the keyword with backticks (`) when using it as a name. However, avoid using keywords as names unless you have absolutely no choice.

You can change the value of an existing variable to another value of a compatible type.

```
var welcome = "Hello!"
welcome = "Bonjour!"
// welcome is now "Bonjour!"
```

Unlike a variable, the value of a constant can't be changed after it's set.

```
let language = "Swift"
language = "Java"
// This is a compile-time error: language cannot be changed.
```

Printing Constants and Variables

You can print the current value of a constant or variable with the print(_:separator:terminator:) function:

```
var welcome = "Bonjour!"
print(welcome)
```

Swift uses string interpolation to include the name of variable as a placeholder in a longer string, and replace it with the current value of that variable.

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Wrap the name in parentheses and escape it with a backslash before the opening parenthesis:

```
print("The current value of welcome is \(welcome)")
// Prints "The current value of welcome is Bonjour!"
```

Comments

Single-line comments begin with two forward-slashes (//):

```
// This is a comment
```

Multiline comments start with a forward-slash followed by an asterisk (/*) and end with an asterisk followed by a forward-slash (*/):

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

Semicolons

Swift doesn’t require you to write a semicolon (;) after each statement

Semicolons are required if you want to write multiple separate statements on a single line:

```
let cat = "🐱"; print(cat)
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Data types

In Swift, there are several different types of data, but these are the most common ones:

1. String – this is just text data. The name basically refers to having a *string* of characters:

```
var aString = "This is a string"
```

2. Int – this is short for “integer.” This type represents whole numbers: positive and negative.

```
var myInt = 108
```

```
var itemsInStock = -20
```

3. Float and Double – these two represent decimal numbers. The difference is that Doubles have more precision than Floats, so they can store longer decimal numbers.

```
let pi = 3.15159265359
```

4. Boolean – Swift shortens this to “Bool.” Booleans can only store either “true” or “false”. They are perfect for when there are only one of two options.

```
var isTVOn = true
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Integers

Integers are whole numbers with no fractional component.

Integers are either signed (positive, zero, or negative)
or unsigned (positive or zero).

Swift provides signed and unsigned integers in 8, 16, 32, and 64 bit forms

You can access the minimum and maximum values of each integer type with its `min` and `max` properties:

```
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
```

Int

Swift provides an additional integer type, `Int`, which has the same size as the current platform's native word size:

- On a 32-bit platform, `Int` is the same size as `Int32`.
- On a 64-bit platform, `Int` is the same size as `Int64`.

UInt

Swift also provides an unsigned integer type, `UInt`,

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Floating-Point Numbers

Floating-point numbers are numbers with a fractional component, such as 3.14159

Swift provides two signed floating-point number types:

- `Double` represents a 64-bit floating-point number.
- `Float` represents a 32-bit floating-point number.

> NOTE
>
> `Double` has a precision of at least 15 decimal digits, whereas the precision of `Float` can be as little as 6 decimal digits. The appropriate floating-point type to use depends on the nature and range of values you need to work with in your code. In situations where either type would be appropriate, `Double` is preferred.

Numeric Literals

Integer literals can be written as:

- A ***decimal*** number, with no prefix
- A ***binary*** number, with a `0b` prefix
- An ***octal*** number, with a `0o` prefix
- A ***hexadecimal*** number, with a `0x` prefix

All of these integer literals have a decimal value of `17`:

```
let decimalInteger = 17
let binaryInteger = 0b10001 // 17 in binary notation
let octalInteger = 0o21 // 17 in octal notation
let hexadecimalInteger = 0x11 // 17 in hexadecimal notation
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Floating-point literals can be decimal (with no prefix), or hexadecimal (with a 0x prefix).

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

Numeric literals can contain extra formatting to make them easier to read

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

Numeric Type Conversion

An `Int8` variable can store numbers between `-128` and `127`, whereas a `UInt8` variable can store numbers between `0` and `255`.

A number that won't fit into a constant or variable of a sized integer type is reported as an error when your code is compiled:

```
let x: UInt8 = -1
// UInt8 can't store negative numbers, and so this will report an error

let tooBig: Int8 = Int8.max + 1
// Int8 can't store a number larger than its maximum value,
// and so this will also report an error
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Integer Conversion

To convert one specific number type to another, you initialize a new number of the desired type with the existing value

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

Integer and Floating-Point Conversion

Conversions between integer and floating-point numeric types must be made explicit:

```
let x = 3
let y = 0.14159
let z = Double(x) + y
// pi equals 3.14159, and is inferred to be of type Double
```

Floating-point to integer conversion must also be made explicit

```
let integerPi = Int(pi)
// integerPi equals 3, and is inferred to be of type Int
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Booleans

Boolean values are referred to as logical, because they can only ever be `true` or `false`.

```
let orangesAreOrange = true
let logged_in = false
```

Type Aliases

Type aliases define an alternative name for an existing type.

You define type aliases with the `typealias` keyword.

```
typealias myType = UInt16

var x: myType
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Optionals

You use *optionals* in situations where a value may be absent.

An optional represents two possibilities: Either there is a value, or there isn't a value at all.

If you define an optional variable without providing a default value, the variable is automatically set to `nil` for you:

```
var name: String?
// name is automatically set to nil
```

> NOTE
>
> Swift's `nil` isn't the same as `nil` in Objective-C. In Objective-C, `nil` is a pointer to a nonexistent object. In Swift, `nil` isn't a pointer—it's the absence of a value of a certain type. Optionals of *any* type can be set to `nil`, not just object types.

Here's an example of how optionals can be used to cope with the absence of a value. Swift's `Int` type has an initializer which tries to convert a `String` value into an `Int` value. However, not every string can be converted into an integer. The string `"123"` can be converted into the numeric value `123`, but the string `"hello, world"` doesn't have an obvious numeric value to convert to.

The example below uses the initializer to try to convert a `String` into an `Int`:

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?", or "optional Int"
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Because the initializer might fail, it returns an *optional* `Int`, rather than an `Int`. An optional `Int` is written as `Int?`, not `Int`. The question mark indicates that the value it contains is optional, meaning that it might contain *some* `Int` value, or it might contain *no value at all*.

You set an optional variable to a valueless state by assigning it the special value `nil`:

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

If Statements and Forced Unwrapping

You can use an `if` statement to find out whether an optional contains a value by comparing the optional against `nil`.

```
var x: Int? = 123

if x != nil {
      print("x has an integer value of \(x!).")
}
// Prints "x has an integer value of 123."
```

Eng: Khalid Mohamed Mesbah
Tel: 01061085541

Optional Binding

Optional binding can be used with `if` and `while` statements to check for a value inside an optional, and to extract that value into a variable.

Write an optional binding for an `if` statement as follows:

```
if let constantName = someOptional {
    statements
}
```

This code can be read as:

```
let possibleNumber = "123"
if let actualNumber = Int(possibleNumber) {
      print("\(actualNumber)")
} else {
      print("The string couldn't be converted to an integer")
}
// Prints "123"
```