iOS

How to develop apps for iPhones and iPads
Welcome to Xcode

Version 8.1 (8B62)

Get started with a playground
Explore new ideas quickly and easily.

Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch or Apple TV.

Check out an existing project
Start working on something from an SCM repository.

Show this window when Xcode launches

Open another project...
Swift
Swift

- Swift features several programming constructs from other languages including ObjectiveC and Smalltalk
- Swift is a multi-paradigm programming environment
- Swift allows type inference e.g. loose typing
- Swift programming statements do not require line ending characters such as semicolons
Our first application

print("Hello, world!")
Playground

```swift
// Playground - noun: a place where people can play

import UIKit

print("Hello, world!")
```

"Hello, world!"
Variables and constants

//Variables
var name: String = "Jane Doe"
var year: Int = 2014
var isFast: Bool = true

//Data type detection
var name = "Jane Doe"
var year = 2014
var isFast = true

//Constants
let name: String = "Jane Doe"
let year: Int = 2014
let isFast: Bool = true
Type Safety and Type Inference

• Swift is type safe
  – Encourages you to be clear about types
  – It performs type checks during compilation
• You don’t have to specify the types explicitly
  – The type is inferred at compilation time depending on the assignments that are made
  – Fewer declarations than with C and Objective-C
  – It’s a good idea to initialize the variable when we declare them
Tuples

let http404Error = (404, "Not Found")
let (statusCode, statusMessage) = http404Error

print("The status code is \(statusCode)"")
print("The status message is \(statusMessage)"")

let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)"")
print("The status code is \(http404Error.0)"")
print("The status message is \(http404Error.1)"")

let http200Status = (statusCode: 200, description: "OK")

print("The status code is \(http200Status.statusCode)"")
print("The status message is \(http200Status.description)"
Optionals

• You use optionals in situations where a value may be absent
  — There is a value and it equals x or there is no value at all (nil)

```swift
let posNum = "123"
let convNum: Int? = Int(posNum)

if convNum != nil {
    print("convNum has an integer value of \(convNum!).")
}

// Without ! it would print “convNum has an integer value of Optional(123)”.

let possibleString: String? = "An optional string."
let forcedString: String = possibleString!

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString
```
Operators

• Nil Coalescing Operator
  – \((a ?? b)\) is equivalent to \(a != nil ? a! : b\)
  – if the optional \(a\) contains a value we take the unwrapped value, if not we take a default \(b\)

• Closed Range Operator
  – \((a...b)\) defines a range that goes from \(a\) to \(b\) (included)

• Half-Open Range Operator
  – \((a..<b)\) defines a range that goes from \(a\) to \(b\) (excluded)
Strings

• Strings are represented by type String
  ```swift
  for character in "Dog!".characters {
    print(character)
  }
  ```
  — String mutability depends on let/var
  — Support for String interpolation

• Swift’s String type is a value type
  — If you create a new String value, that String value is copied when it is passed to a function or method, or when it is assigned to a constant or variable
  — In each case, a new copy of the existing String value is created, and the new copy is passed or assigned, not the original version
Collections

Array

<table>
<thead>
<tr>
<th>Indexes</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Six Eggs</td>
</tr>
<tr>
<td>1</td>
<td>Milk</td>
</tr>
<tr>
<td>2</td>
<td>Flour</td>
</tr>
<tr>
<td>3</td>
<td>Baking Powder</td>
</tr>
<tr>
<td>4</td>
<td>Bananas</td>
</tr>
</tbody>
</table>

Set

Values

- Rock
- Jazz
- Classical
- Hip Hop

Dictionary

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYZ</td>
<td>Toronto Pearson</td>
</tr>
<tr>
<td>DUB</td>
<td>London Heathrow</td>
</tr>
<tr>
<td>LHR</td>
<td>Dublin Airport</td>
</tr>
</tbody>
</table>
var shoppingList: [String] = ["Eggs", "Milk"]
// or var ShoppingList = ["Eggs", "Milk"]

print("The shopping list contains \(shoppingList.count) items.")

if shoppingList.isEmpty {
    print("The shopping list is empty")
} else {
    print("The shopping list is not empty")
}

shoppingList += ["Baking Powder"]
// shoppingList now contains 3 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 6 items

shoppingList.insert("Maple Syrup", at: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list

shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items

var firstItem = shoppingList[0]

var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
Ways to iterate over a Array

```python
for item in shoppingList {
    print(item)
}
for (index, value) in shoppingList.enumerated() {
    print("Item \(index + 1\): \(value\)")
}
```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items

print("I have \(favoriteGenres.count) favorite music genres.")
// Prints "I have 3 favorite music genres."

favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items

if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it."))
} else {
    print("I never much cared for that.")
}
// Prints "Rock? I'm over it."

if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
// Prints "It's too funky in here."

for genre in favoriteGenres {
    print("\(genre)"
}
Set operations

```javascript
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()  // []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()  // [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()  // [1, 2, 9]
```
Dictionaries

var airports: [String: String] = ["YYZ": "Toronto", "DUB": "Dublin"]

print("The airports dictionary contains \(airports.count) items.")

if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary is not empty.")
}

airports["LHR"] = "London"
airports["LHR"] = "London Heathrow"
airports["LHR"] = nil //removing an existing value!

if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
    print("The old value for DUB was \(oldValue).")
}
Ways to iterate over a Dictionary

```python
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}

for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
```
for index in 1...5 {
    print("(index) times 5 is (index * 5)")
}

let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("(base) to the power of (power) is (answer)")

let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("(animalName)s have (legCount) legs")
}
While loops

while condition {
    statements
}

repeat {
    statements
} while condition
let someCharacter: Character = "e"
switch someCharacter {
    case "a", "e", "i", "o", "u":
        print("(someCharacter) is a vowel")
    case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m", "n", "p", "q", "t", "s", "v", "w", "x", "y", "z":
        print("(someCharacter) is a consonant")
    default:
        print("(someCharacter) is not a vowel or a consonant")
}
let count = 3_000_000_000_000
let countedThings = "stars in the Milky Way"
var naturalCount: String
switch count {
    case 0:
        naturalCount = "no"
    case 1...3:
        naturalCount = "a few"
    case 4...9:
        naturalCount = "several"
    case 10...99:
        naturalCount = "tens of"
    case 100...999:
        naturalCount = "hundreds of"
    case 1000...999_999:
        naturalCount = "thousands of"
    default:
        naturalCount = "millions and millions of"
}
print("There are \(naturalCount) \(countedThings).")
... and with tuples

```swift
let somePoint = (1, 1)
switch somePoint {
    case (0, 0):
        print("(0, 0) is at the origin")
    case (_, 0):
        print("((somePoint.0), 0) is on the x-axis")
    case (0, _):
        print("(0, (somePoint.1)) is on the y-axis")
    case (-2...2, -2...2):
        print("((somePoint.0), (somePoint.1)) is inside the box")
    default:
        print("((somePoint.0), (somePoint.1)) is outside of the box")
}

let anotherPoint = (2, 0)
switch anotherPoint {
    case (let x, 0):
        print("on the x-axis with an x value of \(x)")
    case (0, let y):
        print("on the y-axis with a y value of \(y)")
    case let (x, y):
        print("somewhere else at \((x), \(y))")
}
```
func sayHello(personName: String) -> String {
    return "Hello again, " + personName + "!"
}

print(sayHello(personName: "Anna"))

func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}

print(halfOpenRangeLength(start: 4, end: 9))

func sayHelloWorld() -> String {
    return "hello, world"
}

print(sayHelloWorld())

func sayGoodbye(personName: String) {
    print("Goodbye, \(personName)!")
}

sayGoodbye(personName: "Anna")
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}

let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```swift
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty {
        return nil
    }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}

if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
```
Argument labels

```swift
func sayHello(person: String, anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}

print(sayHello(person: "Bill", anotherPerson: "Ted"))

func sayHello(to person: String, and anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}

print(sayHello(to: "Bill", and: "Ted"))

func sayHello(_, person: String, _ anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}

print(sayHello("Bill", "Ted"))

Trying to change the value of a function parameter from within the body of that function results in a compile-time error
```
Default values and variadic parameters

```swift
func someFunction(parameterWithDefault: Int = 12) {
    // function body goes here
}

someFunction(parameterWithDefault: 6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12

func arithmeticMean(numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}

print(arithmeticMean(numbers: 1, 2, 3, 4, 5))
// returns 3.0, which is the arithmetic mean of these five numbers

print(arithmeticMean(numbers: 3, 8.25, 18.75))
// returns 10.0, which is the arithmetic mean of these three numbers
```
In-out parameters

• Variable parameters, as described above, can only be changed within the function itself
  – Some limitations

```swift
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```
Function Types (I)

- Every function has a specific function type, made up of the parameter types and the return type of the function

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

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

var mathFunction: (Int, Int) -> Int = addTwoInts

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

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

let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```
```swift
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5) // Prints "Result: 8"

func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}

func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}

var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0) // moveNearerToZero now refers to the stepBackward() function

print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```
Nested Functions

• All of the functions so far have been examples of global functions, defined at a global scope
• Nested functions are functions inside the bodies of other functions
  – They are hidden from the outside world by default, but can still be called and used by their enclosing function
  – An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
  func stepForward(input: Int) -> Int {
    return input + 1
  }
  func stepBackward(input: Int) -> Int {
    return input - 1
  }
  return backward ? stepBackward : stepForward
}

var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
  print("\((currentValue)... "
  currentValue = moveNearerToZero(currentValue)
}
print("zero!")
Closures

• Closures are self-contained blocks of functionality that can be passed around and used in your code
  – Closures in Swift are similar to blocks in C and to lambdas in other programming languages
• Closures take one of three forms
  – Global functions are closures that have a name and do not capture any values
  – Nested functions are closures that have a name and can capture values from their enclosing function
  – Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context
Closure expressions

- Nested functions are a convenient means of naming and defining self-contained blocks of code as part of a larger function

```swift
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

- Closure expression syntax

```swift
var reversedNames = names.sorted(by: {(s1: String, s2: String) -> Bool in
    return s1 > s2})
```
Closures

• Inferring Type from Context
  \[
  \text{reversedNames} = \text{names}.\text{sorted}(\text{by:} \{s1, s2 \text{ in return } s1 > s2\})
  \]

• Implicit Returns
  – Single-expression closures can implicitly return the result of their single expression
  \[
  \text{reversedNames} = \text{names}.\text{sorted}(\text{by:} \{s1, s2 \text{ in } s1 > s2\})
  \]

• Shorthand Argument Names
  \[
  \text{reversedNames} = \text{names}.\text{sorted}(\text{by:} \{0 > 1\})
  \]

• Operator Functions
  – Swift’s String type defines its string-specific implementation of the greater-than operator
  \[
  \text{reversedNames} = \text{names}.\text{sorted}(\text{by: }>)
  \]

• Trailing Closures
  \[
  \text{reversedNames} = \text{names}.\text{sorted()}\{0 > 1\}
  \]
  \[
  \text{reversedNames} = \text{names}.\text{sorted}\{0 > 1\}
  \]
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
Capturing Values

```swift
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}

let incrementByTen = makeIncrementer(forIncrement: 10)
incrementByTen() // returns a value of 10
incrementByTen() // returns a value of 20
incrementByTen() // returns a value of 30

let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven() // returns a value of 7
incrementByTen() // returns a value of 40
```
Classes and Structures

- Classes and structures can
  - Define properties to store values
  - Define methods to provide functionality
  - Define subscripts to provide access to their values using subscript syntax
  - Define initializers to set up their initial state
  - Be extended to expand their functionality beyond a default implementation
  - Conform to protocols to provide standard functionality of a certain kind

- Classes have additional capabilities that structures do not:
  - Inheritance enables one class to inherit the characteristics of another
  - Type casting enables you to check and interpret the type of a class instance at runtime
  - Deinitializers enable an instance of a class to free up any resources it has assigned
  - Reference counting allows more than one reference to a class instance
Conditions for using structures

- They are used to encapsulate a few relatively simple data values
- Encapsulated values will be *copied* rather than *referenced* when you assign or pass around an instance of that structure
- Any properties stored by the structure are themselves value types, which would also be expected to be *copied* rather than *referenced*
- A structure does not need to inherit properties or behavior from another existing type

- Many basic data types such as String, Array, and Dictionary are implemented as structures in Swift
  - This means that they are copied when assigned
First example

```swift
struct Resolution {
    var width = 0
    var height = 0
}

class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}

let someResolution = Resolution()
let someVideoMode = VideoMode()

• This creates a new instance of the class or structure, with any properties initialized to their default values

print("The width of someResolution is \(someResolution.width)")
print("The width of someVideoMode is \(someVideoMode.resolution.width)")

someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
```