Swift is a programming language for iOS and OS X development created by Apple. Designed to coexist with Objective-C and to be more resilient against erroneous code, Swift was introduced in 2014 at Apple's developer conference WWDC. It is built with the LLVM compiler included in Xcode 6+.

The official Swift Programming Language book from Apple is now available via iBooks.

See also Apple's getting started guide, which has a complete tutorial on Swift.

```
// import a module
import UIKit

//
// MARK: Basics
//

// Xcode supports landmarks to annotate your code and lists them in the jump bar
// MARK: Section mark
// MARK: - Section mark with a separator line
// TODO: Do something soon
// FIXME: Fix this code

// In Swift 2, println and print were combined into one print method. Print automatically appends a new line.
print("Hello, world") // println is now print
print("Hello, world", terminator: "") // printing without appending a newline

// variables (var) value can change after being set
// constants (let) value can NOT be changed after being set

var myVariable = 42
let ø Ω = "value" // unicode variable names
let  = 3.1415926
let convenience = "keyword" // contextual variable name
let weak = "keyword"; let override = "another keyword" // statements can be separated by a semi-colon
let `class` = "keyword" // backticks allow keywords to be used as variable names
let explicitDouble: Double = 70
let intValue = 0007 // 7
let largeIntValue = 77_000 // 77000
let label = "some text " + String(myVariable) // String construction
let piText = "Pi = \( ), Pi 2 = \(  * 2)" // String interpolation

// Build Specific values
// uses -D build configuration
#if false
    print("Not printed")
    let buildValue = 3
#else
    let buildValue = 7
#endif
print("Build value: \(buildValue)") // Build value: 7

/*
Optionals are a Swift language feature that either contains a value,
or contains nil (no value) to indicate that a value is missing.
A question mark (?) after the type marks the value as optional.
```

```
Because Swift requires every property to have a value, even nil must be
explicitly stored as an Optional value.

Optional<T> is an enum.
*/
var someOptionalString: String? = "optional" // Can be nil
// same as above, but ? is a postfix operator (syntax candy)
var someOptionalString2: Optional<String> = "optional"

if someOptionalString != nil {
    // I am not nil
    if someOptionalString!.hasPrefix("opt") {
        print("has the prefix")
    }

    let empty = someOptionalString?.isEmpty
}
someOptionalString = nil

/*
Trying to use ! to access a non-existent optional value triggers a runtime
error. Always make sure that an optional contains a non-nil value before
using ! to force-unwrap its value.
*/

// implicitly unwrapped optional
var unwrappedString: String! = "Value is expected."
// same as above, but ! is a postfix operator (more syntax candy)
var unwrappedString2: ImplicitlyUnwrappedOptional<String> = "Value is expected."

if let someOptionalStringConstant = someOptionalString {
    // has `Some` value, non-nil
    if !someOptionalStringConstant.hasPrefix("ok") {
        // does not have the prefix
    }
}

// Swift has support for storing a value of any type.
// AnyObject == id
// Unlike Objective-C `id`, AnyObject works with any value (Class, Int, struct, etc.)
var anyObjectVar: AnyObject = 7
anyObjectVar = "Changed value to a string, not good practice, but possible."

/*
    Comment here

    /*
        Nested comments are also supported
    */
*/

//
// MARK: Collections
//
```

```
/*
Array and Dictionary types are structs. So `let` and `var` also indicate
that they are mutable (var) or immutable (let) when declaring these types.
*/

// Array
var shoppingList = ["catfish", "water", "lemons"]
shoppingList[1] = "bottle of water"
let emptyArray = [String]() // let == immutable
let emptyArray2 = Array<String>() // same as above
var emptyMutableArray = [String]() // var == mutable
var explicitEmptyMutableStringArray: [String] = [] // same as above


// Dictionary
var occupations = [
    "Malcolm": "Captain",
    "kaylee": "Mechanic"
]
occupations["Jayne"] = "Public Relations"
let emptyDictionary = [String: Float]() // let == immutable
let emptyDictionary2 = Dictionary<String, Float>() // same as above
var emptyMutableDictionary = [String: Float]() // var == mutable
var explicitEmptyMutableDictionary: [String: Float] = [:] // same as above


//
// MARK: Control Flow
//

// Condition statements support "where" clauses, which can be used
// to help provide conditions on optional values.
// Both the assignment and the "where" clause must pass.
let someNumber = Optional<Int>(7)
if let num = someNumber where num > 3 {
    print("num is greater than 3")
}

// for loop (array)
let myArray = [1, 1, 2, 3, 5]
for value in myArray {
    if value == 1 {
        print("One!")
    } else {
        print("Not one!")
    }
}

// for loop (dictionary)
var dict = ["one": 1, "two": 2]
for (key, value) in dict {
    print("\(key): \(value)")
}
```

```
// for loop (range)
for i in -1...shoppingList.count {
    print(i)
}
shoppingList[1...2] = ["steak", "peacons"]
// use ..< to exclude the last number

// while loop
var i = 1
while i < 1000 {
    i *= 2
}

// repeat-while loop
repeat {
    print("hello")
} while 1 == 2

// Switch
// Very powerful, think `if` statements with syntax candy
// They support String, object instances, and primitives (Int, Double, etc)
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let localScopeValue where localScopeValue.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(localScopeValue)?"
default: // required (in order to cover all possible input)
    let vegetableComment = "Everything tastes good in soup."
}

//
// MARK: Functions
//

// Functions are a first-class type, meaning they can be nested
// in functions and can be passed around

// Function with Swift header docs (format as Swift-modified Markdown syntax)

/**
A greet operation

- A bullet in docs
- Another bullet in the docs

- Parameter name    : A name
- Parameter day : A day
- Returns : A string containing the name and day value.
*/
func greet(name: String, day: String) -> String {
```

```
    return "Hello \(name), today is \(day)."
}
greet("Bob", day: "Tuesday")

// similar to above except for the function parameter behaviors
func greet2(requiredName requiredName: String, externalParamName localParamName: String) -> String {
    return "Hello \(requiredName), the day is \(localParamName)"
}
greet2(requiredName: "John", externalParamName: "Sunday")

// Function that returns multiple items in a tuple
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
let pricesTuple = getGasPrices()
let price = pricesTuple.2 // 3.79
// Ignore Tuple (or other) values by using _ (underscore)
let (_, price1, _) = pricesTuple // price1 == 3.69
print(price1 == pricesTuple.1) // true
print("Gas price: \(price)")

// Labeled/named tuple params
func getGasPrices2() -> (lowestPrice: Double, highestPrice: Double, midPrice: Double) {
    return (1.77, 37.70, 7.37)
}
let pricesTuple2 = getGasPrices2()
let price2 = pricesTuple2.lowestPrice
let (_, price3, _) = pricesTuple2
print(pricesTuple2.highestPrice == pricesTuple2.1) // true
print("Highest gas price: \(pricesTuple2.highestPrice)")

// guard statements
func testGuard() {
   // guards provide early exits or breaks, placing the error handler code near the conditions.
    // it places variables it declares in the same scope as the guard statement.
    guard let aNumber = Optional<Int>(7) else {
        return
    }

    print("number is \(aNumber)")
}
testGuard()

// Variadic Args
func setup(numbers: Int...) {
    // its an array
    let _ = numbers[0]
    let _ = numbers.count
}

// Passing and returning functions
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
```

```
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)

// pass by ref
func swapTwoInts(inout a: Int, inout b: Int) {
    let tempA = a
    a = b
    b = tempA
}
var someIntA = 7
var someIntB = 3
swapTwoInts(&someIntA, b: &someIntB)
print(someIntB) // 7


//
// MARK: Closures
//
var numbers = [1, 2, 6]

// Functions are special case closures ({})

// Closure example.
// `->` separates the arguments and return type
// `in` separates the closure header from the closure body
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})

// When the type is known, like above, we can do this
numbers = numbers.map({ number in 3 * number })
// Or even this
//numbers = numbers.map({ $0 * 3 })

print(numbers) // [3, 6, 18]

// Trailing closure
numbers = numbers.sort { $0 > $1 }

print(numbers) // [18, 6, 3]

//
// MARK: Structures
//

// Structures and classes have very similar capabilities
struct NamesTable {
    let names: [String]
```

```
    // Custom subscript
    subscript(index: Int) -> String {
        return names[index]
    }
}

// Structures have an auto-generated (implicit) designated initializer
let namesTable = NamesTable(names: ["Me", "Them"])
let name = namesTable[1]
print("Name is \(name)") // Name is Them

//
// MARK: Error Handling
//

// The `ErrorType` protocol is used when throwing errors to catch
enum MyError: ErrorType {
    case BadValue(msg: String)
    case ReallyBadValue(msg: String)
}

// functions marked with `throws` must be called using `try`
func fakeFetch(value: Int) throws -> String {
    guard 7 == value else {
        throw MyError.ReallyBadValue(msg: "Some really bad value")
    }

    return "test"
}

func testTryStuff() {
    // assumes there will be no error thrown, otherwise a runtime exception is raised
    let _ = try! fakeFetch(7)

    // if an error is thrown, then it proceeds, but if the value is nil
    // it also wraps every return value in an optional, even if its already optional
    let _ = try? fakeFetch(7)

    do {
        // normal try operation that provides error handling via `catch` block
        try fakeFetch(1)
    } catch MyError.BadValue(let msg) {
        print("Error message: \(msg)")
    } catch {
        // must be exhaustive
    }
}
testTryStuff()

//
// MARK: Classes
//

// Classes, structures and its members have three levels of access control
```

```
// They are: internal (default), public, private

public class Shape {
    public func getArea() -> Int {
        return 0
    }
}

// All methods and properties of a class are public.
// If you just need to store data in a
// structured object, you should use a `struct`

internal class Rect: Shape {
    var sideLength: Int = 1

    // Custom getter and setter property
    private var perimeter: Int {
        get {
            return 4 * sideLength
        }
        set {
            // `newValue` is an implicit variable available to setters
            sideLength = newValue / 4
        }
    }

    // Computed properties must be declared as `var`, you know, cause' they can change
    var smallestSideLength: Int {
        return self.sideLength - 1
    }

    // Lazily load a property
    // subShape remains nil (uninitialized) until getter called
    lazy var subShape = Rect(sideLength: 4)

    // If you don't need a custom getter and setter,
    // but still want to run code before and after getting or setting
    // a property, you can use `willSet` and `didSet`
    var identifier: String = "defaultID" {
        // the `willSet` arg will be the variable name for the new value
        willSet(someIdentifier) {
            print(someIdentifier)
        }
    }

    init(sideLength: Int) {
        self.sideLength = sideLength
        // always super.init last when init custom properties
        super.init()
    }

    func shrink() {
        if sideLength > 0 {
            --sideLength
```

```
        }
    }

    override func getArea() -> Int {
        return sideLength * sideLength
    }
}

// A simple class `Square` extends `Rect`
class Square: Rect {
    convenience init() {
        self.init(sideLength: 5)
    }
}

var mySquare = Square()
print(mySquare.getArea()) // 25
mySquare.shrink()
print(mySquare.sideLength) // 4

// cast instance
let aShape = mySquare as Shape

// compare instances, not the same as == which compares objects (equal to)
if mySquare === mySquare {
    print("Yep, it's mySquare")
}

// Optional init
class Circle: Shape {
    var radius: Int
    override func getArea() -> Int {
        return 3 * radius * radius
    }

    // Place a question mark postfix after `init` is an optional init
    // which can return nil
    init?(radius: Int) {
        self.radius = radius
        super.init()

        if radius <= 0 {
            return nil
        }
    }
}

var myCircle = Circle(radius: 1)
print(myCircle?.getArea())    // Optional(3)
print(myCircle!.getArea())    // 3
var myEmptyCircle = Circle(radius: -1)
print(myEmptyCircle?.getArea())    // "nil"
if let circle = myEmptyCircle {
    // will not execute since myEmptyCircle is nil
```

```
    print("circle is not nil")
}


//
// MARK: Enums
//

// Enums can optionally be of a specific type or on their own.
// They can contain methods like classes.

enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func getIcon() -> String {
        switch self {
        case .Spades: return " "
        case .Hearts: return " "
        case .Diamonds: return " "
        case .Clubs: return " "
        }
    }
}

// Enum values allow short hand syntax, no need to type the enum type
// when the variable is explicitly declared
var suitValue: Suit = .Hearts

// String enums can have direct raw value assignments
// or their raw values will be derived from the Enum field
enum BookName: String {
    case John
    case Luke = "Luke"
}
print("Name: \(BookName.John.rawValue)")

// Enum with associated Values
enum Furniture {
    // Associate with Int
    case Desk(height: Int)
    // Associate with String and Int
    case Chair(String, Int)

    func description() -> String {
        switch self {
        case .Desk(let height):
            return "Desk with \(height) cm"
        case .Chair(let brand, let height):
            return "Chair of \(brand) with \(height) cm"
        }
    }
}

var desk: Furniture = .Desk(height: 80)
print(desk.description())     // "Desk with 80 cm"
```

```
var chair = Furniture.Chair("Foo", 40)
print(chair.description())    // "Chair of Foo with 40 cm"


//
// MARK: Protocols
//

// `protocol`s can require that conforming types have specific
// instance properties, instance methods, type methods,
// operators, and subscripts.

protocol ShapeGenerator {
    var enabled: Bool { get set }
    func buildShape() -> Shape
}

// Protocols declared with @objc allow optional functions,
// which allow you to check for conformance
@objc protocol TransformShape {
    optional func reshape()
    optional func canReshape() -> Bool
}

class MyShape: Rect {
    var delegate: TransformShape?

    func grow() {
        sideLength += 2

        // Place a question mark after an optional property, method, or
        // subscript to gracefully ignore a nil value and return nil
        // instead of throwing a runtime error ("optional chaining").
        if let reshape = self.delegate?.canReshape?() where reshape {
            // test for delegate then for method
            self.delegate?.reshape?()
        }
    }
}


//
// MARK: Other
//

// `extension`s: Add extra functionality to an already existing type

// Square now "conforms" to the `CustomStringConvertible` protocol
extension Square: CustomStringConvertible {
    var description: String {
        return "Area: \(self.getArea()) - ID: \(self.identifier)"
    }
}
```

```
print("Square: \(mySquare)")

// You can also extend built-in types
extension Int {
    var customProperty: String {
        return "This is \(self)"
    }

    func multiplyBy(num: Int) -> Int {
        return num * self
    }
}

print(7.customProperty) // "This is 7"
print(14.multiplyBy(3)) // 42

// Generics: Similar to Java and C#. Use the `where` keyword to specify the
//   requirements of the generics.

func findIndex<T: Equatable>(array: [T], _ valueToFind: T) -> Int? {
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
let foundAtIndex = findIndex([1, 2, 3, 4], 3)
print(foundAtIndex == 2) // true

// Operators:
// Custom operators can start with the characters:
//      / = - + * % < > ! & | ^ . ~
// or
// Unicode math, symbol, arrow, dingbat, and line/box drawing characters.
prefix operator !!! {}

// A prefix operator that triples the side length when used
prefix func !!! (inout shape: Square) -> Square {
    shape.sideLength *= 3
    return shape
}

// current value
print(mySquare.sideLength) // 4

// change side length using custom !!! operator, increases size by 3
!!!mySquare
print(mySquare.sideLength) // 12

// Operators can also be generics
infix operator <-> {}
func <-><T: Equatable> (inout a: T, inout b: T) {
    let c = a
```

```
    a = b
    b = c
}

var foo: Float = 10
var bar: Float = 20

foo <-> bar
print("foo is \(foo), bar is \(bar)") // "foo is 20.0, bar is 10.0"
```