```
// import a module
import Foundation

// Single-line comments are prefixed with //
// Multi-line comments start with /* and end with */
/* Nested multiline comments
 /* ARE */
 allowed
 */

// 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

//MARK: Hello, World
// From Swift 3 on, to print, just use the `print` method.
// It automatically appends a new line.
print("Hello, world")

//
// MARK: - Variables
//


//Use `let` to declare a constant and `var` to declare a variable.
let theAnswer = 42
var theQuestion = "What is the Answer?"
theQuestion = "How many roads must a man walk down?"
theQuestion = "What is six by nine?"
// Atttempting to reassign a constant throws a compile-time error
//theAnswer = 54

// Both variables and constants can be declared before they are given a value,
//   but must be given a value before they are used
let someConstant: Int
var someVariable: String
// These lines will throw errors:
//print(someConstant)
//print(someVariable)
someConstant = 0
someVariable = "0"
// These lines are now valid:
print(someConstant)
print(someVariable)

// As you can see above, variable types are automatically inferred.
//   To explicitly declare the type, write it after the variable name,
//   separated by a colon.
let aString: String = "A string"
let aDouble: Double = 0

// Values are never implicitly converted to another type.
// Explicitly make instances of the desired type.
let stringWithDouble = aString + String(aDouble)
let intFromDouble = Int(aDouble)

// For strings, use string interpolation
let descriptionString = "The value of aDouble is \(aDouble)"
// You can put any expression inside string interpolation.
let equation = "Six by nine is \(6 * 9), not 42!"
// To avoid escaping double quotes and backslashes, change the string delimiter
let explanationString = #"The string I used was "The value of aDouble is \(aDoub
le)" and the result was \#(descriptionString)"#
// You can put as many number signs as you want before the opening quote,
//   just match them at the ending quote. They also change the escape character
//   to a backslash followed by the same number of number signs.
```



```
let multiLineString = """
    This is a multi-line string.
    It's called that because it takes up multiple lines (wow!)
        Any indentation beyond the closing quotation marks is kept, the rest is
discarded.
    You can include " or "" in multi-line strings because the delimiter is three
 "s.
    """

// Arrays
let shoppingList = ["catfish", "water", "tulips",] //commas are allowed after th
e last element
let secondElement = shoppingList[1] // Arrays are 0-indexed

// Arrays declared with let are immutable; the following line throws a compile-t
ime error
//shoppingList[2] = "mango"

// Arrays are structs (more on that later), so this creates a copy instead of re
ferencing the same object
var mutableShoppingList = shoppingList
mutableShoppingList[2] = "mango"

// == is equality
shoppingList == mutableShoppingList // false

// Dictionaries declared with let are also immutable
var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic"
]
occupations["Jayne"] = "Public Relations"
// Dictionaries are also structs, so this also creates a copy
let immutableOccupations = occupations

immutableOccupations == occupations // true

// Arrays and dictionaries both automatically grow as you add elements
mutableShoppingList.append("blue paint")
occupations["Tim"] = "CEO"

// They can both be set to empty
mutableShoppingList = []
occupations = [:]

let emptyArray = [String]()
let emptyArray2 = Array<String>() // same as above
// [T] is shorthand for Array<T>
let emptyArray3: [String] = [] // Declaring the type explicitly allows you to se
t it to an empty array
let emptyArray4: Array<String> = [] // same as above

// [Key: Value] is shorthand for Dictionary<Key, Value>
let emptyDictionary = [String: Double]()
let emptyDictionary2 = Dictionary<String, Double>() // same as above
var emptyMutableDictionary: [String: Double] = [:]
var explicitEmptyMutableDictionary: Dictionary<String, Double> = [:] // same as
above

// MARK: Other variables
let Ã¸ÏM-^@Î© = "value" // unicode variable names
let ðM-^_¤¯ = "wow" // emoji variable names

// Keywords can be used as variable names
// These are contextual keywords that wouldn't be used now, so are allowed
let convenience = "keyword"
let weak = "another keyword"
let override = "another keyword"
```

```
// Using backticks allows keywords to be used as variable names even if they wou
ldn't be allowed normally
let `class` = "keyword"

// MARK: - Optionals

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

 If a type is not optional, it is guaranteed to have a value.

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

 Optional<T> is an enum, with the cases .none (nil) and .some(T) (the value)
 */

var someOptionalString: String? = "optional" // Can be nil
// T? is shorthand for Optional<T> âM-^@M-^T ? is a postfix operator (syntax can
dy)
let someOptionalString2: Optional<String> = nil
let someOptionalString3 = String?.some("optional") // same as the first one
let someOptionalString4 = String?.none //nil

/*
 To access the value of an optional that has a value, use the postfix
 operator !, which force-unwraps it. Force-unwrapping is like saying, "I
 know that this optional definitely has a value, please give it to me."

 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.
 */

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

// Swift supports "optional chaining," which means that you can call functions
//   or get properties of optional values and they are optionals of the appropri
ate type.
// You can even do this multiple times, hence the name "chaining."

let empty = someOptionalString?.isEmpty // Bool?

// if-let structure -
// if-let is a special structure in Swift that allows you to check
//   if an Optional rhs holds a value, and if it does unwrap
//   and assign it to the lhs.
if let someNonOptionalStringConstant = someOptionalString {
    // has `Some` value, non-nil
    // someOptionalStringConstant is of type String, not type String?
    if !someNonOptionalStringConstant.hasPrefix("ok") {
        // does not have the prefix
    }
}

//if-var is allowed too!
if var someNonOptionalString = someOptionalString {
    someNonOptionalString = "Non optional AND mutable"
    print(someNonOptionalString)
}
```



```
// You can bind multiple optional values in one if-let statement.
//   If any of the bound values are nil, the if statement does not execute.
if let first = someOptionalString, let second = someOptionalString2,
    let third = someOptionalString3, let fourth = someOptionalString4 {
    print("\(first), \(second), \(third), and \(fourth) are all not nil")
}

//if-let supports "," (comma) clauses, which can be used to
//   enforce conditions on newly-bound optional values.
// Both the assignment and the "," clause must pass.
let someNumber: Int? = 7
if let num = someNumber, num > 3 {
    print("num is not nil and is greater than 3")
}

// Implicitly unwrapped optional âM-^@M-^T An optional value that doesn't need t
o be unwrapped
let unwrappedString: String! = "Value is expected."

// Here's the difference:
let forcedString = someOptionalString! // requires an exclamation mark
let implicitString = unwrappedString // doesn't require an exclamation mark

/*
 You can think of an implicitly unwrapped optional as giving permission
 for the optional to be unwrapped automatically whenever it's used.
 Rather than placing an exclamation mark after the optional's name each time you
 use it,
 you place an exclamation mark after the optional's type when you declare it.
 */

// Otherwise, you can treat an implicitly unwrapped optional the same way the yo
u treat a normal optional
//   (i.e., if-let, != nil, etc.)

// Pre-Swift 5, T! was shorthand for ImplicitlyUnwrappedOptional<T>
// Swift 5 and later, using ImplicitlyUnwrappedOptional throws a compile-time er
ror.
//var unwrappedString2: ImplicitlyUnwrappedOptional<String> = "Value is expected
." //error

// The nil-coalescing operator ?? unwraps an optional if it contains a non-nil v
alue, or returns a default value.
someOptionalString = nil
let someString = someOptionalString ?? "abc"
print(someString) // abc
// a ?? b is shorthand for a != nil ? a! : b

// MARK: - Control Flow

let condition = true
if condition { print("condition is true") } // can't omit the braces

if theAnswer > 50 {
    print("theAnswer > 50")
} else if condition {
    print("condition is true")
} else {
    print("Neither are true")
}

// The condition in an `if` statement must be a `Bool`, so the following code is
 an error, not an implicit comparison to zero
//if 5 {
//    print("5 is not zero")
//}

// Switch
// Must be exhaustive
```

```
// Does not implicitly fall through, use the fallthrough keyword
// Very powerful, think `if` statements with syntax candy
// They support String, object instances, and primitives (Int, Double, etc)
let vegetable = "red pepper"
let vegetableComment: String
switch vegetable {
case "celery":
    vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress": // match multiple values
    vegetableComment = "That would make a good tea sandwich."
case let localScopeValue where localScopeValue.hasSuffix("pepper"):
    vegetableComment = "Is it a spicy \(localScopeValue)?"
default: // required (in order to cover all possible input)
    vegetableComment = "Everything tastes good in soup."
}
print(vegetableComment)

// You use the `for-in` loop to iterate over a sequence, such as an array, dicti
onary, range, etc.
for element in shoppingList {
    print(element) // shoppingList is of type `[String]`, so element is of type
`String`
}
//Iterating through a dictionary does not guarantee any specific order
for (person, job) in immutableOccupations {
    print("\(person)'s job is \(job)")
}
for i in 1...5 {
    print(i, terminator: " ") // Prints "1 2 3 4 5"
}
for i in 0..<5 {
    print(i, terminator: " ") // Prints "0 1 2 3 4"
}
//for index in range can replace a C-style for loop:
//    for (int i = 0; i < 10; i++) {
//        //code
//    }
//becomes:
//    for i in 0..<10 {
//        //code
//    }
//To step by more than one, use the stride(from:to:by:) or stride(from:through:b
y) functions
//`for i in stride(from: 0, to: 10, by: 2)` is the same as `for (int i = 0; i <
10; i += 2)`
//`for i in stride(from: 0, through: 10, by: 2)` is the same as `for (int i = 0;
 i <= 10; i += 2)

// while loops are just like most languages
var i = 0
while i < 5 {
    i += Bool.random() ? 1 : 0
    print(i)
}

// This is like a do-while loop in other languages âM-^@M-^T the body of the loo
p executes a minimum of once
repeat {
    i -= 1
    i += Int.random(in: 0...3)
} while i < 5

// The continue statement continues executing a loop at the next iteration
// The break statement ends a swift or loop immediately

// 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.
///
/// - Parameters:
///   - name: A name.
///   - 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(name: "Bob", day: "Tuesday")

// Ideally, function names and parameter labels combine to make function calls s
imilar to sentences.
func sayHello(to name: String, onDay day: String) -> String {
    return "Hello \(name), the day is \(day)"
}
sayHello(to: "John", onDay: "Sunday")

//Functions that don't return anything can omit the return arrow; they don't nee
d to say that they return Void (although they can).
func helloWorld() {
    print("Hello, World!")
}

// Argument labels can be blank
func say(_ message: String) {
    print(#"I say "\#(message)""#)
}
say("Hello")

// Default parameters can be omitted when calling the function.
func printParameters(requiredParameter r: Int, optionalParameter o: Int = 10) {
    print("The required parameter was \(r) and the optional parameter was \(o)")
}
printParameters(requiredParameter: 3)
printParameters(requiredParameter: 3, optionalParameter: 6)

// Variadic args âM-^@M-^T only one set per function.
func setup(numbers: Int...) {
    // it's an array
    let _ = numbers[0]
    let _ = numbers.count
}

// pass by ref
func swapTwoInts(a: inout Int, b: inout Int) {
    let tempA = a
    a = b
    b = tempA
}
var someIntA = 7
var someIntB = 3
swapTwoInts(a: &someIntA, b: &someIntB) //must be called with an & before the va
riable name.
print(someIntB) // 7

type(of: greet) // (String, String) -> String
type(of: helloWorld) // () -> Void

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

```
var increment = makeIncrementer()
increment(7)

func performFunction(_ function: (String, String) -> String, on string1: String,
 and string2: String) {
    let result = function(string1, string2)
    print("The result of calling the function on \(string1) and \(string2) was \
(result)")
}

// 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: Do
uble) {
    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.
    // They make it easier to avoid the "pyramid of doom"
    guard let aNumber = Optional<Int>(7) else {
        return // guard statements MUST exit the scope that they are in.
        // They generally use `return` or `throw`.
    }

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

// Note that the print function is declared like so:
//     func print(_ input: Any..., separator: String = " ", terminator: String =
 "\n")
// To print without a newline:
print("No newline", terminator: "")
print("!")

// 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.sorted { $0 > $1 }

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

// 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
    var icon: Character {
        switch self {
        case .spades:
            return "âM-^Y¤"
        case .hearts:
            return "âM-^Y¡"
        case .diamonds:
            return "âM-^Y¢"
        case .clubs:
            return "âM-^Y£"
        }
    }
}

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

// Conforming to the CaseIterable protocol automatically synthesizes the allCase
s property,
//   which contains all the values. It works on enums without associated values
or @available attributes.
enum Rank: CaseIterable {
    case ace
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king
    var icon: String {
        switch self {
        case .ace:
            return "A"
        case .two:
            return "2"
        case .three:
            return "3"
        case .four:
            return "4"
        case .five:
            return "5"
        case .six:
            return "6"
        case .seven:
            return "7"
        case .eight:
            return "8"
        case .nine:
            return "9"
        case .ten:
            return "10"
        case .jack:
            return "J"
```

```
        case .queen:
            return "Q"
        case .king:
            return "K"
        }
    }
}

for suit in [Suit.clubs, .diamonds, .hearts, .spades] {
    for rank in Rank.allCases {
        print("\(rank.icon)\(suit.icon)")
    }
}

// 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 {
        //either placement of let is acceptable
        switch self {
        case .desk(let height):
            return "Desk with \(height) cm"
        case let .chair(brand, 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: - Structures & Classes

/*
 Structures and classes in Swift have many things in common. Both 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 don't have:
 - 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.

 Unless you need to use a class for one of these reasons, use a struct.

 Structures are value types, while classes are reference types.
 */
```



```
// MARK: Structures

struct NamesTable {
    let names: [String]

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

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

// MARK: Classes

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

class Rect: Shape {
    var sideLength: Int = 1

    // Custom getter and setter property
    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 -= 1
        }
```

```
    }

    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

// downcast instance:
// Because downcasting can fail, the result can be an optional (as?) or an impli
citly unwrpped optional (as!).
let anOptionalSquare = aShape as? Square // This will return nil if aShape is no
t a Square
let aSquare = aShape as! Square // This will throw a runtime error if aShape is
not a Square

// 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: - Protocols

// protocols are also known as interfaces in some other languages

// `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
}

// MARK: - Other

// MARK: Typealiases

// Typealiases allow one type (or composition of types) to be referred to by ano
ther name
typealias Integer = Int
let myInteger: Integer = 0

// MARK: = Operator

// Assignment does not return a value. This means it can't be used in conditiona
l statements,
//   and the following statement is also illegal
//    let multipleAssignment = theQuestion = "No questions asked"
//But you can do this:
let multipleAssignment = "No questions asked", secondConstant = "No answers give
n"

// MARK: Ranges

// The ..< and ... operators create ranges.

// ... is inclusive on both ends (a "closed range") âM-^@M-^T mathematically, [0
, 10]
let _0to10 = 0...10
// ..< is inclusive on the left, exclusive on the right (a "range") âM-^@M-^T ma
thematically, [0, 10)
let singleDigitNumbers = 0..<10
// You can omit one end (a "PartialRangeFrom") âM-^@M-^T mathematically, [0, â
M-^H)
let toInfinityAndBeyond = 0...
// Or the other end (a "PartialRangeTo") âM-^@M-^T mathematically, (-âM-^H, 0)
let negativeInfinityToZero = ..<0
// (a "PartialRangeThrough") âM-^@M-^T mathematically, (-âM-^H, 0]
let negativeInfinityThroughZero = ...0

// MARK: Wildcard operator

// In Swift, _ (underscore) is the wildcard operator, which allows values to be
ignored

// It allows functions to be declared without argument labels:
func function(_ labelLessParameter: Int, label labeledParameter: Int, labelAndPa
rameterName: Int) {
    print(labelLessParameter, labeledParameter, labelAndParameterName)
}
function(0, label: 0, labelAndParameterName: 0)

// You can ignore the return values of functions
func printAndReturn(_ str: String) -> String {
    print(str)
    return str
}
let _ = printAndReturn("Some String")

// You can ignore part of a tuple and keep part of it
func returnsTuple() -> (Int, Int) {
    return (1, 2)
}
let (_, two) = returnsTuple()
```

```
// You can ignore closure parameters
let closure: (Int, Int) -> String = { someInt, _ in
    return "\(someInt)"
}
closure(1, 2) // returns 1

// You can ignore the value in a for loop
for _ in 0..<10 {
    // Code to execute 10 times
}

// MARK: Access Control

/*
 Swift has five levels of access control:
 - Open: Accessible *and subclassible* in any module that imports it.
 - Public: Accessible in any module that imports it, subclassible in the module
it is declared in.
 - Internal: Accessible and subclassible in the module it is declared in.
 - Fileprivate: Accessible and subclassible in the file it is declared in.
 - Private: Accessible and subclassible in the enclosing declaration (think inne
r classes/structs/enums)

 See more here: https://docs.swift.org/swift-book/LanguageGuide/AccessControl.ht
ml
 */

// MARK: Preventing Overrides

// You can add keyword `final` before a class or instance method, or a property
to prevent it from being overridden
class Shape {
    final var finalInteger = 10
}

// Prevent a class from being subclassed
final class ViewManager {
}

// MARK: Conditional Compilation, Compile-Time Diagnostics, & Availability Condi
tions

// Conditional Compilation
#if false
print("This code will not be compiled")
#else
print("This code will be compiled")
#endif
/*
 Options are:
 os()                   macOS, iOS, watchOS, tvOS, Linux
 arch()                 i386, x86_64, arm, arm64
 swift()                >= or < followed by a version number
 compiler()             >= or < followed by a version number
 canImport()            A module name
 targetEnvironment()    simulator
 */
#if swift(<3)
println()
#endif

// Compile-Time Diagnostics
// You can use #warning(message) and #error(message) to have the compiler emit w
arnings and/or errors
#warning("This will be a compile-time warning")
//  #error("This would be a compile-time error")

//Availability Conditions
```



```
if #available(iOSMac 10.15, *) {
    // macOS 10.15 is available, you can use it here
} else {
    // macOS 10.15 is not available, use alternate APIs
}

// MARK: Any and AnyObject

// Swift has support for storing a value of any type.
// For that purpose there are two keywords: `Any` and `AnyObject`
// `AnyObject` == `id` from Objective-C
// `Any` works with any values (class, Int, struct, etc.)
var anyVar: Any = 7
anyVar = "Changed value to a string, not good practice, but possible."
let anyObjectVar: AnyObject = Int(1) as NSNumber

// MARK: Extensions

// Extensions allow you to add extra functionality to an already-declared type,
even one that you don't have the source code for.

// 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 doubled: Int {
        return self * 2
    }

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

    mutating func multiplyBy(num: Int) {
        self *= num
    }
}

print(7.doubled) // 14
print(7.doubled.multipliedBy(num: 3)) // 42

// MARK: Generics

// 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.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
findIndex(array: [1, 2, 3, 4], valueToFind: 3) // Optional(2)

// You can extend types with generics as well
extension Array where Array.Element == Int {
    var sum: Int {
        var total = 0
        for el in self {
            total += el
```

```
        }
        return total
    }
}

// MARK: 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 !!! (shape: inout 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> (a: inout T, b: inout 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"

// MARK: - Error Handling

// The `Error` protocol is used when throwing errors to catch
enum MyError: Error {
    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 r
aised
    let _ = try! fakeFetch(value: 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 opti
onal
    let _ = try? fakeFetch(value: 7)

    do {
        // normal try operation that provides error handling via `catch` block
```



```
        try fakeFetch(value: 1)
    } catch MyError.badValue(let msg) {
        print("Error message: \(msg)")
    } catch {
        // must be exhaustive
    }
}
testTryStuff()
```