Swift Take-Home Points

Welcome Developers!

This document is for anyone who aspires to learn Swift in a couple of days or for iOS developers as an all-time quick reference document.

The document gives a brief overview of Swift 2.2. You can download the iBook from here.

You must be wondering what would be the difference between the Apple's Swift 2.2 book and this document. In this document I have jotted down take-home points from the book and organised the sections into points for easy reading and quick reference.

The structure in this document reflects the same structure followed by Apple in the iBook. You are free to refer to the document you are comfortable with.

The Swift version that is documented is 2.2. I will update the document as and when Apple rolls out new versions of Swift.

If you would like to contribute, feel free to get in touch with me @ E-mail

Read On!!!! Code On!!!

Table of Contents

1. printing a line
   println("Hello, world!")

2. Integers
   UInt8, Int32

   Integer bounds
   ```
   let minValue = UInt8.min      // equals to 0
   let maxValue = UInt8.max      // equals to 255
   ```

3. Numeric Literals
   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

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

4. **Type Aliases** - **typealias**
   Type aliases define an alternative name for an existing type. You define type aliases with the typealias keyword.
   Ex: typealias AudioSample = UInt16
   var maxAmplitudeFound = AudioSample.min

5. **Tuples**
   Tuples group multiple values into a single compound value. The values within a tuple can be of any type and do not have to be of the same type as each other.
   Ex: let http404Error = (404, "Not Found")

   You can decompose a tuple's contents into separate constants or variables, which you then access as usual:
   Ex: let (statusCode, statusMessage) = http404Error

   If you only need some of the tuple's values, ignore parts of the tuple with an underscore (_) when you decompose the tuple:
   Ex: let (justTheStatusCode, _) = http404Error

   Alternatively, access the individual element values in a tuple using index numbers starting at zero:
   print("The status code is \(http404Error.0)")

   You can name the individual elements in a tuple when the tuple is defined:
   let http200Status = (statusCode: 200, description: "OK")

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

7. **Optional Binding - if let | if var | where**
   ```
   if let constantName = someOptional {
      ....
   }

   if var variableName = someOptional {
      ....
   }

   if let firstNumber = Int("4"), secondNumber = Int("42") where firstNumber < secondNumber {
      print("\(firstNumber) < \(secondNumber)")          // prints "4 < 42"
   }
   ```

8. **Optional Values** - **?**

If the value can become nil in the future, so that you test for this

9. **Implicitly unwrapped optionals - !**
   If it really shouldn't become nil in the future, but it needs to be nil initially

```
var optionalString: String? = "Hello"
optionalString = nil

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalString {
        greeting = "Hello, \(name)"
}
```

10. **Assertions - assert**
    In some cases, it is simply not possible for your code to continue execution if a particular condition is not satisfied. In these situations, you can trigger an assertion in your code to end code execution and to provide an opportunity to debug the cause of the absent or invalid value.

    An assertion is a runtime check that a Boolean condition definitely evaluates to true. If the condition evaluates to true, code execution continues as usual; if the condition evaluates to false, code execution ends, and your app is terminated.

    An assertion also lets you provide a suitable debug message as to the nature of the assert.

```
let age = -3
assert(age >= 0, "A person's age cannot be less than zero")
```

    Assertions are disabled when your code is compiled with optimizations, such as when building with an app target's default Release configuration in Xcode.

11. **Nil coalescing operator** - **??**
    let a = b **??** c

12. **Range operators** - **… | ..< | ..>**
    closed range: a…b
    range that runs from a to b, and includes the values a and b. The value of a must not be greater than b.
    Ex for index in 1**...**5 {
    print("\(index) times 5 is \(index * 5)")
    }

    half-open range: a..<b
    range that runs from a to b, but does not include b.

13. **String Indices - startIndex | endIndex | predecessor() | successor() | advancedBy() | indices**
    let greeting = "Guten Tag!

```
greeting[greeting.startIndex]            // G

greeting[greeting.endIndex.predecessor()] // !

greeting[greeting.startIndex.successor()]    // u

let index = greeting.startIndex.advancedBy(7)
greeting[index]          // a

greeting.endIndex.successor()                  // runtime-error
```

    Use the **indices** property of the characters property to create a Range of all of the indexes used to access individual characters in a string.

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

14. **Prefix and Suffix - hasPrefix(_:) | hasSuffix(_:)**
hasPrefix(_:) and hasSuffix(_:) : To check whether a string has a particular string prefix or suffix

15. declaring variables and constants

```
var myVariable = 42
myVariable = 50
let myConstant = 42
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
let label = "The width for this example is "
let width = 94
let widthLabel = label + String(width)
let apples = 3
let oranges = 5
let appleSummary = "I have " + String(apples) + " apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

16. **let**

an object as let cannot be assigned another object of the same type but can have the variables in that object re-assigned

```
let number = 1
number = 2                    // ERROR

class classA {
        var testNumber
}
let object = classA()
object. testNumber = 3
object. testNumber = 4        //POSSIBLE
```

17. **arrays and dictionary**

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water”
var occupations = [ "Malcolm": “Captain", "Kaylee": “Mechanic" ]
occupations["Jayne"] = "Public Relations”
empty array and dictionary
let emptyArray = [String]()                OR        let shoppingList = []
let emptyDictionary = [String: Float]()    OR        let occupations = [:]
var retval = [Int: Double]()

Removing a key value pair from dictionary
airports["APL"] = nil
```

18. **Sets**

A set stores distinct values of the same type in a collection with no defined ordering.
A type must be hashable in order to be stored in a set.

19. **Hash value in set**

A hash value is an Int value that is the same for all objects that compare equally, such that if a == b, it follows that a.hashValue == b.hashValue.
All of Swift’s basic types (such as String, Int, Double, and Bool) are hashable by default, and can be used as set value types or dictionary key types.

Enumeration member values without associated values are also hashable by default.

You can use your own custom types as set value types or dictionary key types by making them conform to the Hashable protocol from Swift’s standard library.
Types that conform to the Hashable protocol must provide a gettable Int property called hashValue. The value returned by a type’s hashValue property is not required to be the same across different executions of the same program, or in different programs.

20. Control Flow - **for … in …**

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
        if score > 50 {
                teamScore += 3
        } else {
                teamScore += 1
        }
}
println(teamScore)

var airports = ["YYZ": "Toronto Pearson", "DUB": “Dublin"]
for (airportCode, airportName) in airports {
        println(\(airportCode) “:” \(airportName) )
}
```

21. Control Flow - **While - while | repeat…while**

Swift provides two kinds of while loops:

- while evaluates its condition at the start of each pass through the loop.
- repeat-while evaluates its condition at the end of each pass through the loop.

```
while condition {
   statements
}

repeat {
   statements
} while condition
```

22. Control Flow - **if…else**

```
if condition {
  statement
} else condition {
  statement
}
```

23. Control Flow - Switch - **Optional Value**

```
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 x where x.hasSuffix("pepper"):
                let vegetableComment = "Is it a spicy \(x)?"
        default:
                let vegetableComment = "Everything tastes good in soup."
}
```

The “default:” is a must. else the compiler will throw an error

24. Control Flow - Switch - **interval counting**

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
var naturalCount: String
switch approximateCount {
case 0:
   naturalCount = "no"
case 1..<5:
   naturalCount = "a few"
case 5…12:
   naturalCount = "several"
case 12..<100:
   naturalCount = "dozens of"
```

```
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
```

25. Control Flow - Switch - **tuples**

```
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”)
}
```

26. Control Flow - Switch - **value binding** and **where**

```
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) where x == y:
    print("somewhere else at (\(x), \(y))")
}
```

27. Control Flow - **Continue**

The continue statement tells a loop to stop what it is doing and start again at the beginning of the next iteration through the loop.

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput.characters {
    switch character {
    case "a", "e", "i", "o", "u", " ":
        continue
    default:
        puzzleOutput.append(character)
    }
}
```

28. Control Flow - **Break**

The break statement ends execution of an entire control flow statement immediately.

29. Control Flow - **Fallthrough**

If you really need C-style (C requires you to insert an explicit break statement at the end of every switch case to prevent fallthrough.) fallthrough behavior, you can opt in to this behavior on a case-by-case basis with the fallthrough keyword.

30. Control Flow - **Labeled statements**

With a conditional statement, you can use a statement label with the break statement to end the execution of the labeled statement.

```
gameLoop: while square != finalSquare {
   switch square + diceRoll {
    case finalSquare:
      break gameLoop
    }
}
```

31. Control Flow - **Guard**

A guard statement, like an if statement, executes statements depending on the Boolean value of an expression. You use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed.

```
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }
……..
}
```

Any variables or constants that were assigned values using an optional binding as part of the condition are available for the rest of the code block that the guard statement appears in.

32. Control Flow - **API Availability**

Swift has built-in support for checking API availability, which ensures that you don't accidentally use APIs that are unavailable on a given deployment target.

```
if #available(iOS 9, OSX 10.10, *) {
    // Use iOS 9 APIs on iOS, and use OS X v10.10 APIs on OS X
} else {
    // Fall back to earlier iOS and OS X APIs
}
```

33. Control Flow - loops example

```
let interestingNumbers = [
        "Prime": [2, 3, 5, 7, 11, 13],
        "Fibonacci": [1, 1, 2, 3, 5, 8],
        "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
        for number in numbers {
                if number > largest {
                        largest = number
                }
        }
}
println(largest)
```

34. declaring/initializing array variables from a value (using its key) from a dictionary

```
let interestingNumbers = [
        "Prime": [2, 3, 5, 7, 11, 13],
        "Fibonacci": [1, 1, 2, 3, 5, 8],
        "Square": [1, 4, 9, 16, 25],
]
var arrayNumbers:Array = interestingNumbers[“Square"]!
———
func returnPossibleTips() -> [Int: (tipAmt:Double, total:Double)] {
        let possibleTipsInferred = [0.15, 0.18, 0.20]
        let possibleTipsExplicit:[Double] = [0.15, 0.18, 0.20]

        var retval = Dictionary<Int, (tipAmt:Double, total:Double)>()

        for possibleTip in possibleTipsInferred {
                let intPct = Int(possibleTip*100)
                retval[intPct] = calcTipWithTipPct(possibleTip)
        }
        return retval

}
```

---

35. **Functions**

```
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
greet("Bob", “Tuesday")
//greet should have the exact number of list of arguments. neither less or more
```

36. Functions- **Tuple**

returning a compound of values called **tuple**

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
     var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}

let statistics = calculateStatistics([5, 3, 100, 3, 9])
println(statistics.sum)        OR      println(statistics.2)
```

———————

```
let http404Error = (404, "Not Found”)
let (statusCode, statusMessage) = http404Error
println("The status code is \(statusCode)”)

if you wanna avoid the statusMessage
let (justTheStatusCode, _) = http404Error
```

37. Functions - **Variadic Parameters**

A variadic parameter accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called.

```
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}

sumOf()
sumOf(42, 597, 12)
```

38. Functions - **first-class type**

This means that a function can return another function as its value.

```
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

39. Functions - **As Arguments**

Functions can take another function as one of its arguments

```
func hasAnyMatches(list: [Int], condition: Int -> Bool) -> Bool {
```

```
        for item in list {
            if condition(item) {
                return true
            }
        }
        return false
    }
    func lessThanTen(number: Int) -> Bool {
            return number < 10
    }
    var numbers = [20, 19, 7, 12]
    hasAnyMatches(numbers, lessThanTen)
```

// “lessThanTen” is sent as an argument to the function “hasAnyMatches”. “hasAnyMatches” function internally calls the “lessThanTen” function with the integer in the array.

40. Function - **External parameter name and Local parameter name**

```
func sayHello(to person: String, and anotherPerson: String) -> String {
   return "Hello \(person) and \(anotherPerson)!"
}
print(sayHello(to: "Bill", and: “Ted”))
```

from above example “to” & “and” are external parameter names
“person” & “anotherPerson” are internal parameter names

omitting external parameter name

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

41. Functions - **Default parameter values**

You can define a default value for any parameter in a function by assigning a value to the parameter after that parameter’s type. If a default value is defined, you can omit that parameter when calling the function.

```
func someFunction(parameterWithDefault: Int = 12) {
   // function body goes here
}
someFunction(6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12
```

42. Functions - **constants and variable** parameters

Function parameters are constants by default.

```
func alignRight(var string: String, totalLength: Int, pad: Character) -> String {
        ……..
}
```

The changes you make to a variable parameter do not persist beyond the end of each call to the function, and are not visible outside the function’s body. The variable parameter only exists for the lifetime of that function call.

43. Functions - **in-out parameters**

If you want a function to modify a parameter’s value, and you want those changes to persist after the function call has ended, define that parameter as an in-out parameter instead.

```
func swapTwoInts(inout a: Int, inout _ b: Int) {
   let temporaryA = a
   a = b
   b = temporaryA
}
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
```

You can only pass a variable as the argument for an in-out parameter.

In-out parameters cannot have default values, and variadic parameters cannot be marked as inout.

44. Functions - **Function Types**

```
func addTwoInts(a: Int, b: Int) -> Int {
    return a + b
}
let anotherMathFunction = addTwoInts
anotherMathFunction(2, 3)
```

45. Functions - **Function types as parameters**

```
func addTwoInts(a: Int, b: Int) -> Int {
    return a + b
}
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {
    println("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)              //Result: 8
```

46. Functions - **Function types as return types**

```
func stepForward(input: Int) -> Int {
    return input + 1
}
func stepBackward(input: Int) -> Int {
    return input - 1
}
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
```

The reference to “stepBackward” returned function is stored in a constant called moveNearerToZero

47. Functions - **Nested functions**

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward
}
```

48. **Closures - { … }**

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is executed. You can write a closure without a name by surrounding code with braces (**{}**). Use “in” to separate the arguments and return type from the body.

```
numbers.map({
        (number: Int) -> Int in
        let result = 2 * number
        return result
})
println(numbers)                //[40, 38, 14, 24]
```

//this closure will iterate through all the elements in the array “numbers” (if its in the scope) and will double each element in the array and will return the updated array.

Default values cannot be provided.

You have several options for writing closures more concisely. When a closure’s type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both. Single statement closures implicitly return the value of their only statement.

```
let mappedNumbers = numbers.map({ number in 3 * number })
println(mappedNumbers)              //[60, 57, 21, 36]
```

You can refer to parameters by number instead of by name—this approach is especially useful in very short closures. A closure passed as the last argument to a function can appear immediately after the parentheses.

```
let sortedNumbers = sorted(numbers) { $0 > $1 }
sortedNumbers                   //will sort the array in descending order
```

49. **Closure as a Parameter**

```
reversed = sorted(names, { (s1: String, s2: String) -> Bool in
        return s1 > s2
})
```

The parameter types and the return type can be inferred by the compiler

```
reversed = names.sort( { s1, s2 in return s1 > s2 } )
```

Shorthand version

```
reversed = sorted(names, { $0 > $1 } )
```

Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure’s arguments by the names $0, $1, $2, and so on

Shorthand version 2

```
reversed = names.sort(>)
```

50. **Trailing Closures**

A trailing closure is a closure expression that is written outside of (and after) the parentheses of the function call it supports:

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
   // function body goes here
}
```

the call to the func can be written as

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

51. **Capturing values using closures**

A nested function can capture any of its outer function’s arguments and can also capture any constants and variables defined within the outer function.

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
```

```
        var runningTotal = 0
        func incrementor() -> Int {
                runningTotal += amount
                return runningTotal
        }
        return incrementor
}
```

incrementor is returned by makeIncrementor as a closure that increments runningTotal by amount each time it is called. The return type of makeIncrementor is () -> Int. This means that it returns a function, rather than a simple value.

Because incrementor modifies the runningTotal variable each time it is called, incrementor captures a reference to the current runningTotal variable, and not just a copy of its initial value. Capturing a reference ensures that runningTotal does not disappear when the call to makeIncrementor ends, and ensures that runningTotal will continue to be available the next time that the incrementor function is called.

```
let incrementByTen = makeIncrementor(forIncrement: 10)
incrementByTen()            // returns a value of 10
incrementByTen()            // returns a value of 20

let incrementBySeven = makeIncrementor(forIncrement: 7)
incrementBySeven()          // returns a value of 7
```

In the example above, incrementBySeven and incrementByTen are constants, but the closures these constants refer to are still able to increment the runningTotal variables that they have captured. This is because **functions and closures are reference types**.

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()        // returns a value of 50
```

If you assign a closure to a property of a class instance, and the closure captures that instance by referring to the instance or its members, you will create a strong reference cycle between the closure and the instance. Swift uses capture lists to break these strong reference cycles.

52. Closure - **@noescape**

A closure is said to escape a function when the closure is passed as an argument to the function, but is called after the function returns. **When you declare a function that takes a closure as one of its parameters, you can write @noescape before the parameter name to indicate that the closure is not allowed to escape.**

Marking a closure with @noescape lets the compiler make more aggressive optimizations because it knows more information about the closure's lifespan.

```
func someFunctionWithNoescapeClosure(@noescape closure: () -> Void) {
   closure()
}
```

One way that a closure can escape is by being stored in a variable that is defined outside the function.

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: () -> Void) {
   completionHandlers.append(completionHandler)
}

class SomeClass {
   var x = 10
   func doSomething() {
      someFunctionWithEscapingClosure { self.x = 100 }
      someFunctionWithNonescapingClosure { x = 200 }
   }
}
```

Marking a closure with **@noescape** lets you refer to self implicitly within the closure (meaning "self." is not required).

53. Closure - **Autoclosure - @autoclosure**

An autoclosure is a closure that is automatically created to wrap an expression that's being passed as an argument to a function. It doesn't take any arguments, and when it's called, it returns the value of the expression that's wrapped inside of it.

Example: let customerProvider = { customersInLine.removeAtIndex(0) }

An autoclosure lets you delay evaluation, because the code inside isn't run until you call the closure.

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)                    // Prints "5

let customerProvider = { customersInLine.removeAtIndex(0) }
print(customersInLine.count)                    // Prints "5

print("Now serving \(customerProvider())!")   // Prints "Now serving Chris!"
print(customersInLine.count)                    // Prints "4
```

——

Below the type of customerProvider is not String but () -> String—a function with no parameters that returns a string.

You get the same behavior of delayed evaluation when you pass a closure as an argument to a function.

```
func serveCustomer(customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serveCustomer( { customersInLine.removeAtIndex(0) } )          // Prints "Now serving Alex!"
```

——

The version of serveCustomer(_:) below performs the same operation but, instead of taking an explicit closure, it takes an autoclosure by marking its parameter with the @autoclosure attribute. Now you can call the function as if it took a String argument instead of a closure. The argument is automatically converted to a closure, because the customerProvider parameter is marked with the @autoclosure attribute.

```
func serveCustomer(@autoclosure customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serveCustomer(customersInLine.removeAtIndex(0))
```

54. Closure - **@autoclosure(escaping)**

The @autoclosure attribute implies the @noescape attribute. If you want an autoclosure that is allowed to escape, use the **@autoclosure(escaping)** form of the attribute.

In the code below, instead of calling the closure passed to it as its customerProvider argument, the collectCustomerProviders(_:) function appends the closure to the customerProviders array. The array is declared outside the scope of the function, which means the closures in the array can be executed after the function returns. As a result, the value of the customerProvider argument must be allowed to escape the function's scope.

```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(@autoclosure(escaping) customerProvider: () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.removeAtIndex(0))
collectCustomerProviders(customersInLine.removeAtIndex(0))

print("Collected \(customerProviders.count) closures.")          // Prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
// Prints "Now serving Barry!"
// Prints "Now serving Daniella!
```

55. **Enumerations**

An enumeration defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

56. Enums - **associated values**

This enables you to store additional custom information along with the case value and permits this information to vary each time you use that case in your code.
You can define Swift enumerations to store associated values of any given type, and the value types can be different for each case of the enumeration if needed.

```
enum Barcode {
    case UPCA(Int, Int, Int, Int)
    case QRCode(String)
}
```

assigning the associated value to an enum object

```
var productBarcode = Barcode.UPCA(8, 85909, 51226, 3)
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

extracting the values from the enum

```
switch productBarcode {
case .UPCA(let numberSystem, let manufacturer, let product, let check):
    print("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).")
case .QRCode(let productCode):
    print("QR code: \(productCode).")
}
```

In the above snippet, if all the associated values are extracted as constants or variables, then the code can be written as

```
switch productBarcode {
case let .UPCA(numberSystem, manufacturer, product, check):
    print("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).")
case let .QRCode(productCode):
    print("QR code: \(productCode).")
}
```

57. Enums - **Raw values**

enumeration cases can come prepopulated with default values (called raw values), which are all of the same type.

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

58. Enums - **implicitly assigned raw values**

When you're working with enumerations that store integer or string raw values, you don't have to explicitly assign a raw value for each case. When you don't, Swift will automatically assign the values for you.

For instance, when integers are used for raw values, the implicit value for each case is one more than the previous case. If the first case doesn't have a value set, it's value is 0.

```
enum Planet: Int {
    case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

In the example above, Planet.Mercury has an explicit raw value of 1, Planet.Venus has an implicit raw value of 2, and so on.

When strings are used for raw values, the implicit value for each case is the text of that case's name.

```
enum CompassPoint: String {
    case North, South, East, West
}
```

In the example above, CompassPoint.South has an implicit raw value of "South", and so on.

59. Enums - initialising from raw value

If you define an enumeration with a raw-value type, the enumeration automatically receives an initializer that takes a value of the raw value's type (as a parameter called rawValue) and returns either an enumeration case or nil. You can use this initializer to try to create a new instance of the enumeration.
let possiblePlanet = Planet(rawValue: 7)
Not all possible Int values will find a matching planet, however. Because of this, the raw value initializer always returns an optional enumeration case.

The **raw value initializer is a failable initializer**, because not every raw value will return an enumeration case.

60. Enums - **Recursive Enums - indirect**

A recursive enumeration is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases. You indicate that an enumeration case is recursive by writing indirect before it.

```
enum ArithmeticExpression {
    case Number(Int)
    indirect case Addition(ArithmeticExpression, ArithmeticExpression)
    indirect case Multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

You can also write indirect before the beginning of the enumeration, to enable indirection for all of the enumeration's cases that need it:

```
indirect enum ArithmeticExpression {
    case Number(Int)
    case Addition(ArithmeticExpression, ArithmeticExpression)
    case Multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

Example

```
func evaluate(expression: ArithmeticExpression) -> Int {
    switch expression {
    case .Number(let value):
        return value
    case let .Addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .Multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

let five = ArithmeticExpression.Number(5)
let four = ArithmeticExpression.Number(4)
let sum = ArithmeticExpression.Addition(five, four)
let product = ArithmeticExpression.Multiplication(sum, ArithmeticExpression.Number(2))
print(evaluate(product))
```

---

61. **Class and Structs**

Swift does not require you to create separate interface and implementation files for custom classes and structures. In Swift, you define a class or a structure in a single file, and the external interface to that class or structure is automatically made available for other code to use.

Classes and structures 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 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.

Structs have an additional capability that classes do not:

- Structures have an automatically-generated memberwise initializer, which you can use to initialize the member properties of new structure instances.

Structures are always copied when they are passed around in your code, and do not use reference counting.

definition

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

Stored properties are constants or variables that are bundled up and stored as part of the class or structure. Ex: width, height, resolution, …

Initialising

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

Unlike Objective-C, Swift enables you to set sub-properties of a structure property directly. Hence this is possible:

```
someVideoMode.resolution.width = 1280
```

62. Structs - **Memberwise Initialization**

All structures have an automatically-generated memberwise initializer, which you can use to initialize the member properties of new structure instances.

```
let vga = Resolution(width: 640, height: 480)
```

Unlike structures, class instances do not receive a default memberwise initializer.

63. Structs & Enums - **Value Types**

A value type is a type whose value is copied when it is assigned to a variable or constant, or when it is passed to a function.

In fact, all of the basic types in Swift—integers, floating-point numbers, Booleans, strings, arrays, dictionaries, structs and enums—are value types, and are implemented as structures behind the scenes.

64. Classes - **Reference Types**

As reference types, a reference to the same existing instance is used instead.

```
let tenEighty = VideoMode()
```

```
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0

let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Even though tenEighty and alsoTenEighty are declared as constants, you can still change tenEighty.frameRate and alsoTenEighty.frameRate because the values of the tenEighty and alsoTenEighty constants themselves do not actually change. **tenEighty and alsoTenEighty themselves do not "store" the VideoMode instance—instead, they both refer to a VideoMode instance behind the scenes**. It is the frameRate property of the underlying VideoMode that is changed, not the values of the constant references to that VideoMode.

65. Classes - **identity operators === & !===**

identity operators are useful to find out if two constants or variables refer to exactly the same instance of a class. === and !==

Ex: With reference to instances of classes of VideoMode created just above

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
}
```

"identical to" (===) does not mean the same thing as "equal to" (==):

"Identical to" means that two constants or variables of class type refer to exactly the same class instance.

"Equal to" means that two instances are considered "equal" or "equivalent" in value, for some appropriate meaning of "equal", as defined by the type's designer.

66. Structs and Classes - Choosing them

As a general guideline, consider creating a structure when one or more of these conditions apply:

- The structure's primary purpose is to encapsulate a few relatively simple data values.
- It is reasonable to expect that the 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.
- The structure does not need to inherit properties or behavior from another existing type.

67. Strings, Arrays, and Dictionaries VS NSString, NSArray, and NSDictionary

strings, arrays, and dictionaries are copied when they are assigned to a new constant or variable, or when they are passed to a function or method.

NSString, NSArray, and NSDictionary are implemented as classes, not structures. Strings, arrays, and dictionaries in Foundation are always assigned and passed around as a reference to an existing instance, rather than as a copy.

---

68. Properties - **stored properties**
    Stored properties store constant and variable values as part of an instance
    Stored properties are provided only by classes and structures

69. Properties - **computed properties**
    Computed properties calculate (rather than store) a value.
    Computed properties are provided by classes, structures, and enumerations

70. Properties - **Type properties**
    Properties that are associated with the type itself

71. Properties - **property observers**
    define property observers to monitor changes in a property's value, which you can respond to with custom actions. Property observers can be added to stored properties you define yourself, and also to properties that a subclass inherits from its superclass

72. Properties - Stored Properties and Structs
    If you create an instance of a structure and assign that instance to a constant, you cannot modify the instance's properties, even if they were declared as variable properties. This behavior is due to structures being value types. When an instance of a value type is marked as a constant, so are all of its properties.

```
struct Person {
   var name = ""
   var age = 1
}
let myself = Person(name: "PP", age: 25)
myself.name = "Prabhu"        // compiler will give error stating that “myself” has to be a variable
```

73. Properties - **Lazy**
    A lazy stored property is a property whose initial value is not calculated until the first time it is used. You indicate a lazy stored property by writing the lazy modifier before its declaration.

    You must always declare a lazy property as a variable (with the var keyword), because its initial value might not be retrieved until after instance initialization completes. Constant properties must always have a value before initialization completes, and therefore cannot be declared as lazy.

    If a property marked with the lazy modifier is accessed by multiple threads simultaneously and the property has not yet been initialized, there is no guarantee that the property will be initialized only once.

74. Properties - **Computed Properties**
    computed properties do not actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.
    In the below example, “center” is a computed property

```
struct Point {
   var x = 0.0, y = 0.0
}
struct Size {
   var width = 0.0, height = 0.0
}
struct Rect {
   var origin = Point()
   var size = Size()
   var center: Point {
      get {
         let centerX = origin.x + (size.width / 2)
         let centerY = origin.y + (size.height / 2)
         return Point(x: centerX, y: centerY)
      }
      set(newCenter) {
         origin.x = newCenter.x - (size.width / 2)
         origin.y = newCenter.y - (size.height / 2)
      }
   }
}
```

    Shorthand Setter Declaration

If a computed property's setter does not define a name for the new value to be set, a default name of **newValue** is used. In the above example, "newCenter" can be removed and instead "newValue" should be used

75. Properties - **Read-Only Computed Properties**

A computed property with a getter but no setter is known as a read-only computed property. A read-only computed property always returns a value, and can be accessed through dot syntax, but cannot be set to a different value.

You must declare computed properties—including read-only computed properties—as variable properties with the var keyword, because their value is not fixed.

76. Properties - **Property Observers**

Property observers observe and respond to changes in a property's value. Property observers are called every time a property's value is set.

You can add property observers to any stored properties you define, apart from lazy stored properties. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass.

You don't need to define property observers for non-overridden computed properties, because you can observe and respond to changes to their value in the computed property's setter.

willSet is called just before the value is stored.

didSet is called immediately after the new value is stored.

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
```

77. Properties - **Global Variables**

Unlike lazy stored properties, global constants and variables do not need to be marked with the lazy modifier.

78. Properties - **Type Properties**

There will only ever be one copy of these properties, no matter how many instances of that type you create. These kinds of properties are called type properties.

Type properties are useful for defining values that are universal to all instances of a particular type, such as a constant property that all instances can use (like a static constant in C), or a variable property that stores a value that is global to all instances of that type (like a static variable in C).

Unlike stored instance properties, you must always give stored type properties a default value. This is because the type itself does not have an initializer that can assign a value to a stored type property at initialization time.

Stored type properties are lazily initialized on their first access. They are guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they do not need to be marked with the lazy modifier.

You define type properties with the **static** keyword.

79. **Methods**

The fact that structures and enumerations can define methods in Swift is a major difference from C and Objective-C. In Swift, you can choose whether to define a class, structure, or enumeration, and still have the flexibility to define methods on the type you create.

80. Methods - **Instance methods**

Instance methods are functions that belong to instances of a particular class, structure, or enumeration. An instance method has implicit access to all other instance methods and properties of that type. An instance method can be called only on a specific instance of the type it belongs to.

81. Methods - **Local and External Params**

Swift gives the first parameter name in a method a local parameter name by default, and gives the second and subsequent parameter names both local and external parameter names by default.

82. Methods - **Self Property**

Every instance of a type has an implicit property called self, which is exactly equivalent to the instance itself. You use the self property to refer to the current instance within its own instance methods.

83. Methods - **Mutating**

If you need to modify the properties of your structure or enumeration within a particular method, you can opt in to mutating behavior for that method. The method can then mutate (that is, change) its properties from within the method, and any changes that it makes are written back to the original structure when the method ends.
The method can also assign a completely new instance to its implicit self property, and this new instance will replace the existing one when the method ends.

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")          // prints "The point is now at (3.0, 4.0)
```

Note: that you cannot call a mutating method on a constant of structure type, because its properties cannot be changed, even if they are variable properties

84. Methods - **Mutating & Self**

Mutating methods can assign an entirely new instance to the implicit self property. The Point example shown above could have been written in the following way instead:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

85. Methods - **Type Methods**

You can also define methods that are called on the type itself. These kinds of methods are called type methods. You indicate type methods by writing the **static** keyword before the method's func keyword.

Classes may also use the **class** keyword to allow subclasses to override the superclass's implementation of that method. Ex:

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

Using static in class type methods and trying to override the func would result in error as "static" means final and cannot be overridden. Hence we use "class" if overriding functionality is to be given

86. **Subscript**

You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. You can define multiple subscripts for a single type, and the appropriate subscript overload to use is selected based on the type of index value you pass to the subscript. Subscripts are not limited to a single dimension, and you can define subscripts with multiple input parameters to suit your custom type's needs.

subscripts can be read-write or read-only.

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

```
if you wanna make it as a read-only
subscript(index: Int) -> Int {
        // return an appropriate subscript value here
}
```

```
Ex: “struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")        // prints "six times three is 18
```

87. Subscripts - **Subscripts Options**

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return any type.

Subscripts can use variable parameters and variadic parameters, but cannot use in-out parameters or provide default parameter values.

While it is most common for a subscript to take a single parameter, you can also define a subscript with multiple parameters if it is appropriate for your type.

88. Subscripts - **Overloading**

A class or structure can provide as many subscript implementations as it needs, and the appropriate subscript to be used will be inferred based on the types of the value or values that are contained within the subscript braces at the point that the subscript is used. This definition of multiple subscripts is known as subscript overloading.

---

89. **Inheritance**

Classes can also add property observers to inherited properties in order to be notified when the value of a property changes. Property observers can be added to any property, regardless of whether it was originally defined as a stored or computed property.

90. Inheritance - **Subclassing**

Subclassing is the act of basing a new class on an existing class. The subclass inherits characteristics from the existing class, which you can then refine. You can also add new characteristics to the subclass.

91. Inheritance - **Overriding**

A subclass can provide its own custom implementation of an instance method, type method, instance property, type property, or subscript that it would otherwise inherit from a superclass. This is known as overriding.

To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the **override** keyword.

92. Inheritance - **Accessing super class**

Where this is appropriate, you access the superclass version of a method, property, or subscript by using the **super** prefix

93. Inheritance - **Overriding methods**

You can override an inherited instance or type method to provide a tailored or alternative implementation of the method within your subclass.

94. Inheritance - **Overriding properties**

You can override an inherited instance or type property to provide your own custom getter and setter for that property, or to add property observers to enable the overriding property to observe when the underlying property value changes.

95. Inheritance - **Overriding property get and set**

You can provide a custom getter (and setter, if appropriate) to override any inherited property, regardless of whether the inherited property is implemented as a stored or computed property at source.

You can present an inherited read-only property as a read-write property by providing both a getter and a setter in your subclass property override. You cannot, however, present an inherited read-write property as a read-only property.

If you provide a setter as part of a property override, you must also provide a getter for that override. If you don't want to modify the inherited property's value within the overriding getter, you can simply pass through the inherited value by returning super.someProperty from the getter, where someProperty is the name of the property you are overriding.

96. Inheritance - **Overriding property observers**

You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented.

You cannot add property observers to inherited constant stored properties or inherited read-only computed properties. The value of these properties cannot be set, and so it is not appropriate to provide a willSet or didSet implementation as part of an override.

Note also that you cannot provide both an overriding setter and an overriding property observer for the same property. If you want to observe changes to a property's value, and you are already providing a custom setter for that property, you can simply observe any value changes from within the custom setter.

97. Inheritance - **Final**

You can prevent a method, property, or subscript from being overridden by marking it as **final**. Do this by writing the final modifier before the method, property, or subscript's introducer keyword (such as final var, final func, final class func, and final subscript).

Methods, properties, or subscripts that you add to a class in an extension can also be marked as final within the extension's definition.

You can mark an entire class as final by writing the final modifier before the class keyword in its class definition (final class). Any attempt to subclass a final class is reported as a compile-time error.

98. **Initialization**

Swift initializers do not return a value. Their primary role is to ensure that new instances of a type are correctly initialized before they are used for the first time.

Instances of class types can also implement a deinitializer, which performs any custom cleanup just before an instance of that class is deallocated.

```
init() {
    // perform some initialization here
}
```

99. Initialization - **Stored Properties**

Classes and structures must set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created. Stored properties cannot be left in an indeterminate state.

You can set an initial value for a stored property within an initializer, or by assigning a default property value as part of the property's definition.
When you assign a default value to a stored property, or set its initial value within an initializer, the value of that property is set directly, without calling any property observers.

100. Initialization - **Custom init()**

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}
let bodyTemperature = Celsius(fromFahrenheit: 212.0)
let bodyTemperature2 = Celsius(37.0)
```

101. Initialization - **Optional Property Types**

Properties of optional type are automatically initialized with a value of nil, indicating that the property is deliberately intended to have "no value yet" during initialization.

102. Initialization - **Assigning value to Constant Properties During Initialization**

You can assign a value to a constant property at any point during initialization, as long as it is set to a definite value by the time initialization finishes. Once a constant property is assigned a value, it can't be further modified.

For class instances, a constant property can only be modified during initialization by the class that introduces it. It cannot be modified by a subclass.

103. Initialization - **Default Initializers**

Swift provides a default initializer for any structure or class that provides default values for all of its properties and does not provide at least one initializer itself. The default initializer simply creates a new instance with all of its properties set to their default values.

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
```

104. Initialization - **Memberwise Initializers for Structure Types**

Structure types automatically receive a memberwise initializer if they do not define any of their own custom initializers. Unlike a default initializer, the structure receives a memberwise initializer even if it has stored properties that do not have default values.

Initial values for the properties of the new instance can be passed to the memberwise initializer by name.

```
struct Size {
   var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

105. Initialization - **Initializer Delegation for Value Types**

Initializers can call other initializers to perform part of an instance's initialization. This process, known as initializer delegation, avoids duplicating code across multiple initializers.

Value types (structures and enumerations) do not support inheritance, and so their initializer delegation process is relatively simple, because they can only delegate to another initializer that they provide themselves.
Classes can inherit from other classes. This means that classes have additional responsibilities for ensuring that all stored properties they inherit are assigned a suitable value during initialization.

If you define a custom initializer for a value type, you will no longer have access to the default initializer (or the memberwise initializer, if it is a structure) for that type. However, If you still want to access the default initializer and memberwise initializer, with your own custom initializers, write your custom initializers in an extension rather than as part of the value type's original implementation.

106. Initialization - **Class Inheritance and Initialization - Designated Initializers and Convenience Initializers**

Designated initializers are the primary initializers for a class. A designated initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.

Every class must have at least one designated initializer.

Convenience initializers are secondary, supporting initializers for a class. You can define a convenience initializer to call a designated initializer from the same class as the convenience initializer with some of the designated initializer's parameters set to default values.

You can also define a convenience initializer to create an instance of that class for a specific use case or input value type. You do not have to provide convenience initializers if your class does not require them.

107. Initialization - **Syntax Designated & Convenience Initializers**

Designated initializers

```
 init(parameters) {
   statements
}
```

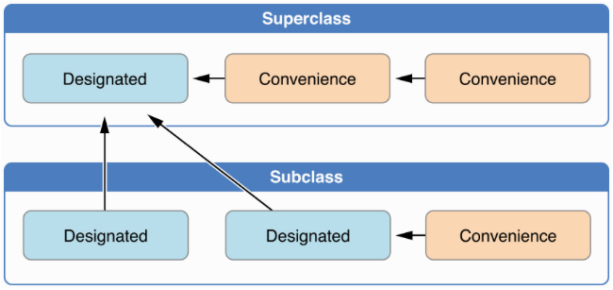
Convenience initializers

```
convenience init(parameters) {
   // statements
}
```

108. Initialization - **Delegations for Initializers**

Rule 1: A designated initializer must call a designated initializer from its immediate superclass.
Rule 2: A convenience initializer must call another initializer from the same class.
Rule 3: A convenience initializer must ultimately call a designated initializer.

A simple way to remember this is:
Designated initializers must always delegate up.
Convenience initializers must always delegate across.



109. Initialization - **2 Phase Init**

Class initialization in Swift is a two-phase process. In the first phase, each stored property is assigned an initial value by the class that introduced it. Once the initial state for every stored property has been determined, the second phase begins, and each class is given the opportunity to customize its stored properties further before the new instance is considered ready for use.

The class instance is not fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase.

Here's how two-phase initialization plays out,

Phase 1

1. A designated or convenience initializer is called on a class.
2. Memory for a new instance of that class is allocated. The memory is not yet initialized.
3. A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized.
4. The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties.
5. This continues up the class inheritance chain until the top of the chain is reached.
6. Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

Phase 2

1. Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further. Initializers are now able to access self and can modify its properties, call its instance methods, and so on.
2. Any convenience initializers in the chain have the option to customize the instance and to work with self.

110. Initialization - **Initializer Inheritance and Overriding**

Unlike subclasses in Objective-C, Swift subclasses do not inherit their superclass initializers by default. Swift's approach prevents a situation in which a simple initializer from a superclass is inherited by a more specialized subclass and is used to create a new instance of the subclass that is not fully or correctly initialized.
Superclass initializers are inherited in certain circumstances, but only when it is safe and appropriate to do so.

If you want a custom subclass to present one or more of the same initializers as its superclass, you can provide a custom implementation of those initializers within the subclass.

When you write a subclass initializer that matches a superclass designated initializer, you are effectively providing an override of that designated initializer. Therefore, you must write the override modifier before the subclass's initializer definition.

You always write the override modifier when overriding a superclass designated initializer, even if your subclass's implementation of the initializer is a convenience initializer.

Conversely, if you write a subclass initializer that matches a superclass convenience initializer, that superclass convenience initializer can never be called directly by your subclass. As a result, you do not write the override modifier when providing a matching implementation of a superclass convenience initializer.

111. Initialization - **Automatic Initializer Inheritance**

Subclasses do not inherit their superclass initializers by default. However, superclass initializers are automatically inherited if certain conditions are met.

Rule 1

If your subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

Rule 2

If your subclass provides an implementation of all of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

A subclass can implement a superclass designated initializer as a subclass convenience initializer

Example

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}

class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

The init(name: String) convenience initializer provided by RecipeIngredient takes the same parameters as the init(name: String) designated initializer from Food. Because this convenience initializer overrides a designated initializer from its superclass, it must be marked with the override modifier

Even though RecipeIngredient provides the init(name: String) initializer as a convenience initializer, RecipeIngredient has nonetheless provided an implementation of all of its superclass's designated initializers. Therefore, RecipeIngredient automatically inherits all of its superclass's convenience initializers too.

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

ShoppingListItem does not define an initializer to provide an initial value for purchased, because items in a shopping list (as modelled here) always start out unpurchased.

112. Initialization - **Failable Initializers**

It is sometimes useful to define a class, structure, or enumeration for which initialization can fail. This failure might be triggered by invalid initialization parameter values, the absence of a required external resource, or some other condition that prevents initialization from succeeding.

To cope with initialization conditions that can fail, define one or more failable initializers as part of a class, structure, or enumeration definition. Write a failable initializer by placing a question mark after the init keyword (**init?**).

A failable and a nonfailable initializer cannot be defined with the same parameter types and names.

A failable initializer creates an optional value of the type it initializes. Write "return nil" within a failable initializer to indicate a point at which initialization failure can be triggered.

Initializers do not return a value. Rather, their role is to ensure that self is fully and correctly initialized by the time that initialization ends. To indicate initialization success, "return" keyword is not used.

```
struct Animal {
```

```
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}

let someCreature = Animal(species: "Giraffe")

if let giraffe = someCreature {
    print("An animal was initialized with a species of \(giraffe.species)")
}
```

113. Initialization - Failable Initializers - **Enumerations**

Use a failable initializer to select an appropriate enumeration case based on one or more parameters. The initializer can then fail if the provided parameters do not match an appropriate enumeration case.

```
enum TemperatureUnit {
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
            return nil
        }
    }
}

let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
```

114. Initialization - Failable Initializers - Enumerations - **Raw values**

Enumerations with raw values automatically receive a failable initializer, init?(rawValue:), that takes a parameter called rawValue of the appropriate raw-value type and selects a matching enumeration case if one is found, or triggers an initialization failure if no matching value exists.

From the above example, it can be rewritten as

```
enum TemperatureUnit: Character {
    case Kelvin = "K", Celsius = "C", Fahrenheit = "F"
}

let fahrenheitUnit = TemperatureUnit(rawValue: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
```

115. Initialization - Failable Initializers - **Propagation**

A failable initializer of a class, structure, or enumeration can delegate across to another failable initializer from the same class, structure, or enumeration. Similarly, a subclass failable initializer can delegate up to a superclass failable initializer.

In either case, if delegated to another initializer that causes initialization to fail, the entire initialization process fails immediately, and no further initialization code is executed.

A failable initializer can also delegate to a nonfailable initializer. Use this approach if needed to add a potential failure state to an existing initialization process that does not otherwise fail.

116. Initialization - Failable Initializers - **Overriding**

You can override a superclass failable initializer in a subclass, just like any other initializer. Alternatively, you can override a superclass failable initializer with a subclass nonfailable initializer. This enables you to define a subclass for which initialization cannot fail, even though initialization of the superclass is allowed to fail.

You can override a failable initializer with a nonfailable initializer but not the other way around.

```
class Document {
    var name: String?
    init() {}
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {
        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"
        } else {
            self.name = name
        }
    }
}
```

117. Initialization - Failable Initializers - **init!**

You typically define a failable initializer that creates an optional instance of the appropriate type by placing a question mark after the init keyword (init?). Alternatively, you can define a failable initializer that creates an implicitly unwrapped optional instance of the appropriate type. Do this by placing an exclamation mark after the init keyword (init!) instead of a question mark.

You can delegate from init? to init! and vice versa, and you can override init? with init! and vice versa. You can also delegate from init to init!, although doing so will trigger an assertion if the init! initializer causes initialization to fail.

118. Initialization - **Required Initializers**

Write the required modifier before the definition of a class initializer to indicate that every subclass of the class must implement that initializer:

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}
```

You must also write the required modifier before every subclass implementation of a required initializer, to indicate that the initializer requirement applies to further subclasses in the chain. You do not write the override modifier when overriding a required designated initializer:

```
class SomeSubclass: SomeClass {
    required init() {
        // subclass implementation of the required initializer goes here
    }
}
```

You do not have to provide an explicit implementation of a required initializer if you can satisfy the requirement with an inherited initializer.

119. Initialization - **Default Property Value with a Closure or Function**

If a stored property's default value requires some customization or setup, you can use a closure or global function to provide a customized default value for that property. Whenever a new instance of the type that the property belongs to is initialized, the closure or function is called, and its return value is assigned as the property's default value.

```
class SomeClass {
    let someProperty: SomeType = {
        // create a default value for someProperty inside this closure
        // someValue must be of the same type as SomeType
        return someValue
    }()
}
```

Note that the closure's end curly brace is followed by an empty pair of parentheses. This tells Swift to execute the closure immediately. If you omit these parentheses, you are trying to assign the closure itself to the property, and not the return value of the closure.

If you use a closure to initialize a property, remember that the rest of the instance has not yet been initialized at the point that the closure is executed. This means that you cannot access any other property values from within your closure, even if those properties have default values. You also cannot use the implicit self property, or call any of the instance's methods.

---

120. **Deinitialization**

A deinitializer is called immediately before a class instance is deallocated. You write deinitializers with the deinit keyword, similar to how initializers are written with the init keyword. Deinitializers are only available on class types.

Class definitions can have at most one deinitializer per class. The deinitializer does not take any parameters and is written without parentheses:

```
deinit {
    // perform the deinitialization
}
```

Deinitializers are called automatically, just before instance deallocation takes place. You are not allowed to call a deinitializer yourself.
Superclass deinitializers are inherited by their subclasses, and the superclass deinitializer is called automatically at the end of a subclass deinitializer implementation. Superclass deinitializers are always called, even if a subclass does not provide its own deinitializer.

Because an instance is not deallocated until after its deinitializer is called, a deinitializer can access all properties of the instance it is called on and can modify its behavior based on those properties.

If no other properties or variables are still referring to the instance that has become nil, it is deallocated in order to free up its memory. Just before this happens, its deinitializer is called automatically.

---

121. **ARC**

Reference counting only applies to instances of classes. Structures and enumerations are value types, not reference types, and are not stored and passed by reference.

122. ARC - How it works

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances do not take up space in memory when they are no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don't disappear while they are still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a strong reference to the instance. The reference is called a "**strong**" reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains.

123. ARC - **Strong reference cycle**

if two class instances hold a strong reference to each other, such that each instance keeps the other alive. This is known as a strong reference cycle.

124. ARC - **Resolving Strong reference cycles**

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: **weak references and unowned references**.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance without keeping a strong hold on it. The instances can then refer to each other without creating a strong reference cycle.

Use a **weak reference** whenever it is valid for that reference to become nil at some point during its lifetime. Conversely, use an **unowned reference** when you know that the reference will never be nil once it has been set during initialization.

125. ARC - **weak reference**

A weak reference is a reference that does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance. This behavior prevents the reference from becoming part of a strong reference cycle. You indicate a weak reference by placing the weak keyword before a property or variable declaration.

Use a weak reference to avoid reference cycles whenever it is possible for that reference to have a missing value at some point in its life.

Weak references must be declared as variables, to indicate that their value can change at runtime. A weak reference cannot be declared as a constant.

Because a weak reference does not keep a strong hold on the instance it refers to, it is possible for that instance to be deallocated while the weak reference is still referring to it.
Therefore, ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated.
Because weak references need to allow nil as their value, they always have an optional type.
You can check for the existence of a value in the weak reference, just like any other optional value, and you will never end up with a reference to an invalid instance that no longer exists.

126. ARC - **unowned reference**

Unowned reference does not keep a strong hold on the instance it refers to. An unowned reference is assumed to always have a value.

Because of this, an unowned reference is always defined as a nonoptional type. You indicate an unowned reference by placing the **unowned** keyword before a property or variable declaration.

Because an unowned reference is nonoptional, you don't need to unwrap the unowned reference each time it is used. An unowned reference can always be accessed directly.
However, ARC cannot set the reference to nil when the instance it refers to is deallocated, because variables of a nonoptional type cannot be set to nil.

If you try to access an unowned reference after the instance that it references is deallocated, you will trigger a runtime error. Use unowned references only when you are sure that the reference will always refer to an instance.

127. ARC - **Unowned References and Implicitly Unwrapped Optional Properties**
it is useful to combine an unowned property on one class with an implicitly unwrapped optional property on the other class.
This enables both properties to be accessed directly (without optional unwrapping) once initialization is complete, while still avoiding a reference cycle.

```
class Country {
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

The initializer for City takes a Country instance, and stores this instance in its country property.
The initializer for City is called from within the initializer for Country.

Declare the capitalCity property of Country as an implicitly unwrapped optional property. This means that the capitalCity property has a default value of nil, like any other optional, but can be accessed without the need to unwrap its value

Because capitalCity has a default nil value, a new Country instance is considered fully initialized as soon as the Country instance sets its name property within its initializer. This means that the Country initializer can start to reference and pass around the implicit self property as soon as the name property is set. The Country initializer can therefore pass self as one of the parameters for the City initializer when the Country initializer is setting its own capitalCity property.

you can create the Country and City instances in a single statement, without creating a strong reference cycle, and the capitalCity property can be accessed directly, without needing to use an exclamation mark to unwrap its optional value

128. ARC - **Closure and capture list**
A strong reference cycle can also occur if you assign a closure to a property of a class instance, and the body of that closure captures the instance. This capture might occur because the closure's body accesses a property of the instance, such as self.someProperty, or because the closure calls a method on the instance, such as self.someMethod(). In either case, these accesses cause the closure to "capture" self, creating a strong reference cycle.
This strong reference cycle occurs because closures, like classes, are reference types. When you assign a closure to a property, you are assigning a reference to that closure. In essence, it's the same problem as above—two strong references are keeping each other alive. However, rather than two class instances, this time it's a class instance and a closure that are keeping each other alive.

```
class HTMLElement {
```

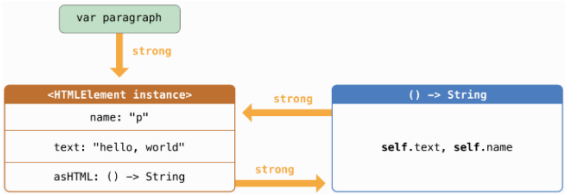
```
    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }
}

var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
```



If you set the paragraph variable to nil and break its strong reference to the HTMLElement instance, neither the HTMLElement instance nor its closure are deallocated, because of the strong reference cycle

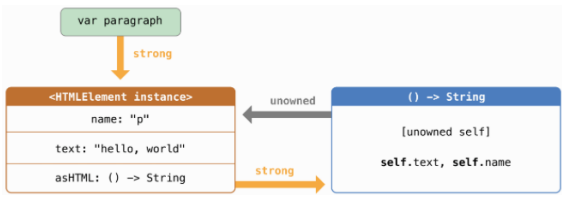
You resolve a strong reference cycle between a closure and a class instance by defining a **capture list** as part of the closure's definition. A capture list defines the rules to use when capturing one or more reference types within the closure's body. As with strong reference cycles between two class instances, you declare each captured reference to be a weak or unowned reference rather than a strong reference.\

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!] (index: Int, stringToProcess: String) -> String in
    // closure body goes here
}
OR
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
}
```

The usage of weak and unowned is as per properties in a class/reference type

above code for asHTML should change to

```
lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
}
```



129. **Optional Chaining**

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil.
If the optional contains a value, the property, method, or subscript call succeeds; if the optional is nil, the property, method, or subscript call returns nil.
Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is nil.

You specify optional chaining by placing a **?** after the optional value on which you wish to call a property, method or subscript if the optional is non-nil.

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
```

You can link together multiple levels of optional chaining to drill down to properties, methods, and subscripts deeper within a model. However, multiple levels of optional chaining do not add more levels of optionality to the returned value.

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
```

---

130. **Error Handling**

Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.
Error handling in Swift interoperates with error handling patterns that use the NSError class in Cocoa and Objective-C.

131. Error Handling - **ErrorType protocol**

In Swift, errors are represented by values of types that conform to the **ErrorType** protocol. This empty protocol indicates that a type can be used for error handling.
Swift enumerations are particularly well suited to modeling a group of related error conditions, with associated values allowing for additional information about the nature of an error to be communicated.

```
enum VendingMachineError: ErrorType {
   case InvalidSelection
   case InsufficientFunds(coinsNeeded: Int)
   case OutOfStock
}
```

Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue. You use a throw statement to throw an error.

```
throw VendingMachineError.InsufficientFunds(coinsNeeded: 5)
```

There are four ways to handle errors in Swift.

1. You can propagate the error from a function to the code that calls that function
2. handle the error using a do-catch statement
3. handle the error as an optional value
4. assert that the error will not occur.

write the **try** keyword—or the **try?** or **try!** variation—before a piece of code that calls a function, method, or initializer that can throw an error.
Error handling in Swift resembles exception handling in other languages, with the use of the **try**, **catch** and **throw** keywords.

132. Error Handling - **Propagating Errors Using Throwing Functions**

To indicate that a function, method, or initializer can throw an error, you write the **throws** keyword in the function's declaration after its parameters. A function marked with **throws** is called a **throwing function**. If the function specifies a return type, you write the throws keyword before the return arrow (->).

func canThrowErrors() throws -> String

A throwing function propagates errors that are thrown inside of it to the scope from which it's called.

Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.

```
struct Item {
   var price: Int
   var count: Int
}

class VendingMachine {
   var inventory = [
      "Candy Bar": Item(price: 12, count: 7),
      "Chips": Item(price: 10, count: 4),
      "Pretzels": Item(price: 7, count: 11)
   ]
   var coinsDeposited = 0
   func dispenseSnack(snack: String) {
      print("Dispensing \(snack)")
   }

   func vend(itemNamed name: String) throws {
      guard let item = inventory[name] else {
```

```
            throw VendingMachineError.InvalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.OutOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.InsufficientFunds(coinsNeeded: item.price - coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        dispenseSnack(name)
    }
}

let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]

func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

133. Error Handling - **Do-Catch**

You use a do-catch statement to handle errors by running a block of code. If an error is thrown by the code in the do clause, it is matched against the catch clauses to determine which one of them can handle the error.

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack("Alice", vendingMachine: vendingMachine)
} catch VendingMachineError.InvalidSelection {
    …
} catch VendingMachineError.OutOfStock {
    …
} catch VendingMachineError.InsufficientFunds(let coinsNeeded) {
    …
}
```

You write a pattern after catch to indicate what errors that clause can handle. If a catch clause doesn't have a pattern, the clause matches any error and binds the error to a local constant named error.

The catch clauses don't have to handle every possible error that the code in its do clause can throw. If none of the catch clauses handle the error, the error propagates to the surrounding scope. However, the error must be handled by some surrounding scope—either by an enclosing do-catch clause that handles the error or by being inside a throwing function.

134. Error Handling - **Optional Values**

You use **try?** to handle an error by converting it to an optional value. If an error is thrown while evaluating the try? expression, the value of the expression is nil.

```
func someThrowingFunction() throws -> Int {
    // ...
}

let x = try? someThrowingFunction()
```

OR

```
let y: Int?
do {
   y = try someThrowingFunction()
} catch {
   y = nil
}

func fetchData() -> Data? {
   if let data = try? fetchDataFromDisk() { return data }
   if let data = try? fetchDataFromServer() { return data }
   return nil
}
```

135. Error Handling - **Optional unwrapped values**

Sometimes you know a throwing function or method won't, in fact, throw an error at runtime. On those occasions, you can write **try!** before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you'll get a runtime error.

Example, the following code uses a loadImage(_:) function, which loads the image resource at a given path or throws an error if the image can't be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it is appropriate to disable error propagation.

```
let photo = try! loadImage("./Resources/John Appleseed.jpg")
```

---

136. **Cleanup - Defer**

You use a defer statement to execute a set of statements just before code execution leaves the current block of code. This statement lets you do any necessary cleanup that should be performed regardless of how execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as return or break.

A defer statement defers execution until the current scope is exited. This statement consists of the defer keyword and the statements to be executed later.
The deferred statements may not contain any code that would transfer control out of the statements, such as a break or a return statement, or by throwing an error.

Deferred actions are executed in reverse order of how they are specified—that is, the code in the first defer statement executes after code in the second, and so on.

You can use a defer statement even when no error handling code is involved.

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

137. **Type Casting**

Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

Type casting in Swift is implemented with the **is** and **as** operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.

You can use type casting with a hierarchy of classes and subclasses to check the type of a particular class instance and to cast that instance to another class within the same hierarchy.

138. Type Casting - **Type Checking "is"**

Use the type check operator (is) to check whether an instance is of a certain subclass type. The type check operator returns true if the instance is of that subclass type and false if it is not.

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}

class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes")
]

var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}
```

139. Type Casting - **Downcast "as?" "as!"**

A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to downcast to the subclass type with a type cast operator (as? or as!).

Use the conditional form of the type cast operator (as?) when you are not sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be nil if the downcast was not possible. This enables you to check for a successful downcast.

Use the forced form of the type cast operator (as!) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.

Continuing from the above example of MediaItem and its subclasses and the library array

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}
```

Casting does not actually modify the instance or change its values. The underlying instance remains the same; it is simply treated and accessed as an instance of the type to which it has been cast.

140. Type Casting - **Any** and **AnyObject**

AnyObject can represent an instance of any class type.
Any can represent an instance of any type at all, including function types.

141. Type Casting - **AnyObject**

```
let someObjects: [AnyObject] = [
    Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),
    Movie(name: "Moon", director: "Duncan Jones"),
    Movie(name: "Alien", director: "Ridley Scott")
]

for movie in someObjects as! [Movie] {
    print("Movie: \(movie.name), dir. \(movie.director)")
}
```

142. Type Casting - **Any**

```
var things = [Any]()
things.append(0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))

for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
      case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    default:
        print("something else")
    }
```

143. **Nested Types**

Similarly, it can be convenient to define utility classes and structures purely for use within the context of a more complex type. To accomplish this, Swift enables you to define nested types, whereby you nest supporting enumerations, classes, and structures within the definition of the type they support.

To nest a type within another type, write its definition within the outer braces of the type it supports. Types can be nested to as many levels as are required.

```
struct BlackjackCard {

   // nested Suit enumeration
   enum Suit: Character {
      case Spades = "♠", Hearts = "♡", Diamonds = "♢", Clubs = "♣"
   }

   // nested Rank enumeration
   enum Rank: Int {
      case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
      case Jack, Queen, King, Ace
      struct Values {
         let first: Int, second: Int?
      }
      var values: Values {
         switch self {
         case .Ace:
            return Values(first: 1, second: 11)
         case .Jack, .Queen, .King:
            return Values(first: 10, second: nil)
         default:
            return Values(first: self.rawValue, second: nil)
         }
      }
   }

   // BlackjackCard properties and methods
   let rank: Rank, suit: Suit
   var description: String {
      var output = "suit is \(suit.rawValue),"
      output += " value is \(rank.values.first)"
      if let second = rank.values.second {
         output += " or \(second)"
      }
      return output
   }
}
```

---

144. **Extensions**

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as retroactive modeling).

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

In Swift, you can even extend a protocol to provide implementations of its requirements or add additional functionality that conforming types can take advantage of.

Extensions can add new functionality to a type, but they cannot override existing functionality.

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

If you define an extension to add new functionality to an existing type, the new functionality will be available on all existing instances of that type, even if they were created before the extension was defined.

Extensions can add new computed properties, but they cannot add stored properties, or add property observers to existing properties.

145. Extensions - **Computed Properties**

Extensions can add computed instance properties and computed type properties to existing types.

In the below example, a Double value of 1.0 is considered to represent “one meter”. This is why the computed property returns self—the expression 1.m is considered to calculate a Double value of 1.0.

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")          // Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")      // Prints "Three feet is 0.914399970739201 meters
```

146. Extensions - **Initializers**

Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type’s original implementation.

Extensions can add new convenience initializers to a class, but they cannot add new designated initializers or deinitializers to a class. Designated initializers and deinitializers must always be provided by the original class implementation.

If you use an extension to add an initializer to a value type that provides default values for all of its stored properties and does not define any custom initializers, you can call the default initializer and memberwise initializer for that value type from within your extension’s initializer.

Example

```
struct Size {
    var width = 0.0, height = 0.0
}
```

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}
```
Because the Rect structure provides default values for all of its properties, it receives a default initializer and a memberwise initializer automatically. These initializers can be used to create new Rect instances:
```
let defaultRect = Rect()
```
OR
```
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0), size: Size(width: 5.0, height: 5.0))
```

You can extend the Rect structure to provide an additional initializer that takes a specific center point and size:
```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```
This new initializer starts by calculating an appropriate origin point based on the provided center point and size value. The initializer then calls the structure's automatic memberwise initializer init(origin:size:), which stores the new origin and size values in the appropriate properties:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0), size: Size(width: 3.0, height: 3.0))
```

If you provide a new initializer with an extension, you are still responsible for making sure that each instance is fully initialized once the initializer completes.

147. Extensions - **Methods**

Extensions can add new instance methods and type methods to existing types.

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
}

3.repetitions() {
    print("Goodbye!")
}
```

148. Extensions - **Mutating Instance Methods**

Instance methods added with an extension can also modify (or mutate) the instance itself. Structure and enumeration methods that modify self or its properties must mark the instance method as mutating, just like mutating methods from an original implementation.

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()                // someInt is now 9
```

149. Extensions - **Subscripts**

Extensions can add new subscripts to an existing type. This example adds an integer subscript to Swift's built-in Int type. This subscript [n] returns the decimal digit n places in from the right of the number:

```
123456789[0] returns 9
123456789[1] returns 8
```

…and so on:

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}
746381295[0]  // returns 5
746381295[1]  // returns 9
```

If the Int value does not have enough digits for the requested index, the subscript implementation returns 0, as if the number had been padded with zeros to the left:

```
746381295[9]  // returns 0, as if you had requested:
```

150. Extensions - **Nested Types**

Extensions can add new nested types to existing classes, structures and enumerations:

This example adds a new nested enumeration to Int. This enumeration, called Kind, expresses whether the number is negative, zero, or positive. This example also adds a new computed instance property to Int, called kind, which returns the appropriate Kind enumeration case for that integer.

```
extension Int {
    enum Kind {
        case Negative, Zero, Positive
    }
    var kind: Kind {
        switch self {
        case 0:
            return .Zero
        case let x where x > 0:
            return .Positive
        default:
            return .Negative
        }
    }
}

func printIntegerKinds(numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .Negative:
            print("- ", terminator: "")
        case .Zero:
            print("0 ", terminator: "")
        case .Positive:
            print("+ ", terminator: "")
        }
    }
    print("")
}

printIntegerKinds([3, 19, -27, 0, -6, 0, 7])      // Prints "+ + - 0 - 0 +”
```

151. **Protocols**

A **protocol** defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol.

In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

```
protocol SomeProtocol {
    // protocol definition goes here
}

struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}

class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

152. Protocol - **Property Requirements**

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn't specify whether the property should be a stored property or a computed property—it only specifies the required property name and type. The protocol also specifies whether each property must be gettable or gettable and settable.

Property requirements are always declared as variable properties, prefixed with the var keyword. **Gettable and settable properties** are indicated by writing **{ get set }** after their type declaration, and **gettable properties** are indicated by writing **{ get }**.

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}

protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

Example:
Starship class implements the fullName property requirement as a computed read-only property for a starship.

```
protocol FullyNamed {
    var fullName: String { get }
}

class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: “USS")
print(ncc1701.fullName)        //Prints “USS Enterprise”
```

153. Protocol - **Method Requirements**

Protocols can require specific instance methods and type methods to be implemented by conforming types. These methods are written as part of the protocol's definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body. Variadic parameters are allowed, subject to the same rules as for normal methods.

Default values, however, cannot be specified for method parameters within a protocol's definition.

```
protocol SomeProtocol {
    static func someTypeMethod()
    func someInstanceMethod()
}
```

Example:

```
protocol RandomNumberGenerator {
    func random() -> Double
}

class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c) % m)
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.37464991998171"
print("And another one: \(generator.random())")
// Prints "And another one: 0.729023776863283”
```

154. Protocol - **Mutating Method Requirements**

If you define a protocol instance method requirement that is intended to mutate instances of any type that adopts the protocol, mark the method with the mutating keyword as part of the protocol's definition. This enables structures and enumerations to adopt the protocol and satisfy that method requirement.

If you mark a protocol instance method requirement as mutating, you do not need to write the mutating keyword when writing an implementation of that method for a class. The mutating keyword is only used by structures and enumerations.

```
protocol Togglable {
    mutating func toggle()
}

enum OnOffSwitch: Togglable {
    case Off, On
    mutating func toggle() {
        switch self {
        case Off:
            self = On
        case On:
            self = Off
        }
    }
}
var lightSwitch = OnOffSwitch.Off
lightSwitch.toggle()
```

155. Protocol - **Initializer Requirements**

Protocols can require specific initializers to be implemented by conforming types.

```
protocol SomeProtocol {
    init(someParameter: Int)
```

```
}
```
You can implement a protocol initializer requirement on a conforming class as either a designated initializer or a convenience initializer.
In both cases, you must mark the initializer implementation with the **required** modifier

```
class SomeClass: SomeProtocol {
    required init(someParameter: Int) {
        // initializer implementation goes here
    }
}
```

The use of the required modifier ensures that you provide an explicit or inherited implementation of the initializer requirement on all subclasses of the conforming class, such that they also conform to the protocol.

You do not need to mark protocol initializer implementations with the required modifier on classes that are marked with the final modifier, because final classes cannot be subclassed.

If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the **required** and **override** modifiers:

```
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}
```

Protocols can define failable initializer requirements for conforming types

156. Protocols as **Types**

Protocols do not actually implement any functionality themselves. Nonetheless, any protocol you create will become a fully-fledged type for use in your code.

Because it is a type, you can use a protocol in many places where other types are allowed, including:

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

From below example, since the class LinearCongruentialGenerator conforms to RandomNumberGenerator and it gives us the functionality required for this class to generate a random number between 0.0 and 1.0, we would use that class.

```
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    print("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

157. Protocols - **Delegations**

Delegation is a design pattern that enables a class or structure to hand off (or delegate) some of its responsibilities to an instance of another type.
This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated.
Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

158. Protocols - **Conformance with Extensions**

You can extend an existing type to adopt and conform to a new protocol, even if you do not have access to the source code for the existing type. Extensions can add new properties, methods, and subscripts to an existing type, and are therefore able to add any requirements that a protocol may demand.

Existing instances of a type automatically adopt and conform to a protocol when that conformance is added to the instance's type in an extension.

For example, this protocol, called TextRepresentable, can be implemented by any type that has a way to be represented as text.

```
protocol TextRepresentable {
    var textualDescription: String { get }
}

extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}
```

This extension adopts the new protocol in exactly the same way as if Dice had provided it in its original implementation.

159. Protocols - **Adoption with Extensions**

If a type already conforms to all of the requirements of a protocol, but has not yet stated that it adopts that protocol, you can make it adopt the protocol with an empty extension:

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

Instances of Hamster can now be used wherever TextRepresentable is the required type:

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
print(somethingTextRepresentable.textualDescription)
// Prints "A hamster named Simon"
```

160. Protocols - **Collection Types**

A protocol can be used as the type to be stored in a collection such as an array or a dictionary, as mentioned in Protocols as Types. This example creates an array of TextRepresentable things:

let things: [TextRepresentable] = [game, d12, simonTheHamster]

"game", "d12", "simonTheHamster" are all objects of instances of different classes that conforms to the protocol "TextRepresentable". It is now possible to iterate over the items in the array, and print each item's textual description:

```
for thing in things {
    print(thing.textualDescription)
}
```

Note that the thing constant is of type TextRepresentable. It is not of type Dice, or DiceGame, or Hamster, even if the actual instance behind the scenes is of one of those types. Nonetheless, because it is of type TextRepresentable, and anything that is TextRepresentable is known to have a textualDescription property, it is safe to access thing.textualDescription each time through the loop.

161. Protocols - **Protocol Inheritance**

A protocol can inherit one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

Here's an example of a protocol that inherits the TextRepresentable protocol from above:

```
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

162. Protocols - **Class-Only**

You can limit protocol adoption to class types (and not structures or enumerations) by adding the **class** keyword to a protocol's inheritance list. The **class** keyword must always appear first in a protocol's inheritance list, before any inherited protocols:

```
protocol SomeClassOnlyProtocol: class, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

In the example above, SomeClassOnlyProtocol can only be adopted by class types. It is a compile-time error to write a structure or enumeration definition that tries to adopt SomeClassOnlyProtocol.

Use a class-only protocol when the behavior defined by that protocol's requirements assumes or requires that a conforming type has reference semantics rather than value semantics.

163. Protocols - **Protocol Composition - <>**

It can be useful to require a type to conform to multiple protocols at once. You can combine multiple protocols into a single requirement with a protocol composition. Protocol compositions have the form protocol<SomeProtocol, AnotherProtocol>. You can list as many protocols within the pair of angle brackets (**<>**) as you need, separated by commas.

Here's an example that combines two protocols called Named and Aged into a single protocol composition requirement on a function parameter:

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(celebrator: protocol<Named, Aged>) {
```

```
    print("Happy birthday \(celebrator.name) - you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(birthdayPerson)
```

164. Protocols - **Protocol Conformance**

Checking for Protocol Conformance

You can use the **is, as?** and **as!** operators to check for protocol conformance, and to cast to a specific protocol.

```
protocol HasArea {
    var area: Double { get }
}

class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}

class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}

let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]

for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}
```

165. Protocols - **Optional Protocol Requirements**

You can define optional requirements for protocols. These requirements do not have to be implemented by types that conform to the protocol.

Optional requirements are prefixed by the **optional** modifier as part of the protocol's definition. When you use a method or property in an optional requirement, its type automatically becomes an optional. For example, a method of type (Int) -> String while accessing the method becomes ((Int) -> String)?. The entire function type is wrapped in the optional, not the method's return value.

An optional protocol requirement can be called with optional chaining, to account for the possibility that the requirement was not implemented by a type that conforms to the protocol. You check for an implementation of an optional method by writing a question mark after the name of the method when it is called, such as someOptionalMethod?(someArgument).

Optional protocol requirements can only be specified if your protocol is marked with the @objc attribute.

This attribute indicates that the protocol should be exposed to Objective-C code.

@objc protocols can be adopted only by classes that inherit from Objective-C classes or other @objc classes. They can't be adopted by structures or enumerations.

@objc protocol CounterDataSource {

```
    optional func incrementForCount(count: Int) -> Int
    optional var fixedIncrement: Int { get }
}

class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.incrementForCount?(count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

166. Protocols - **Extensions**

Protocols can be extended to provide method and property implementations to conforming types. This allows you to define behavior on protocols themselves, rather than in each type's individual conformance or in a global function.

```
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
```

By creating an extension on the protocol, all conforming types automatically gain this method implementation without any additional modification.

167. Protocols - **Providing Extensions**

You can use protocol extensions to provide a default implementation to any method or property requirement of that protocol.
If a conforming type provides its own implementation of a required method or property, that implementation will be used instead of the one provided by the extension.

Protocol requirements with default implementations provided by extensions are distinct from optional protocol requirements.
Although conforming types don't have to provide their own implementation of either, requirements with default implementations can be called without optional chaining.

For example, the PrettyTextRepresentable protocol, which inherits the TextRepresentable protocol can provide a default implementation of its required prettyTextualDescription property to simply return the result of accessing the textualDescription property:

```
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

168. Protocols - **Adding Constraints**

When you define a protocol extension, you can specify constraints that conforming types must satisfy before the methods and properties of the extension are available. You write these constraints after the name of the protocol you're extending using a **where** clause, as described in Where Clauses.

For instance, you can define an extension to the CollectionType protocol that applies to any collection whose elements conform to the TextRepresentable protocol from the example above.

```
extension CollectionType where Generator.Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joinWithSeparator(", ") + "]"
    }
}
```

The textualDescription property returns the textual description of the entire collection by concatenating the textual representation of each element in the collection into a comma-separated list, enclosed in brackets.

Consider the Hamster structure from before, which conforms to the TextRepresentable protocol, and an array of Hamster values:

```
let murrayTheHamster = Hamster(name: "Murray")
let morganTheHamster = Hamster(name: "Morgan")
let mauriceTheHamster = Hamster(name: "Maurice")
let hamsters = [murrayTheHamster, morganTheHamster, mauriceTheHamster]
```

Because Array conforms to CollectionType and the array's elements conform to the TextRepresentable protocol, the array can use the textualDescription property to get a textual representation of its contents:

```
print(hamsters.textualDescription)
// Prints "[A hamster named Murray, A hamster named Morgan, A hamster named Maurice]
```

If a conforming type satisfies the requirements for multiple constrained extensions that provide implementations for the same method or property, Swift will use the implementation corresponding to the most specialized constraints.

169. **Generics**

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code. In fact, you've been using generics throughout the Language Guide, even if you didn't realize it.
For example, Swift's Array and Dictionary types are both generic collections. You can create an array that holds Int values, or an array that holds String values, or indeed an array for any other type that can be created in Swift. Similarly, you can create a dictionary to store values of any specified type, and there are no limitations on what that type can be.

```
func swapTwoValues<T>(inout a: T, inout _ b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

The generic version of the function uses a placeholder type name (called T, in this case) instead of an actual type name (such as Int, String, or Double). The placeholder type name doesn't say anything about what T must be, but it does say that both a and b must be of the same type T, whatever T represents. The actual type to use in place of T will be determined each time the swapTwoValues(_:_:) function is called.

The other difference is that the generic function's name (swapTwoValues(_:_:)) is followed by the placeholder type name (T) inside angle brackets (<T>). The brackets tell Swift that T is a placeholder type name within the swapTwoValues(_:_:) function definition. Because T is a placeholder, Swift does not look for an actual type called T.

The placeholder type **T** is an example of a **type parameter**.

type parameters have descriptive names, such as **Key and Value in Dictionary<Key, Value>** and **Element in Array<Element>**, which tells the reader about the relationship between the type parameter and the generic type or function it's used in. However, when there isn't a meaningful relationship between them, it's traditional to name them using single letters such as **T**, **U**, and **V**

170. Generics - **Generics Type**

In addition to generic functions, Swift enables you to define your own generic types. These are custom classes, structures, and enumerations that can work with any type, in a similar way to Array and Dictionary.

Below is an example on a generic collection type called Stack. The stack allows new items to be appended only to the end of the collection (push). Similarly, a stack allows items to be removed only from the end of the collection (pop).

```
struct Stack<Element> {
    var items = [Element]()
    mutating func push(item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

Element defines a placeholder name for "some type Element" to be provided later on. This future type can be referred to as "Element" anywhere within the structure's definition.

You create a new Stack instance by writing the type to be stored in the stack within angle brackets.:

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
```

171. Generics - **Extending a Generic Type**

When you extend a generic type, you do not provide a type parameter list as part of the extension's definition. Instead, the type parameter list from the original type definition is available within the body of the extension, and the original type parameter names are used to refer to the type parameters from the original definition.

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}

if let topItem = stackOfStrings.topItem {
    print("The top item on the stack is \(topItem).")
}
```

172. Generics - **Type Constraints**

It is sometimes useful to enforce certain type constraints on the types that can be used with generic functions and generic types. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.

For example, Swift's Dictionary type places a limitation on the types that can be used as keys for a dictionary must be hashable. All of Swift's basic types (such as String, Int, Double, and Bool) are hashable by default.

You can define your own type constraints when creating custom generic types, and these constraints provide much of the power of generic programming.

You write type constraints by placing a single class or protocol constraint after a type parameter's name, separated by a colon, as part of the type parameter list. The basic syntax for type constraints on a generic function is shown below:

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

Here's a generic function called findIndex, which is given a value to find and an array of values within which to find it. The findIndex(_:_:) function returns an optional Int value, which will be the index of the first matching element in the array if it is found, or nil if the element cannot be found

```
func findIndex<T: Equatable>(array: [T], _ valueToFind: T) -> Int? {
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The single type parameter for findIndex is written as T: Equatable, which means "any type T that conforms to the Equatable protocol".

173. Generics - **Associated Types**

When defining a protocol, it is sometimes useful to declare one or more associated types as part of the protocol's definition. An associated type gives a placeholder name to a type that is used as part of the protocol. The actual type to use for that associated type is not specified until the protocol is adopted. Associated types are specified with the **associatedtype** keyword.

```
protocol Container {
    associatedtype ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

The protocol only specifies the three bits of functionality that any type must provide in order to be considered a Container. A conforming type can provide additional functionality, as long as it satisfies these three requirements.

Any type that conforms to the Container protocol must be able to specify the type of values it stores. Specifically, it must ensure that only items of the right type are added to the container, and it must be clear about the type of the items returned by its subscript.

```
struct Stack<Element>: Container {
    // original Stack<Element> implementation
    var items = [Element]()
    mutating func push(item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}
```

Stack specifies that for this implementation of Container, the appropriate ItemType to use is a type of Element. The definition of typealias ItemType = Element turns the abstract type of ItemType into a concrete type of Element for this implementation of the Container protocol.
Thanks to Swift's type inference, you don't actually need to declare a concrete ItemType of Element as part of the definition of Stack. Because Stack conforms to all of the requirements of the Container protocol, Swift can infer the appropriate ItemType to use, simply by looking at the type of the append(_:) method's item parameter and the return type of the subscript.

174. Generics - **Extending an Existing Type to Specify an Associated Type**

You can extend an existing type to add conformance to a protocol. This includes a protocol with an associated type.

Swift's Array type already provides an append(_:) method, a count property, and a subscript with an Int index to retrieve its elements.
These three capabilities match the requirements of the Container protocol. This means that you can extend Array to conform to the Container protocol simply by declaring that Array adopts the protocol.:

```
extension Array: Container {}
```

Array's existing append(_:) method and subscript enable Swift to infer the appropriate type to use for ItemType, just as for the generic Stack type above. After defining this extension, you can use any Array as a Container.

175. Generics - **Where clause**

Type constraints, enable you to define requirements on the type parameters associated with a generic function or type.

It can also be useful to define requirements for associated types. You do this by defining **where** clauses as part of a type parameter list. A **where** clause enables you to require that an associated type must conform to a certain protocol, or that certain type parameters and associated types must be the same.
You write a **where** clause by placing the where keyword immediately after the list of type parameters, followed by constraints for associated types or equality relationships between types and associated types.

The example below defines a generic function called allItemsMatch, which checks to see if two Container instances contain the same items in the same order. The function returns a Boolean value of true if all items match and a value of false if they do not.

```
func allItemsMatch <C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, _ anotherContainer: C2) -> Bool {

    // check that both containers contain the same number of items
    if someContainer.count != anotherContainer.count {
        return false
    }
    // check each pair of items to see if they are equivalent
    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }
    // all items match, so return true
    return true
}

var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    print("All items match.")
} else {
    print("Not all items match.")
}
```

arrayOfStrings conforms to protocol because we extended the Array class to conform to Container protocol. (refer above)

---

176. **Access Control**

Access control restricts access to parts of your code from code in other source files and modules. This feature enables you to hide the implementation details of your code, and to specify a preferred interface through which that code can be accessed and used.

You can assign specific access levels to individual types (classes, structures, and enumerations), as well as to properties, methods, initializers, and subscripts belonging to those types. Protocols can be restricted to a certain context, as can global constants, variables, and functions.

A module is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift's import keyword.

Each build target (such as an app bundle or framework) in Xcode is treated as a separate module in Swift. If you group together aspects of your app's code as a stand-alone framework—perhaps to encapsulate and reuse that code across multiple applications—then everything you define within that framework will be part of a separate module when it is imported and used within an app, or when it is used within another framework.

A source file is a single Swift source code file within a module (in effect, a single file within an app or framework). Although it is common to define individual types in separate source files, a single source file can contain definitions for multiple types, functions, and so on.

177. Access Control - **Access levels - Public | Internal | Private**

Swift provides three different access levels for entities within your code. These access levels are relative to the source file in which an entity is defined, and also relative to the module that source file belongs to.

- **Public** access enables entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use public access when specifying the public interface to a framework.
- **Internal** access enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when defining an app's or a framework's internal structure.
- **Private** access restricts the use of an entity to its own defining source file. Use private access to hide the implementation details of a specific piece of functionality. This means that a type can access any private entities that are defined in the same source file as itself, but an extension cannot access that type's private members if it's defined in a separate source file.

```
public class SomePublicClass {}
~~internal~~ class SomeInternalClass {}                  // implicitly internal
private class SomePrivateClass {}

public var somePublicVariable = 0
~~internal~~ let someInternalConstant = 0                // implicitly internal
private func somePrivateFunction() {}

class SomeInternalClass {}
let someInternalConstant = 0
```

178. Access Control - **Principles of access levels**

No entity can be defined in terms of another entity that has a lower (more restrictive) access level.

- A public variable cannot be defined as having an internal or private type, because the type might not be available everywhere that the public variable is used.
- A function cannot have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are not available to the surrounding code.

179. Access Control - **Default Access Levels**

All entities in your code (with a few specific exceptions, as described later in this chapter) have a default access level of internal if you do not specify an explicit access level yourself. As a result, in many cases you do not need to specify an explicit access level in your code.

180. Access Control - **Access levels for Frameworks**

When you develop a framework, mark the public-facing interface to that framework as public so that it can be viewed and accessed by other modules, such as an app that imports the framework.

Any internal implementation details of your framework can still use the default access level of internal, or can be marked as private if you want to hide them from other parts of the framework's internal code.
You need to mark an entity as public only if you want it to become part of your framework's API.

181. Access Control - **Access levels for Unit Tests Targets - @testable**

When you write an app with a unit test target, the code in your app needs to be made available to that module in order to be tested.
By default, only entities marked as public are accessible to other modules. However, a unit test target can access any internal entity, if you mark the import declaration for a product module with the **@testable** attribute and compile that product module with testing enabled.

182. Access Control - **Custom Types**

If you want to specify an explicit access level for a custom type, do so at the point that you define the type. The new type can then be used wherever its access level permits.
For example, if you define a private class, that class can only be used as the type of a property, or as a function parameter or return type, in the source file in which the private class is defined.

The access control level of a type also affects the default access level of that type's members (its properties, methods, initializers, and subscripts).
If you define a type's access level as private, the default access level of its members will also be private.
If you define a type's access level as internal or public, the default access level of the type's members will be internal. Meaning that a public type defaults to having internal members, not public members. If you want a type member to be public, you must explicitly mark it as such.

```
public class SomePublicClass {          // explicitly public class
    public var somePublicProperty = 0    // explicitly public class member
    var someInternalProperty = 0         // implicitly internal class member
    private func somePrivateMethod() {}  // explicitly private class member
}

class SomeInternalClass {               // implicitly internal class
    var someInternalProperty = 0         // implicitly internal class member
    private func somePrivateMethod() {}  // explicitly private class member
}

private class SomePrivateClass {        // explicitly private class
    var somePrivateProperty = 0          // implicitly private class member
    func somePrivateMethod() {}          // implicitly private class member
}
```

183. Access Control - **Tuple Types**

The access level for a tuple type is the most restrictive access level of all types used in that tuple. For example, if you compose a tuple from two different types, one with internal access and one with private access, the access level for that compound tuple type will be private.

The access level for a function type is calculated as the most restrictive access level of the function's parameter types and return type. You must specify the access level explicitly as part of the function's definition if the function's calculated access level does not match the contextual default.

The example below defines a global function called someFunction:

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

The function's return type is a tuple type composed from two of the custom classes "SomeInternalClass" and "SomePrivateClass" are defined in the above points.
One of these classes was defined as "internal", and the other was defined as "private". Therefore, the overall access level of the compound tuple type is "**private**" (the minimum access level of the tuple's constituent types).

184. Access Control - **Enumeration Types**

The individual cases of an enumeration automatically receive the same access level as the enumeration they belong to. You cannot specify a different access level for individual enumeration cases.

In the example below, the CompassPoint enumeration has an explicit access level of “public”. The enumeration cases North, South, East, and West therefore also have an **access level of “public”**:

```
public enum CompassPoint {
   case North
   case South
   case East
   case West
}
```

185. Access Control - **Raw Values and Associated Values**

The types used for any raw values or associated values in an enumeration definition must have an access level at least as high as the enumeration’s access level. You cannot use a private type as the raw-value type of an enumeration with an internal access level, for example.

186. Access Control - **Nested Types**

Nested types defined within a private type have an automatic access level of private. Nested types defined within a public type or an internal type have an automatic access level of internal. If you want a nested type within a public type to be publicly available, you must explicitly declare the nested type as public.

187. Access Control - **Subclassing**

A subclass cannot have a higher access level than its superclass—for example, you cannot write a public subclass of an internal superclass.

In addition, you can override any class member (method, property, initializer, or subscript) that is visible in a certain access context. **An override can make an inherited class member more accessible than its superclass version**.

In the example below, class A is a public class with a private method called someMethod(). Class B is a subclass of A, with a reduced access level of “internal”. Nonetheless, class B provides an override of someMethod() with an access level of “internal”, which is higher than the original implementation of someMethod():

```
public class A {
   private func someMethod() {}
}

internal class B: A {
   override internal func someMethod() {}
}
```

It is even valid for a subclass member to call a superclass member that has lower access permissions than the subclass member, as long as the call to the superclass’s member takes place within an allowed access level context (that is, within the same source file as the superclass for a private member call, or within the same module as the superclass for an internal member call):

```
public class A {
   private func someMethod() {}
}

internal class B: A {
   override internal func someMethod() {
      super.someMethod()
   }
}
```

Because superclass A and subclass B are defined in the same source file, it is valid for the B implementation of someMethod() to call super.someMethod().

188. Access Control - **Constants | Variables | Properties | Subscripts**

A constant, variable, or property cannot be more public than its type. It is not valid to write a public property with a private type, for example. Similarly, a subscript cannot be more public than either its index type or return type.

If a constant, variable, property, or subscript makes use of a private type, the constant, variable, property, or subscript must also be marked as private:
private var privateInstance = SomePrivateClass()

189. Access Control - **Getters and Setters**

Getters and setters for constants, variables, properties, and subscripts automatically receive the same access level as the constant, variable, property, or subscript they belong to.

You can give a setter a lower access level than its corresponding getter, to restrict the read-write scope of that variable, property, or subscript. You assign a lower access level by writing private(set) or internal(set) before the var or subscript introducer.

This rule applies to stored properties as well as computed properties. Even though you do not write an explicit getter and setter for a stored property, Swift still synthesizes an implicit getter and setter for you to provide access to the stored property's backing storage.

The example below defines a structure called TrackedString, which keeps track of the number of times a string property is modified. The TrackedString structure defines a stored string property called value, with an initial value of "" (an empty string). The structure also defines a stored integer property called numberOfEdits, which is used to track the number of times that value is modified.

```
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
}
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment numberOfEdits."
print("The number of edits is \(stringToEdit.numberOfEdits)")
```

The TrackedString structure and the value property do not provide an explicit access level modifier, and so they both receive the default access level of internal.
The property's getter still has the default access level of internal, but its setter is now private to the source file in which TrackedString is defined. Meaning that the setter will not be accessible from another file.. Example "stringToEdit.numberOfEdits = 3" will not be possible from another file other than the source file where the TrackedString struct is written.

Note that you can assign an explicit access level for both a getter and a setter if required.
The example below shows a version of the TrackedString structure in which the structure is defined with an explicit access level of public. The structure's members (including the numberOfEdits property) therefore have an internal access level by default. You can make the structure's numberOfEdits property getter public, and its property setter private, by combining the public and private(set) access level modifiers:

```
public struct TrackedString {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}
```

190. Access Control - **Initializers**

Custom initializers can be assigned an access level less than or equal to the type that they initialize. The only exception is for required initializers. A required initializer must have the same access level as the class it belongs to.

As with function and method parameters, the types of an initializer's parameters cannot be more private than the initializer's own access level.

191. Access Control - **Default Initializers**

Swift automatically provides a default initializer without any arguments for any structure or base class that provides default values for all of its properties and does not provide at least one initializer itself.

A default initializer has the same access level as the type it initializes, unless that type is defined as public. For a type that is defined as public, the default initializer is considered internal. If you want a public type to be initializable with a no-argument initializer when used in another module, you must explicitly provide a public no-argument initializer yourself as part of the type's definition.

192. Access Control - **Default Memberwise Initializers for Structure Types**

The default memberwise initializer for a structure type is considered private if any of the structure's stored properties are private. Otherwise, the initializer has an access level of internal.

As with the default initializer above, if you want a public structure type to be initializable with a memberwise initializer when used in another module, you must provide a public memberwise initializer yourself as part of the type's definition.

193. Access Control - **Protocols**

If you want to assign an explicit access level to a protocol type, do so at the point that you define the protocol. This enables you to create protocols that can only be adopted within a certain access context.

The access level of each requirement within a protocol definition is automatically set to the same access level as the protocol. You cannot set a protocol requirement to a different access level than the protocol it supports. This ensures that all of the protocol's requirements will be visible on any type that adopts the protocol.

194. Access Control - **Protocol Inheritance**

If you define a new protocol that inherits from an existing protocol, the new protocol can have at most the same access level as the protocol it inherits from. You cannot write a public protocol that inherits from an internal protocol, refer below example

```
internal protocol protocol1 {
    var getCount:Int {set get}
}

public protocol protocol2: protocol1 {          // ERROR - COMPILER ERROR at public definition

}
```

195. Access Control - **Protocol Conformance**

A type can conform to a protocol with a lower access level than the type itself. For example, you can define a public type that can be used in other modules, but whose conformance to an internal protocol can only be used within the internal protocol's defining module.

The context in which a type conforms to a particular protocol is the minimum of the type's access level and the protocol's access level. If a type is public, but a protocol it conforms to is internal, the type's conformance to that protocol is also internal.

When you write or extend a type to conform to a protocol, you must ensure that the type's implementation of each protocol requirement has at least the same access level as the type's conformance to that protocol. For example, if a public type conforms to an internal protocol, the type's implementation of each protocol requirement must be at least "internal".

196. Access Control - **Extensions**

You can extend a class, structure, or enumeration in any access context in which the class, structure, or enumeration is available. Any type members added in an extension have the same default access level as type members declared in the original type being extended. If you extend a public or internal type, any new type members you add will have a default access level of internal. If you extend a private type, any new type members you add will have a default access level of private.

Alternatively, you can mark an extension with an explicit access level modifier (for example, private extension) to set a new default access level for all members defined within the extension. This new default can still be overridden within the extension for individual type members.

197. Access Control - **Adding Protocol Conformance with an Extension**

You cannot provide an explicit access level modifier for an extension if you are using that extension to add protocol conformance. Instead, the protocol's own access level is used to provide the default access level for each protocol requirement implementation within the extension.

198. Access Control - **Generics**

The access level for a generic type or generic function is the minimum of the access level of the generic type or function itself and the access level of any type constraints on its type parameters.

199. Access Control - **Type Alias**

Any type aliases you define are treated as distinct types for the purposes of access control. A type alias can have an access level less than or equal to the access level of the type it aliases. For example, a private type alias can alias a private, internal, or public type, but a public type alias cannot alias an internal or private type.

```
public typealias StringAlias = String
public var string:StringAlias OR internal var string:StringAlias OR private var string:StringAlias

internal typealias StringAlias = String
public var string:StringAlias   // ERROR - As an equal or lower access control than internal has to be used
```

This rule also applies to type aliases for associated types used to satisfy protocol conformances.

200. **Advanced Operators**

Unlike arithmetic operators in C, arithmetic operators in Swift do not overflow by default. Overflow behavior is trapped and reported as an error. To opt in to overflow behavior, use Swift's second set of arithmetic operators that overflow by default, such as the overflow addition operator (&+). All of these overflow operators begin with an ampersand (&).

201. Advanced Operators - **Bitwise operators**

Bitwise operators enable you to manipulate the individual raw data bits within a data structure.

202. Advanced Operators - **Bitwise NOT Operators - ~**

The bitwise NOT operator (~) inverts all bits in a number. The bitwise NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space:

```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

UInt8 integers have eight bits and can store any value between 0 and 255.

203. Advanced Operators - **Bitwise AND Operators - &**

The bitwise AND operator (**&**) combines the bits of two numbers. It returns a new number whose bits are set to 1 only if the bits were equal to 1 in both input numbers:

In the example below, the values of firstSixBits and lastSixBits. The bitwise AND operator combines them to make the number 00111100:

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // equals 00111100 = 60 (decimal value)
```

204. Advanced Operators - **Bitwise OR Operators - |**

The bitwise OR operator (**|**) compares the bits of two numbers. The operator returns a new number whose bits are set to 1 if the bits are equal to 1 in either input number:

In the example below, the values of someBits and moreBits. The bitwise OR operator combines them to make the number 11111110:

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // equals 11111110 = 254 (decimal value)
```

205. Advanced Operators - **Bitwise OR Operators - ^**

The bitwise XOR operator, or "exclusive OR operator" (**^**), compares the bits of two numbers. The operator returns a new number whose bits are set to 1 where the input bits are different and are set to 0 where the input bits are the same:

In the example below, the values of firstBits and otherBits each have a bit set to 1 in a location that the other does not. The bitwise XOR operator sets both of these bits to 1 in its output value. All of the other bits in firstBits and otherBits match and are set to 0 in the output value:

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // equals 00010001
```

206. Advanced Operators - **Bitwise LEFT and RIGHT SHIFT Operators - <<(left) >> (right)**

The bitwise left shift operator (<<) and bitwise right shift operator (>>) move all bits in a number to the left or the right by a certain number of places, according to the rules defined below.

Bitwise left and right shifts have the effect of multiplying or dividing an integer number by a factor of two. **Shifting an integer's bits to the left by one position doubles its value, whereas shifting it to the right by one position halves its value**.

207. Advanced Operators - **Bit-shifting behavior for unsigned integers**

- Existing bits are moved to the left or right by the requested number of places.
- Any bits that are moved beyond the bounds of the integer's storage are discarded.

- Zeros are inserted in the spaces left behind after the original bits are moved to the left or right.

This approach is known as a logical shift.

Lets take an example of 11111111 << 1 (which is 11111111 shifted to the left by 1 place), and 11111111 >> 1 (which is 11111111 shifted to the right by 1 place).
11111111 << 1 will result in 11111110
11111111 >> 1 will result in 01111111

```
let shiftBits: UInt8 = 4          // 00000100 in binary
shiftBits << 1                    // 00001000
shiftBits << 2                    // 00010000
shiftBits << 5                    // 10000000
shiftBits << 6                    // 00000000
shiftBits >> 2                    // 00000001

let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16        // redComponent is 0xCC, or 204
let greenComponent = (pink & 0x00FF00) >> 8       // greenComponent is 0x66, or 102
let blueComponent = pink & 0x0000FF               // blueComponent is 0x99, or 153
```

208. Advanced Operators - **Bit-shifting behavior for signed integers**

Signed integers use their first bit (known as the sign bit) to indicate whether the integer is positive or negative. A sign bit of 0 means positive, and a sign bit of 1 means negative. The remaining bits (known as the value bits) store the actual value.
Positive numbers are stored in exactly the same way as for unsigned integers, counting upwards from 0.

Here's how the bits inside an Int8 look for the number 4:
0(sign bit) 0 0 0 0 1 0 0 = 4 (0000100 are value bits)
The sign bit is 0 (meaning "positive"), and the seven value bits are just the number 4, written in binary notation.

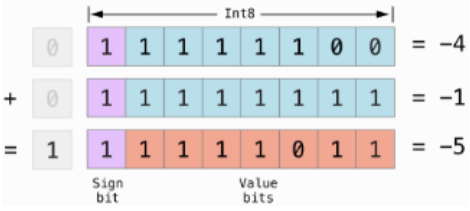
Negative numbers, however, are stored differently. They are stored by subtracting their absolute value from 2 to the power of n, where n is the number of value bits. An eight-bit number has seven value bits, so this means 2 to the power of 7, or 128.

Here's how the bits inside an Int8 look for the number -4:
1(sign bit) 1 1 1 1 1 0 0 = -4 (1111100 are value bits)

This time, the sign bit is 1 (meaning "negative"), and the seven value bits have a binary value of 124 (which is 128 - 4):
1111100 = 124

This encoding for negative numbers is known as a two's complement representation.

First, you can add -1 to -4, simply by performing a standard binary addition of all eight bits (including the sign bit), and discarding anything that doesn't fit in the eight bits once you're done:



Second, the two's complement representation also lets you shift the bits of negative numbers to the left and right like positive numbers, and still end up doubling them for every shift you make to the left, or halving them for every shift you make to the right. To achieve this, an extra rule is used when signed integers are shifted to the right:

When you shift signed integers to the right, apply the same rules as for unsigned integers, but fill any empty bits on the left with the sign bit, rather than with a zero. For example
1111111 becomes 11111111
0111111 becomes 00111111

This action ensures that signed integers have the same sign after they are shifted to the right, and is known as an arithmetic shift.

209. Advanced Operators - **Overflow Operators - &* | &+ | &-**

If you try to insert a number into an integer constant or variable that cannot hold that value, by default Swift reports an error rather than allowing an invalid value to be created. This behavior gives extra safety when you work with numbers that are too large or too small.

For example, the Int16 integer type can hold any signed integer number between -32768 and 32767. Trying to set an Int16 constant or variable to a number outside of this range causes an error:

```
var potentialOverflow = Int16.max    // equals 32767, which is the max value an Int16 can hold
potentialOverflow += 1               // ERROR - this causes an error
```

However, when you specifically want an overflow condition to truncate the number of available bits, you can opt in to this behavior rather than triggering an error. Swift provides three arithmetic overflow operators that opt in to the overflow behavior for integer calculations. These operators all begin with an ampersand (**&**):

Overflow addition (&+)
Overflow subtraction (&-)
Overflow multiplication (&*)

210. Advanced Operators - **Value Overflow**

Numbers can overflow in both the positive and negative direction.

Here's an example of what happens when an unsigned integer is allowed to overflow in the positive direction, using the overflow addition operator (&+):

```
var unsignedOverflow = UInt8.max                // equals 255, which is the max value a UInt8 can hold
unsignedOverflow = unsignedOverflow &+ 1        // unsignedOverflow is now equal to 0
```

The variable unsignedOverflow is initialized with the maximum value a UInt8 can hold (255, or 11111111 in binary). It is then incremented by 1 using the overflow addition operator (&+). This pushes its binary representation just over the size that a UInt8 can hold, causing it to overflow beyond its bounds. The value that remains within the bounds of the UInt8 after the overflow addition is 00000000, or zero.

Something similar happens when an unsigned integer is allowed to overflow in the negative direction. Here's an example using the overflow subtraction operator (&-):

```
var unsignedOverflow = UInt8.min                // equals 0, which is the min value a UInt8 can hold
unsignedOverflow = unsignedOverflow &- 1        // unsignedOverflow is now equal to 255
```

The minimum value that a UInt8 can hold is zero, or 00000000 in binary. If you subtract 1 from 00000000 using the overflow subtraction operator (&-), the number will overflow and wrap around to 11111111, or 255 in decimal.

Overflow also occurs for signed integers. All addition and subtraction for signed integers is performed in bitwise fashion, with the sign bit included as part of the numbers being added or subtracted

```
var signedOverflow = Int8.min                   // equals -128, which is the min value an Int8 can hold
signedOverflow = signedOverflow &- 1            // signedOverflow is now equal to 127
```

The minimum value that an Int8 can hold is -128, or 10000000 in binary. Subtracting 1 from this binary number with the overflow operator gives a binary value of 01111111, which toggles the sign bit and gives positive 127, the maximum positive value that an Int8 can hold.

For both signed and unsigned integers, overflow in the positive direction wraps around from the maximum valid integer value back to the minimum, and overflow in the negative direction wraps around from the minimum value to the maximum.

211. Advanced Operators - **Precedence and Associativity**

Operator precedence gives some operators higher priority than others; these operators are applied first.

Operator associativity defines how operators of the same precedence are grouped together—either grouped from the left, or grouped from the right. Think of it as meaning “they associate with the expression to their left,” or “they associate with the expression to their right.”

It is important to consider each operator’s precedence and associativity when working out the order in which a compound expression will be calculated.

For example, operator precedence explains why the following expression equals 17.

```
2 + 3 % 4 * 5          // this equals 17
```

In Swift, as in C, the remainder operator (%) and the multiplication operator (*) have a higher precedence than the addition operator (+). As a result, they are both evaluated before the addition is considered.

However, remainder and multiplication have the same precedence as each other. To work out the exact evaluation order to use, you also need to consider their associativity.
Remainder and multiplication both associate with the expression to their left. Think of this as adding implicit parentheses around these parts of the expression, starting from their left:
2 + ((3 % 4) * 5)
(3 % 4) is 3, so this is equivalent to:
2 + (3 * 5)
(3 * 5) is 15, so this is equivalent to:
2 + 15
This calculation yields the final answer of 17.

212. Advanced Operators - **Operator Functions - Overloading**

Classes and structures can provide their own implementations of existing operators. This is known as **overloading** the existing operators.

The example below shows how to implement the arithmetic addition operator (+) for a custom structure. The arithmetic addition operator is a binary operator because it operates on two targets and is said to be infix because it appears in between those two targets.

The example defines a Vector2D structure for a two-dimensional position vector (x, y), followed by a definition of an operator function to add together instances of the Vector2D structure. The operator function is defined as a global function with a function name that matches the operator to be overloaded (+)

```
struct Vector2D {
    var x = 0.0, y = 0.0
}

func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
```

The function is defined globally, rather than as a method on the Vector2D structure, so that it can be used as an infix operator between existing Vector2D instances:

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector          // Vector2D instance with values of (5.0, 5.0)
```

The example shown above demonstrates a custom implementation of a binary infix operator.

213. Advanced Operators - **Prefix and Postfix operators**

Classes and structures can also provide implementations of the standard unary operators. Unary operators operate on a single target. They are prefix if they precede their target (such as -a) and postfix operators if they follow their target (such as b!).

You implement a prefix or postfix unary operator by writing the prefix or postfix modifier before the func keyword when declaring the operator function:

```
prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -vector.y)
}
```

The example above implements the unary minus operator (-a) for Vector2D instances. The unary minus operator is a prefix operator, and so this function has to be qualified with the prefix modifier.

For simple numeric values, the unary minus operator converts positive numbers into their negative equivalent and vice versa:

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive          // negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative    // alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

214. Advanced Operators - **Compound Assignment Operators**

Compound assignment operators combine assignment (=) with another operation. For example, the addition assignment operator (+=) combines addition and assignment into a single operation. You mark a compound assignment operator's left input parameter as inout, because the parameter's value will be modified directly from within the operator function.

The example below implements an addition assignment operator function for Vector2D instances:

```
func += (inout left: Vector2D, right: Vector2D) {
    left = left + right
}
```

The addition assignment operator function takes advantage of the existing addition operator function, and uses it to set the left value to be the left value plus the right value:

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd                    // original now has values of (4.0, 6.0)
```

It is not possible to overload the default assignment operator (=).
Only the compound assignment operators can be overloaded.
Similarly, the ternary conditional operator (a ? b : c) cannot be overloaded.

215. Advanced Operators - **Equivalence Operators - == | !=**

Custom classes and structures do not receive a default implementation of the equivalence operators, known as the "equal to" operator (==) and "not equal to" operator (!=).
It is not possible for Swift to guess what would qualify as "equal" for your own custom types, because the meaning of "equal" depends on the roles that those types play in your code.

To use the equivalence operators to check for equivalence of your own custom type:

```
func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}

func != (left: Vector2D, right: Vector2D) -> Bool {
    return !(left == right)
}
```

You can now use these operators to check whether two Vector2D instances are equivalent:

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
```

216. Advanced Operators - **Custom Operators**

You can declare and implement your own custom operators in addition to the standard operators provided by Swift.

New operators are declared at a global level using the operator keyword, and are marked with the prefix, infix or postfix modifiers:

prefix **operator** +++ {}

The example above defines a new prefix operator called +++.

For the purposes of this example, +++ is treated as a new “prefix doubling incrementer” operator. It doubles the x and y values of a Vector2D instance, by adding the vector to itself with the addition assignment operator defined earlier:

```
prefix func +++ (inout vector: Vector2D) -> Vector2D {
   vector += vector
   return vector
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled          // toBeDoubled now has values of (2.0, 8.0)
                                            // afterDoubling also has values of (2.0, 8.0)
```

217. Advanced Operators - **Precedence and Associativity for Custom Infix Operators**

Custom infix operators can also specify a precedence and an associativity.

The possible values for associativity are left, right, and none. Left-associative operators associate to the left if written next to other left-associative operators of the same precedence. Similarly, right-associative operators associate to the right if written next to other right-associative operators of the same precedence. Non-associative operators cannot be written next to other operators with the same precedence.

The associativity value defaults to none if it is not specified. The precedence value defaults to 100 if it is not specified.

The following example defines a new custom infix operator called +-, with left associativity and a precedence of 140:

```
infix operator +- { associativity left precedence 140 }
func +- (left: Vector2D, right: Vector2D) -> Vector2D {
   return Vector2D(x: left.x + right.x, y: left.y - right.y)
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector // Vector2D instance with values of (4.0, -2.0)
```

This operator adds together the x values of two vectors, and subtracts the y value of the second vector from the first. Because it is in essence an “additive” operator, it has been given the same associativity and precedence values (left and 140) as default additive infix operators such as + and -.

You do not specify a precedence when defining a prefix or postfix operator. However, if you apply both a prefix and a postfix operator to the same operand, the postfix operator is applied first.

Copyright and Notice

Being a Swift application developer, I know the relief in getting free knowledge/content over the internet and use them while developing awesomeness mobile applications. However, this goes without saying that every piece of information available on the internet has some license/copyright/policy tagged along with it.

I do not own any rights to the contents in this document. All the content in this document is taken "AS-IS" from the Swift 2.2 Apple iBook. All rights/policies of the contents are owned by Apple. I would like to emphasis the point that Apple has mentioned in their iBook.
"This document is intended to assist application developers to develop applications only for Apple-branded products."
Excerpt From: Apple Inc. "The Swift Programming Language (Swift 2.2)." iBooks. https://itunes.apple.com/us/book/swift-programming-language/id881256329?mt=11"

I would like to hold the rights only for structuring and creating this document. I have solely made this document as a study material or reference-only, non-commercial use.