Learn Swift

Learn Intermediate Swift

Learn Swift Cheatsheet

https://www.codecademy.com/learn/learn-swift/modules/learn-swift-hello-world/cheatsheet

Hello World

Variables

Constants Types String Interpolation

Conditionals & Logic

If statement Ternary conditional operator Switch statement Switch statement compound cases Switch statement where clause

Loops

ranges stride function for in loop continue break using underscore while loop

Arrays and Sets

array

initialize with array literal

index

.count property

.append method

insert and remove

iterating over an array

swift sets

empty sets

populated sets

insert

remove and removeAll

contains

iterating over a set

Learn Intermediate Swift Cheatsheet

Hello World - Learn Swift

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

```
// This line denotes a comment in Swift.
```

```
/*
This is all commented out.
None of it is going to run!
*/
```

Variables - Learn Swift

```
var score = 0
```

Constants

```
let pi = 3.14
```

Types

```
var age: Int = 28

var price: Double = 8.99

var message: String = "good nite"

var lateToWork: Bool = true
```

String Interpolation

```
var apples = 6

print("I have \(apples) apples!")

// Prints: I have 6 apples!
```

Conditionals & Logic - Learn Swift

if Statement

```
var weather = "rainy"

if weather == "sunny" {
  print("Grab some sunscreen")
} else if weather == "rainy" {
  print("Grab an umbrella")
} else if weather == "snowing" {
  print("Wear your snow boots")
} else {
  print("Invalid weather")
}

// Prints: Grab an umbrella
```

Ternary Conditional Operator

```
var driverLicense = true

driverLicense ? print("Driver's Seat") : print("Passenger's Seat")

// Prints: Driver's Seat
```

switch Statement

```swift
var secondaryColor = "green"

switch secondaryColor {
  case "orange":
    print("Mix of red and yellow")
  case "green":
    print("Mix of blue and yellow")
  case "purple":
    print("Mix of red and blue")
  default:
    print("This might not be a secondary color.")
}

// Prints: Mix of blue and yellow
```

switch Statement: Interval Matching

```swift
let year = 1905
var artPeriod: String

switch year {
  case 1860...1885:
    artPeriod = "Impressionism"
  case 1886...1910:
    artPeriod = "Post Impressionism"
  case 1912...1935:
    artPeriod = "Expressionism"
  default:
    artPeriod = "Unknown"
}

// Prints: Post Impressionism
```

switch Statement: Compound Cases

```swift
let service = "Seamless"

switch service {
  case "Uber", "Lyft":
    print("Travel")
  case "DoorDash", "Seamless", "GrubHub":
    print("Restaurant delivery")
  case "Instacart", "FreshDirect":
    print("Grocery delivery")
  default:
    print("Unknown service")
}

// Prints: Restaurant delivery
```

switch Statement: where Clause

```
let num = 7

switch num {
  case let x where x % 2 == 0:
    print("\(num) is even")
  case let x where x % 2 == 1:
    print("\(num) is odd")
  default:
    print("\(num) is invalid")
}

// Prints: 7 is odd
```

Loops - Learn Swift

Ranges

```
let zeroToThree = 0...3

// zeroToThree: 0, 1, 2, 3
```

stride() Function

```
for oddNum in stride(from: 1, to: 5, by: 2) {
  print(oddNum)
}

// Prints: 1
// Prints: 3
```

for-in Loop

```
for char in "hehe" {
  print(char)
}

// Prints: h
// Prints: e
// Prints: h
// Prints: e
```

continue Keyword

```
for num in 0...5 {
  if num % 2 == 0 {
    continue
  }
  print(num)
}

// Prints: 1
// Prints: 3
// Prints: 5
```

break Keyword

```
for char in "supercalifragilisticexpialidocious" {
  if char == "c" {
    break
  }
  print(char)
}

// Prints: s
// Prints: u
// Prints: p
// Prints: e
// Prints: r
```

Using Underscore

Use _ instead of a placeholder variable if the variable is not referenced in the for-in loop body.

```
for _ in 1...3 {
  print("Olé")
}

// Prints: Olé
// Prints: Olé
// Prints: Olé
```

while Loop

```
var counter = 1
var stopNum = Int.random(in: 1...10)

while counter < stopNum {
  print(counter)
  counter += 1
}

// The loop prints until the stop condition is met
```

Arrays and Sets - Learn Swift

Array

```
var scores = [Int]()

// The array is empty: []
```

Initialize with Array Literal

```
// Using type inference:
var snowfall = [2.4, 3.6, 3.4, 1.8, 0.0]

// Being explicit with the type:
var temp: [Int] = [33, 31, 30, 38, 44]
```

Index

```
var vowels = ["a", "e", "i", "o", "u"]

print(vowels[0])  // Prints: a
print(vowels[1])  // Prints: e
print(vowels[2])  // Prints: i
print(vowels[3])  // Prints: o
print(vowels[4])  // Prints: u
```

.count Property

```
var grocery = ["🥓", "🥞", "🍪", "🥛", "🍊"]

print(grocery.count)

// Prints: 5
```

.append() Method and += Operator

```
var gymBadges = ["Boulder", "Cascade"]

gymBadges.append("Thunder")
gymBadges += ["Rainbow", "Soul"]

// ["Boulder", "Cascade", "Thunder", "Rainbow", "Soul"]
```

.insert() and .remove() Methods

```
var moon = ["🌖", "🌗", "🌘", "🌑"]
```

```
moon.insert("🌕", at: 0)

// ["🌕", "🌖", "🌗", "🌘", "🌑"]

moon.remove(at: 4)

// ["🌕", "🌖", "🌗", "🌘"]
```

Iterating Over an Array

```swift
var employees = ["Michael", "Dwight", "Jim", "Pam", "Andy"]

for person in employees {
  print(person)
}

// Prints: Michael
// Prints: Dwight
// Prints: Jim
// Prints: Pam
// Prints: Andy
```

Swift Sets

```swift
var paintingsInMOMA: Set = ["The Dream", "The Starry Night", "The False Mirror"]
```

Empty Sets

```swift
var team = Set<String>()

print(team)
// Prints: []
```

Populated Sets

```swift
var vowels: Set = ["a", "e", "i", "o", "u"]
```

.insert()

```swift
var cookieJar: Set = ["Chocolate Chip", "Oatmeal Raisin"]

// Add a new element
cookieJar.insert("Peanut Butter Chip")
```

.remove() and .removeAll() Methods

```swift
var oddNumbers: Set = [1, 2, 3, 5]

// Remove an existing element
oddNumbers.remove(2)

// Remove all elements
oddNumbers.removeAll()
```

.contains()

```swift
var names: Set = ["Rosa", "Doug", "Waldo"]

print(names.contains("Lola")) // Prints: false

if names.contains("Waldo"){
  print("There's Waldo!")
} else {
  print("Where's Waldo?")
}
// Prints: There's Waldo!
```

Iterating Over a Set

```swift
var recipe: Set = ["Chocolate chips", "Eggs", "Flour", "Sugar"]

for ingredient in recipe {
  print ("Include \(ingredient) in the recipe.")
}
```

.isEmpty Property

```swift
var emptySet = Set<String>()

print(emptySet.isEmpty)  // Prints: true

var populatedSet: Set = [1, 2, 3]

print(populatedSet.isEmpty) // Prints: false
```

.count Property

```swift
var band: Set = ["Guitar", "Bass", "Drums", "Vocals"]

print("There are \(band.count) players in the band.")
// Prints: There are 4 players in the band.
```

.intersection() Operation

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.intersection(setB)
print(setC)  // Prints: ["D", "C"]
```

.union() Operation

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.union(setB)
print(setC)
// Prints: ["B", "A", "D", "F", "C", "E"]
```

.symmetricDifference() Operation

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.symmetricDifference(setB)
print(setC)
// Prints: ["B", "E", "F", "A"]
```

.subtracting() Operation

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D"]

var setC = setA.subtracting(setB)
print(setC)
// Prints: ["B", "A"]
```

Assigning a Value to a Dictionary Variable

```
var primaryHex = [
  "red": "#ff0000",
  "yellow": "#ffff00",
  "blue": "#0000ff",
]

print("The hex code for blue is \(primaryHex["blue"])")
// Prints: The hex code for blue is Optional("#0000ff")

if let redHex = primaryHex["red"] {
  print("The hex code for red is \(redHex)")
}
// Prints: The hex code for red is #ff0000
```

Dictionaries - Learn Swift

Dictionary

```
var dictionaryName = [
  "Key1": "Value1",
  "Key2": "Value2",
  "Key3": "Value3"
]
```

Keys

Every key in a dictionary is unique.

```
// Each key is unique even if they all contain the same value

var fruitStand = [
  "Coconuts": 12,
  "Pineapples": 12,
  "Papaya": 12
]
```

Type Consistency

```
// Contains only String keys and Int values

var numberOfSides = [
  "triangle": 3,
  "square": 4,
  "rectangle": 4
]
```

Initialize a Populated Dictionary

Dictionary literals contain lists of key-value pairs that are separated by commas; this syntax can be used to create dictionaries that are populated with values.

```
var employeeID = [
  "Hamlet": 1367,
  "Horatio": 8261,
  "Ophelia": 9318
]
```

Initialize an Empty Dictionary

```swift
// Initializer syntax:
var yearlyFishPopulation = [Int: Int]()

// Empty dictionary literal syntax:
var yearlyBirdPopulation: [Int: Int] = [:]
```

Adding to a Dictionary

```swift
var pronunciation = [
  "library": "lai·breh·ree",
  "apple": "a·pl"
]

// New key: "programming", New value: "prow·gra·muhng"
pronunciation["programming"] = "prow·gra·muhng"
```

Removing Key-Value Pairs

```swift
var bookShelf = [
  "Goodnight Moon": "Margaret Wise Brown",
  "The BFG": "Roald Dahl",
  "Falling Up": "Shel Silverstein",
  "No, David!": "David Shannon"
]

// Remove value by setting key to nil
bookShelf["The BFG"] = nil

// Remove value using .removeValue()
bookShelf.removeValue(forKey: "Goodnight Moon")

// Remove all values
bookShelf.removeAll()
```

Modifying Key-Value Pairs

```swift
var change = [
  "Quarter": 0.29,
  "Dime": 0.15,
  "Nickel": 0.05,
  "Penny": 0.01
]

// Change value using subscript syntax
change["Quarter"] = .25

// Change value using .updateValue()
change.updateValue(.10, forKey: "Dime")
```

.isEmpty Property

```swift
var bakery = [String:Int]()

// Check if dictionary is empty
print(bakery.isEmpty)  // Prints true

bakery["Cupcakes"] = 12

// Check if dictionary is empty
print(bakery.isEmpty)  // Prints false
```

.count Property

```swift
var fruitStand = [
  "Apples": 12,
  "Bananas": 20,
  "Oranges", 17
]

print(fruitStand.count)  // Prints: 3
```

Iterating Over a Dictionary

```swift
var emojiMeaning = [
  "🤔": "Thinking Face",
  "😪": "Sleepy Face",
  "😵": "Dizzy Face"
]

// Iterate through both keys and values
for (emoji, meaning) in emojiMeaning {
  print("\(emoji) is known as the '\(meaning) Emoji'")
}

// Iterate only through keys
for emoji in emojiMeaning.keys {
  print(emoji)
}

// Iterate only through values
for meaning in emojiMeaning.values {
  print(meaning)
}
```

Functions - Learn Swift

Function Declaration

```
func washCar() -> Void {
  print("Soap")
  print("Scrub")
  print("Rinse")
  print("Dry")
}
```

Calling a Function

```
func greetLearner() {
 print("Welcome to Codecademy!")
}

// Function call:
greetLearner() // Prints: Welcome to Codecademy!
```

Returning a Value

```
let birthYear = 1994
var currentYear = 2020

func findAge() -> Int {
  return currentYear - birthYear
}

print(findAge()) // Prints: 26
```

Multiple Parameters

```
func convertFracToDec(numerator: Double, denominator: Double) -> Double {
  return numerator / denominator
}

let decimal = convertFracToDec(numerator: 1.0, denominator: 2.0)
print(decimal) // Prints:  0.5
```

Returning Multiple Values

```
func smartphoneModel() -> (name: String, version: String, yearReleased: Int) {
  return ("iPhone", "8 Plus", 2017)
}

let phone = smartphoneModel()

print(phone.name) // Prints: iPhone
print(phone.version) // Prints: 8 Plus
print(phone.yearReleased) // Prints: 2017
```

Omitting Argument Labels

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

print(findDifference(6, b: 4)) // Prints: 2
```

Parameters and Arguments

```swift
func findSquarePerimeter(side: Int) -> Int {
  return side * 4
}

let perimeter = findSquarePerimeter(side: 5)
print(perimeter) // Prints: 20

// Parameter: side
// Argument:  5
```

Implicit Return

```swift
func nextTotalSolarEclipse() -> String {
  "April 8th, 2024 🌎 "
}

print(nextTotalSolarEclipse()) // Prints: April 8th, 2024 🌎
```

Default Parameters

```swift
func timeToFinishBook(numWords: Double, wordsPerMin: Double = 200) -> Double {
  let totalMinutes = numWords / wordsPerMin
  return totalMinutes / 60
}

print("\(timeToFinishBook(numWords: 93000)) hours")
// Prints: 7.75 hours
```

Variadic Parameters

```swift
func totalStudents(students: String...) -> Int {
  let numStudents = students.count
  return numStudents
}

print(totalStudents(students: "Jamie", "Michael", "Rose", "Idris")) // Prints: 4
```

In-Out Parameters

```swift
var currentSeason = "Winter"

func determineSeason(monthNum: Int, season: inout String) {

switch monthNum {
  case 1...2:
    season = "Winter ⛄ "
  case 3...6:
    season = "Spring 🌱 "
  case 7...9:
    season = "Summer ⛱"
  case 10...11:
    season = "Autumn 🍂 "
  default:
    season = "Unknown"
  }
}

determineSeason(monthNum: 4, season: &currentSeason)

print(currentSeason) // Spring 🌱
```

Structures - Learn Swift

Struct Creation

```swift
struct Building {
  var address: String
  var floors: Int

  init(address: String, floors: Int, color: String) {
    self.address = address
    self.floors = floors
  }
}
```

Default Property Values

```swift
struct Car {
  var numOfWheels = 4
  var topSpeed = 80
}

var reliantRobin = Car(numOfWheels: 3)

print(reliantRobin.numOfWheels) // Prints: 3
print(reliantRobin.topSpeed)    // Prints: 80
```

Structure Instance Creation

```
struct Person {
  var name: String
  var age: Int

  init(name: String, age: Int) {
    self.name = name
    self.age = age
  }
}

// Instance of Person:
var morty = Person(name: "Morty", age: 14)
```

Checking Type

```
print(type(of: "abc")) // Prints: String
print(type(of: 123))   // Prints: 123
```

init() Method

Structures can have an init() method to initialize values to an instance's properties. Unlike other methods, The init() method does not need the func keyword. In its body, the self keyword is used to reference the actual instance of the structure.

```
struct TV {
  var screenSize: Int
  var displayType: String

  init(screenSize: Int, displayType: String) {
    self.screenSize = screenSize
    self.displayType = displayType
  }
}

var newTV = TV(screenSize: 65, displayType: "LED")
```

Structure Methods

```
struct Dog {
  func bark() {
    print("Woof")
  }
}

let fido = Dog()
fido.bark() // Prints: Woof
```

Mutating Methods

```swift
struct Menu {
  var menuItems = ["Fries", "Burgers"]

  mutating func addToMenu(dish: String) {
    self.menuItems.append(dish)
  }
}

var dinerMenu = Menu()

dinerMenu.addToMenu(dish: "Toast")
print(dinerMenu.menuItems)
// Prints: ["Fries", "Burgers", "Toast"]
```

Classes - Learn Swift

Swift Class

```swift
// Using data types:
class Student {
  var name: String
  var year: Int
  var gpa: Double
  var honors: Bool
}


// Using default property values:
class Student {
  var name = ""
  var year = 0
  var gpa = 0.0
  var honors = false
}
```

Instance of a Class

```swift


class Person {
  var name = ""
  var age = 0
}

var sonny = Person()

// sonny is now an instance of Person
```

Class Properties

```
var ferris = Student()

ferris.name = "Ferris Bueller"
ferris.year = 12
ferris.gpa = 3.81
ferris.honors = false
```

init() Method

```
class Fruit {
  var hasSeeds = true
  var color: String

  init(color: String) {
    self.color = color
  }
}

let apple = Fruit(color: "red")
```

Inheritance

```


// Suppose we have a BankAccount class:

class BankAccount {
  var balance = 0.0

  func deposit(amount: Double) {
    balance += amount
  }

  func withdraw(amount: Double) {
    balance -= amount
  }
}


// And we want a new SavingsAccount class that inherits from BankAccount:

class SavingsAccount: BankAccount {
  var interest = 0.0

  func addInterest() {
    let interest = balance * 0.005
    self.deposit(amount: interest)
  }
}

// Here, the new SavingsAccount class (subclass) automatically gains all of the
characteristics of BankAccount class (superclass). In addition, the SavingsAccount
class defines a .interest property and a .addInterest() method.
```

Overriding

```swift
// Suppose we have a BankAccount class:

class BankAccount {
  var balance = 0.0

  func deposit(amount: Double) {
    balance += amount
  }

  func withdraw(amount: Double) {
    balance -= amount
  }
}


// Suppose we want a new SavingsAccount class and we want to override the .withdraw() method from its superclass BankAccount:

class SavingsAccount: BankAccount {
  var interest = 0.0
  var numWithdraw = 0

  func addInterest() {
    let interest = balance * 0.01
    self.deposit(amount: interest)
  }

  override func withdraw(amount: Double) {
    balance -= amount
    numWithdraw += 1
  }
}
```

Reference Types

Classes are reference types, while structures are value types.

Unlike value types, reference types are not copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used.

Learn Intermediate Swift Cheatsheet

https://www.codecademy.com/learn/learn-intermediate-swift/modules/swift-enumerations/cheatsheet

Enumerations - Learn Intermediate Swift

```swift
enum Day {
  case monday
  case tuesday
  case wednesday
  case thursday
  case friday
  case saturday
  case sunday
}

let casualWorkday: Day = .friday
```

Switch

```
enum Dessert {
    case cake(flavor: String)
    case vanillaIceCream(scoops: Int)
    case brownie
}

let customerOrder: Dessert = .cake(flavor: "Red Velvet")

switch customerOrder {
  case let .cake(flavor):
    print("You ordered a \(flavor) cake")
  case let .vanillaIceCream(scoopCount):
    print("You ordered \(scoopCount) scoops of vanilla ice cream")
  case .brownie:
    print("You ordered a brownie")
}

// Prints: "You ordered a Red Velvet cake"
```

CaseIterable

Add conformance to the CaseIterable protocol to access an allCases property that returns an array of all the cases of an enumeration.

```
enum Season: CaseIterable {
    case winter
    case spring
    case summer
    case fall
}

for season in Season.allCases {
    print(season)
}
```

Raw Values

```
enum Grade: Character {
  case pass = "P"
  case fail = "F"
}
```

```
let mathTest = Grade.pass
print(mathTest.rawValue) // Prints "P"
```

Associated Values

```
enum Dessert {
    case cake(flavor: String)
    case vanillaIceCream(scoops: Int)
    case brownie
}

let customerOrder: Dessert = .cake(flavor: "Red Velvet")
```

Instance Methods

```
enum Traffic {
  case light
  case medium
  case heavy

  mutating func reportAccident() {
    self = .heavy
  }
}

var currentTraffic: Traffic = .light
currentTraffic.reportAccident() // currentTraffic is now .heavy
```

Computed Properties

```
enum ShirtSize: String {
  case small = "S"
  case medium = "M"
  case large = "L"
  case extraLarge = "XL"

    var description: String {
    return "This shirt size is \(self.rawValue)"
  }
}
```

Optionals - Learn Intermediate Swift

Optional Types

```
var email: String? = nil
email = "codey@codecademy.com"
```

Force Unwrapping Optionals

```swift
var name: String? = "Codey"
var email: String? = nil

print("The user's name is \(userName!)") // Prints "The user's name is Codey"
print("The user's email is \(userEmail!)") // Crashes!
```

Optional Binding

```swift
var name: String? = "Codey"
var email: String? = nil

if let name = name {
  print("The user's name is \(name)")
} else {
  print("No name available")
}

if let unwrappedEmail = email {
  print("The user's email is \(unwrappedEmail)")
} else {
  print("No email available")
}

// Prints:
// "The user's name is Codey"
// No email available
```

Multiple Optional Bindings

```swift
var name: String? = "Codey"
var email: String? = "codey@codecademy.com"

if let name = name, let email = email, email.contains("@") {
  print("Welcome to Codecademy \(name)!  Your email address is \(email)")
} else {
  print("Name is nil, email is nil, or the email is invalid")
}

// Prints "Welcome to Codecademy Codey!  Your email address is
codey@codecademy.com"
```

Guard statement

A guard block is another way to write a conditional in Swift. All guard statements must have an else block that exits the current scope if the boolean expression is false. If the guard statement is true, the code below continues executing. Optionals can be bound in a guard block using the guard let syntax.

```swift
var name: String? = "Codey"
var email: String? = "codey@codecademy.com"

func displayMessageIfValid() {
  guard let name = name, let email = email, email.contains("@") else {
    return
  }
  print("Welcome \(name)!  Your email is \(email)")
}

displayMessageIfValid()
// Prints: "Welcome Codey!  Your email is codey@codecademy.com"
```

Nil-Coalescing Operator

```swift
var email: String? = nil
print("Welcome!  Your email is \(email ?? "unknown").")

// Prints: "Welcome!  Your email is unknown."
```

Optionals and Functions

```swift
func getFirstInitial(from name: String?) -> String? {
  return name?.first
}
```

Closures - Learn Intermediate Swift

Defining a Closure

```swift
let displayWelcome = { () -> Void in
  print("Hello World!")
}

displayWelcome() // Prints: "Hello World"
```

Inputs and Outputs

Closures can accept inputs and return a value. Unlike functions, closures cannot have argument labels, only internal argument names.

```swift
let multiply = { (a: Int, b: Int) -> Int in
  return a * b
}

print(multiply(4,3)) // Prints: 12
```

Passing Closures to Functions

```swift
func combine(_ a: Int, _ b: Int, using combiner: (Int, Int) -> Int) -> Int {
  return combiner(a,b)
}

let add = { (a: Int, b: Int) -> Int in
  return a + b
}

let multiply = { (a: Int, b: Int) -> Int in
  return a * b
}

print(combine(2,5, using: add)) // Prints: 7
print(combine(2,5, using: multiply)) // Prints: 10
```

Trailing Closure Syntax

If a function's last argument is a closure, the function can be called using trailing closure syntax. Omit the last argument from the method call and close the parentheses. Then, define the closure immediately after the parentheses are closed.

```swift
func combine(_ a: Int, _ b: Int, using combiner: (Int, Int) -> Int) -> Int {
  return combiner(a,b)
}

let sum = combine(2,5) { (a: Int, b: Int) -> Int in
  return a + b
}

print(sum) // Prints: 7
```

Shorthand Argument Names

When defining a closure, the arguments in parentheses, return type, and the keyword in can be omitted in exchange for shorthand argument labels. $0 refers to the first argument and $1 refers to the second argument.

```swift
func combine(_ a: Int, _ b: Int, using combiner: (Int, Int) -> Int) -> Int {
  return combiner(a,b)
}

let sum = combine(2,5) { $0 + $1 }

print(sum) // Prints: 7
```

Common Higher-order Functions

A higher-order function is a function that takes another function as an argument. The Swift standard library provides a number of useful higher-order methods. The most commonly used are filter, map, reduce, and sorted.

```swift
let scores = [4,10,3,7,5]

let evenScores = scores.filter { $0 % 2 == 0 }
print(evenScores) // Prints: [4,10]
let doubledScores = scores.map { $0 * 2 }
print(doubledScores) // Prints: [8, 20, 6, 14, 10]
let sumOfScores = scores.reduce(0) { $0 + $1 }
print(sumOfScores) // Prints: 29
let sortedScores = scores.sorted { $0 < $1 }
print(sortedScores) // Prints: [3, 4, 5, 7, 10]
```

Escaping Closures

A closure escapes a function when it's called after the function returns. This can happen when the closure is assigned to a variable. Escaping closures must be marked with the @escaping tag in a function signature.

```swift
struct TextSaver {
  var saveAction: (String) -> Void = { print("Saving '\($0)' to disk") }

  mutating func setSaveAction(to newAction: @escaping (String) -> Void) {
    saveAction = newAction
  }
}

var saver = TextSaver()
saver.saveAction("Hello World!")
// Prints: Saving 'Hello World!' to disk
saver.setSaveAction(to: { print("Saving '\($0)' to the cloud") })
saver.saveAction("Hello World!")
// Prints: Saving 'Hello World!' to the cloud
```

Capturing Values

Closures can capture values from their surrounding scope. When a closure captures a value, it keeps track of it and can manipulate the value even if the original function returns.

```swift
func makeCountingPrinter(for str: String) -> () -> Void {
  var printCount = 0
  func strPrinter() -> Void {
    printCount += 1
    print("\(str) print count: \(printCount)")
  }
  return strPrinter
}

let printHello = makeCountingPrinter(for: "Hello!")
let printGoodbye = makeCountingPrinter(for: "Goodbye!")

printHello() // Prints: Hello! print count: 1
printHello() // Prints: Hello! print count: 2
printGoodbye() // Prints: Goodbye! print count: 1
```

```
printHello() // Prints: Hello! print count: 3
printGoodbye() // Prints: Goodbye! print count: 2
```

Properties and Access Control - Learn Intermediate Swift

Access Levels

Swift provides several different levels of access. From least restrictive to most restrictive they are:

open / public internal fileprivate private

```
// public structures can be accessed in other modules
public struct User {
  // internal is the default level of access control
    let name: String
  // fileprivate methods can only be accessed inside of the file they're declared
in
  fileprivate func incrementVisitCount() {
    visitCount += 1
  }
  // private properties can only be accessed inside their structure's definition
  private let visitCount = 0
}
```

Private Properties and Methods

Mark methods and properties as private to prevent them from being accessed outside of the structure, class, or enumeration’s definition.

```
struct User {
  let name: String
  init(name: String) {
    self.name = name
    uploadNewUser()
  }
  private func uploadNewUser() {
    print("Uploading the new user...")
  }
}
```

Read-only Computed Properties

Read-only computed properties can be accessed, but not assigned to a new value. To define a read-only computed property, either use the get keyword without a set keyword, or omit keywords entirely.

```
struct Room {
  let width: Double
  let height: Double
  var squareFeet: Double {
    return width * height
  }
  var description: String {
    get {
      return "This room is \(width) x \(height)"
    }
  }
}
```

Property Observers

Property observers execute code whenever a property is changed. The willSet property observer is triggered right before the property is changed and creates a newValue variable within the block's scope. The didSet property observer is triggered right after the property is changed and creates an oldValue within the block's scope.

```
struct Employee {
  var hourlyWage = 15 {
    willSet {
      print("The hourly wage is about to be changed from \(hourlyWage) to \
(newValue)")
    }
    didSet {
      print("The hourly wage has been changed from \(oldValue) to \(hourlyWage)")
    }
  }
}

var codey = Employee()
codey.hourlyWage = 20

// Prints:
// The hourly wage is about to be changed from 15 to 20
// The hourly wage has been changed from 15 to 20
```

Private Setters

Properties marked as private(set) can be accessed from outside the scope of its structure, but only assigned within it. This allows the setter to be more restrictive than the getter.

```
struct User {
    private(set) var name: String
  mutating func updateName(to newName: String) {
    if newName != "" {
        name = newName
    }
  }
}

var currentUser = User(name: "codey")
currentUser.updateName(to: "Codey")
print(currentUser.name)
// currentUser.name = "Bob" // This line doesn't compile because the 'name' setter
is inaccessible
```

Static Properties and Methods

The static keyword is used to declare type methods and properties. These are accessed from the type itself rather than an instance.

```swift
struct User {
  static var allUsers = [User]()
  let id: Int
  init(id: Int) {
    self.id = id
    User.allUsers.append(self)
  }
}

let userOne = User(id: 1)
let userTwo = User(id: 2)
let userThree = User(id: 3)

print(User.allUsers) // Prints: [User(id: 1), User(id: 2), User(id: 3)]
```

Extensions

The extension keyword is used to continue defining an existing class, structure, or enumeration from anywhere in a codebase. Extensions can have new methods, internal types, and computed properties, but can't contain new stored properties.

```swift
struct User {
  let name: String
}

extension User {
  var description: String {
    return "This is a user named \(name)"
  }
}
```