Swift 5.1 Cheat Sheet and Quick Reference

Declaring Constants and Variables

```
// Declaring a constant using the let keyword
let double: Double = 2.0
// double = 3.0 // Error: You can't reassign a let
let inferredDouble = 2.0 // Inferred as a Double

// Declaring a variable using the var keyword
var mutableInt: Int = 1
mutableInt = 2 // OK: You can reassign a var
```

Numeric Type Conversion

```
let integerValue = 8
let doubleValue = 8.0
// let sum = integerValue + double
// Error: type mismatch

// Use an opt-in approach that prevents hidden
// conversion errors and helps make type conversion
// intentions explicit
let sum = Double(integerValue) + double
// OK: Both values have the same type
```

Strings

```
// Using a string literal as an initial value for
// a constant or variable
let helloWorld = "Hello, World!"

// Using a multiline string literal to span
// over several lines
let helloWorldProgram = """
A "Hello, World!" program generally is a computer
program that outputs or displays the message
"Hello, World!"
"""

// Empty string
let emptyString = "" // Using string literal
// Initializer syntax
let anotherEmptyString = String()

// Mutating a string
var mutableString = "Swift"
mutableString += " is awesome!"

// String interpolation
// Interpolating a Double
print("The value is \(double)")
// Interpolating a String
print("This is my opinion: \(mutableString)")
```

Tuples

```
// Group multiple values into
// a single compound value
let httpError = (503, "Server Error")

// Decomposing a tuple's contents
let (code, reason) = httpError
// Another way to decompose
let codeByIndex = httpError.0
let reasonByIndex = httpError.1
// Ignoring parts of the tuple using _
let (_, justTheReason) = httpError
```

Optionals

```
// catchphrase can hold a String or nil
var catchphrase: String? // Automatically set to nil
catchphrase = "Hey, what's up, everybody?"

// Forced unwrapping operator (!)
// count1 contains catchphrase's count if
// catchphrase isn't nil; crashes otherwise
let count1: Int = catchphrase!.count

// Optional binding
// If the optional Int returned by
// catchphrase?.count contains a value,
// set a new constant called count to the value
// contained in the optional
if let count = catchphrase?.count {
  print(count)
}

// Coalescing operator (??)
// count2 contains catchphrase's count if
// catchphrase isn't nil; 0 otherwise
let count2: Int = catchphrase?.count ?? 0

// Chaining operator (?)
// count3 contains catchphrase's count if
// catchphrase isn't nil; nil otherwise
let count3: Int? = catchphrase?.count

// Implicitly unwrapped optionals
let forcedCatchphrase: String! =
  "Hey, what's up, everybody?"
let implicitCatchphrase = forcedCatchphrase
// No need for an exclamation mark
```

Collection Types: Array

```
let immutableArray: [String] = ["Alice", "Bob"]
// Type of mutableArray inferred as [String]
var mutableArray = ["Eve", "Frank"]
// Test the membership
let isEveThere = immutableArray.contains("Eve")
// Access by index
let name: String = immutableArray[0]
// Update item in list;
// crashes if the index is out of range
mutableArray[1] = "Bart"
// immutableArray[1] = "Bart" // Error: can't change
mutableArray.append("Ellen") // Add an item
// Add an item at index
mutableArray.insert("Gemma", at: 1)
// Delete by index
let removedPerson = mutableArray.remove(at: 1)
// You can't reassign a let collection nor change
// its content; you can reassign a var collection
// and change its content
mutableArray = ["Ilary", "David"]
mutableArray[0] = "John"
```

Collection Types: Dictionary

```
let immutableDict: [String: String] =
  ["name": "Kirk", "rank": "captain"]
// Type of mutableDict inferred as [String: String]
var mutableDict =
  ["name": "Picard", "rank": "captain"]
// Access by key, if the key isn't found returns nil
let name2: String? = immutableDict["name"]
// Update value for key
mutableDict["name"] = "Janeway"
// Add new key and value
mutableDict["ship"] = "Voyager"
// Delete by key, if the key isn't found returns nil
let rankWasRemoved: String? =
mutableDict.removeValue(forKey: "rank")
```

Collection Types: Set

```
// Sets ignore duplicate items, so immutableSet
// has 2 items: "chocolate" and "vanilla"
let immutableSet: Set =
  ["chocolate", "vanilla", "chocolate"]
var mutableSet: Set =
  ["butterscotch", "strawberry"]
// A way to test membership
immutableSet.contains("chocolate")
// Add item
mutableSet.insert("green tea")
// Remove item, if the item isn't found returns nil
let flavorWasRemoved: String? =
mutableSet.remove("strawberry")
```

Swift 5.1 Cheat Sheet and Quick Reference

Control Flow: Loops

```
// Iterate over list or set
for item in listOrSet {
    print(item)
}

// Iterate over dictionary
for (key, value) in dictionary {
    print("\(key) = \(value)")
}

// Iterate over ranges

// Closed range operator (...)
for i in 0...10 {
  print(i) // 0 to 10
}
// Half-open range operator (..<)
for i in 0..<10 {
  print(i) // 0 to 9
}

// while
var x = 0
while x < 10 {
  x += 1
  print(x)
}

// repeat-while
repeat {
  x -= 1
  print(x)
} while(x > 0)
```

Control Flow: Conditionals

```
// Using if to choose different paths
let number = 88
if (number <= 10) {
  // If number <= 10, this gets executed
} else if (number > 10 && number < 100) {
  // If number > 10 && number < 100,
  // this gets executed
} else {
  // Otherwise this gets executed
}

// Ternary operator
// A shorthand for an if-else condition
let height = 100
let isTall = height > 200 ? true : false
```

```
// Using guard to transfer program control
// out of a scope if one or more conditions
// aren't met
for n in 1...30 {
  guard n % 2 == 0 else {
    continue
  }
  print("\(n) is even")
}

// Using switch to choose different paths
let year = 2012
switch year {
case 2003, 2004:
  // Execute this statement if year is 2003 or 2004
  print("Panther or Tiger")
case 2010:
  // Execute this statement if year is exactly 2010
  print("Lion")
case 2012...2015:
  // Execute this statement if year is
  // within the range 2012-2015,
  // range boundaries included
  print("Mountain Lion through El Captain")
default:
  // Every switch statement must be exhaustive
  print("Not already classified")
}
```

Functions

```
// A Void function
func sayHello() {
  print("Hello")
}

// Function with parameters
func sayHello(name: String) {
  print("Hello \(name)!")
}

// Function with default parameters
func sayHello(name: String = "Lorenzo") {
  print("Hello \(name)!")
}

// Function with mix of default and
// regular parameters
func sayHello(name: String = "Lorenzo", age: Int) {
  print("\(name) is \(age) years old!")
}
// Using just the non default value
sayHello(age: 35)
```

```
// Function with parameters and return value
func add(x: Int, y: Int) -> Int {
  return x + y
}
let value = add(x: 8, y: 10)

// If the function contains a single expression,
// the return value can be omitted
func multiply(x: Int, y: Int) -> Int {
  x * y
}

// Specifying arguments labels
func add(x xVal: Int, y yVal: Int) -> Int {
  return xVal + yVal
}

// Omitting the argument label for one
// (or more) parameters
func add(_ x: Int, y: Int) -> Int {
    return x + y
}
let value = add(8, y: 10)

// A function that accepts another function
func doMath(operation: (Int, Int) -> Int, a: Int, b:
Int) -> Int {
  return operation(a, b)
}
```

Closures

```
let adder: (Int, Int) -> Int = { (x, y) in x + y }

// Closures with shorthand argument name
let square: (Int) -> Int = { $0 * $0 }

// Passing a closure to a function
let addWithClosure = doMath(operation: adder, a: 2,
b: 3)
```


Swift 5.1 Cheat Sheet and Quick Reference

Enumerations

```
enum Taste {
  case sweet, sour, salty, bitter, umami
}
let vinegarTaste = Taste.sour

// Iterating through an enum class
enum Food: CaseIterable {
  case pasta, pizza, hamburger
}

for food in Food.allCases {
  print(food)
}

// enum with String raw values
enum Currency: String {
  case euro = "EUR"
  case dollar = "USD"
  case pound = "GBP"
}

// Print the backing value
let euroSymbol = Currency.euro.rawValue
print("The currency symbol for Euro is \
(euroSymbol)")

// enum with associated values
enum Content {
  case empty
  case text(String)
  case number(Int)
}

// Matching enumeration values with a switch
statement
let content = Content.text("Hello")
switch content {
case .empty:
  print("Value is empty")
case .text(let value): // Extract the String value
  print("Value is \(value)")
case .number(_): // Ignore the Int value
  print("Value is a number")
}
```

Structs

```
struct User {
  var name: String
  var age: Int = 40
}

// A memberwise initializer is automatically
// created to accept parameters matching the
// properties of the struct
let john = User(name: "John", age: 35)

// Memberwise initializer uses default parameter
// values for any properties that have them
let dave = User(name: “Dave")

// Accessing properties
print("\(john.name) is \(john.age) years old")
```

Classes

```
class Person {
  let name: String
  // Class initializer
  init(name: String) {
    self.name = name
  }

  // Using deinit to perform
  // object's resources cleanup
  deinit {
    print("Perform the deinitialization")
  }

  var numberOfLaughs: Int = 0
  func laugh() {
    numberOfLaughs += 1
  }

  // Define a computed property
  var isHappy: Bool {
    return numberOfLaughs > 0
  }
}

let david = Person(name: "David")
david.laugh()
let happy = david.isHappy
```

Inheritance

```
class Student: Person {
  var numberOfExams: Int = 0

  // Override isHappy computed property
  // providing additional logic
  override var isHappy: Bool {
    numberOfLaughs > 0 && numberOfExams > 2
  }
}

let ray = Student(name: "Ray")
ray.numberOfExams = 4
ray.laugh()
let happy = ray.isHappy
// Mark Child as final to prevent subclassing
final class Child: Person { }
```

Designated & Convenience Initializers

```
// A class must have at least one
// designated initializer and may have one or more
// convenience initializers
class ModeOfTransportation {
  let name: String
  // Define a designated initializer
  // that takes a single argument called name
  init(name: String) {
    self.name = name
  }

  // Define a convenience initializer
  // that takes no arguments
  convenience init() {
    // Delegate to the internal
    // designated initializer
    self.init(name: "Not classified")
  }
}

class Vehicle: ModeOfTransportation {
  let wheels: Int
  // Define a designated initializer
  // that takes two arguments called name and wheels
  init(name: String, wheels: Int) {
    self.wheels = wheels
    // Delegate up to the superclass
    // designated initializer
    super.init(name: name)
  }
  // Override the superclass convenience initializer
  override convenience init(name: String) {
    // Delegate to the internal
    // designated initializer
    self.init(name: name, wheels: 4)
  }
}
```

Swift 5.1 Cheat Sheet and Quick Reference

Extensions

```
// Extensions add new functionality to an existing
// class, structure, enumeration, or protocol type
extension String {

  // Extending String type to calculate
  // if a String instance is truthy or falsy
  var boolValue: Bool {
    if self == "1" {
      return true
    }
    return false
  }
}

let isTrue = "0".boolValue
```

Error Handling

```
// Representing an error
enum BeverageMachineError: Error {
  case invalidSelection
  case insufficientFunds
  case outOfStock
}

func selectBeverage(_ selection: Int) throws ->
String {
  // Some logic here
  return "Waiting for beverage..."
}

// 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
let message: String
do {
  message = try selectBeverage(20)
} catch BeverageMachineError.invalidSelection {
  print("Invalid selection")
} catch BeverageMachineError.insufficientFunds {
  print("Insufficient funds")
} catch BeverageMachineError.outOfStock {
  print("Out of stock")
} catch {
  print("Generic error")
}

// If an error is thrown while evaluating the try?
// expression, the value of the expression is nil
let nillableMessage = try? selectBeverage(10)

// If an error is throws you'll get a runtime error,
// otherwise the value
let throwableMessage = try! selectBeverage(10)
```

Coding Protocols

```
import Foundation

// Codable conformance is the same as conforming
// separately to Decodable and Encodable protocols
struct UserInfo: Codable {
  let username: String
  let loginCount: Int
}

// Conform to CustomStringConvertible to provide
// a specific representation when converting the
// instance to a string
extension UserInfo: CustomStringConvertible {
  var description: String {
    return "\(username) has tried to login \
(loginCount) time(s)"
  }
}

// Define multiline string literal to represent JSON
let json = """
{ "username": "David", "loginCount": 2 }
"""

// Using JSONDecoder to serialize JSON
let decoder = JSONDecoder()

// Transform string to its data representation
let data = json.data(using: .utf8)!
let userInfo = try! decoder.decode(UserInfo.self,
from: data)
print(userInfo)

// Using Encodable to serialize a struct
let encoder = JSONEncoder()
let userInfoData = try! encoder.encode(userInfo)

// Transform data to its string representation
let jsonString = String(data: userInfoData,
encoding: .utf8)!
print(jsonString)
```

Access Control

```
// A module - a framework or an application - is
// a single unit of code distribution that can be
// imported by another module with import keyword

// Class accessible from other modules
public class AccessLevelsShowcase {

  // Property accessible from other modules
  public var somePublicProperty = 0

  // Property accessible from the module
  // is contained into
  var someInternalProperty = 0

  // Property accessible from its own
  // defining source file
  fileprivate func someFilePrivateMethod() {}

  // Property accessible from its
  // enclosing declaration
  private func somePrivateMethod() {}
}
```