iOS Cheat Sheet

Variables

- `Named storage that we can manipulate later on`
- Main types
  - String, Int, Double, Float, Bool, Character

```
var zero = 0
var one = 1
print(zero + one) // prints 1
```

- You can also declare a variable as such

```
var nameOfPerson: String = "Person"
print(nameOfPerson) // prints "Person
```

- **Let** keyword creates a *constant* variable (value can’t be changed after declaration)

- **Var** keyword creates a variables whose value can be changed later on

---

Functions

- A set of statements organized to facilitate a certain task
- You can also have multiple parameters
- All parameters accepted by a function are treated as constants by default
- You can also have functions as parameters

```
// structure
func funcname(Parameters) -> returntype {
   Statement1
   Statement2
   ---
   Statement N
   return parameters
}

// example
func addTwoNumbers(num1: Int, num2: Int) -> Int {
        return num1 + num2
}

// call function with
var result = addTwoNumbers(num1: 7, num2: 8)
print(result) // 15
```

- If the function contains a *single expression* the “return” statement can be omitted

```
// One expression
func greeting() -> String {
isBirthday ? "Happy Birthday" : "Hello"
}

// Multiple
func greeting() -> String {
if isBirthday {
        return "Happy Birthday"
    } else {
        return "Hello"
  }
}
```

- You can use ‘_’ to match any value, if you’re not interested in the vale of a particular element or variable

```
func greet(_ name: String) {
    print ("Hello, \(name)!")
}

greet("John") // Output: Hello, John!

// OTHERWISE would be

func greet(name: String) {
    print ("Hello, \(name)!")
}

greet(name: "John") // Output: Hello, John!
```

- Default values for parameter and default values for names

```
func buildMessageFor(_ name: String = “Customer”, _ count: Int)
{
```

```
return("\(name), you are customer number \(count)")
}

// call

let message = buildMessageFor(count: 100)

print(message) // Print: Customer, you are customer 100
```

- You can also have multiple returns

```
func testFunction(_ length: Float) -> (yards: Float, feet: Float
    return (yards, feet) // this is a tuple
}
```

- `inout` parameters allow you to pass a parameter into a function by reference, meaning that the function can modify the value of the parameter directly.

```
func swapInts(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}
```

- **Calling a Function with `inout` Parameters**:
  - When calling a function with `inout` parameters, you prefix the argument with `&` to indicate that it can be modified by the function.

```
swiftCopy code
var x = 5
var y = 10
```

```
swapInts(&x, &y)
```

Operators

List of operators: +, -, /, *, +=, -=, /=, *=

- **Ternary operators**
  - Provide a shortcut way of making decisions within code.

  condition ? true expression : false expression
  - condition will return a true or false value
  - If the results is true then the true expression is evaluated
  - If the results is false, the false expression is evaluated.
- **??** allows for a default value in the case of a nil return

Classes

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

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

- Classes can have instance methods and class methods

- **Instance methods** can be called normally, however **class methods** must be called by a class object

```
class testClass {
var name: String
var age: Int

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

deinit {
print("Perform cleanup")
}

func regularFunction() {
    print("Doing something")
}

class func classFunction() {
    print ("I'm a class function")
}

}

var user = testClass(name: "Jaden", age: 20)
user.regularFunction()
testClass.classFunction() // must be called by the class since i
```

Instances

- The occurrence of a class, structure, or enumeration

```
// class
class MyClass {
    var property: String

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

let instanceOfMyClass = MyClass(property: "Some Value")

// structure
struct MyStruct {
    var property: String
}

let instanceOfMyStruct = MyStruct(property: "Some Value")

// enumeration
enum MyEnum {
    case caseOne
    case caseTwo
}

let instanceOfMyEnum = MyEnum.caseOne
```

Control Flow

Control flow refers to the order in which the individual statements or instructions of a program are executed. In Swift, like in many programming languages, control flow mechanisms allow you to dictate the order in which your code is executed based on conditions, loops, and branching

- **for-in**

```
for index in 1...5 {
    print("Value of index is \(index)")
}

// can also be changed to _
for _ in 1...5 {
    print("Value of index is \(index)")
}
```

- **from:to:by**

```
// Allows you to skip unwanted marks
let minuteInterval = 5
let minutes = 60
for tickMark in stride(from: 0, to: minutes, by: minuteInterval)
{
}
```

- **While**

```
while condition {
// code
}
```

- **repeat while**

```
var i = 10
repeat {
i -= 1
} while (i > 0)
```

- **Guard**
  - **IMPORTANT:** Runs when a condition is **NOT** met

```
// no guard
func voteEligibility() {
    var age = 42
if age >= 18 {
    print("Eligible to vote")
}
else {
    print("Not eligible to vote")
}
}
voteEligibility()

//guard
func voteEligibility() {
var age = 42
guard age >= 18 else {
    print("Not Eligible to vote")
return
}
    print("Eligible to vote")
}
voteEligibility()
```

- **Switch statements**

```
let temperature = 83
switch (temperature)
{
case 0...49:
    print("Cold")
case 50...79:
    print("Warm")
case 80...110:
    print("Hot")
default:
```

```
    print("Temperature out of range")
}
```

- **Where and switch**

```
let temperature = 54
switch (temperature)
{
case 0...49 where temperature % 2 == 0:
    print("Cold and even")
case 50...79 where temperature % 2 == 0:
    print("Warm and even")
case 80...110 where temperature % 2 == 0:
    print("Hot and even")
default:
    print("Temperature out of range or odd")
}
```

- **Tuples and switch**

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```

Loops

- Similar to some control flow
- Includes For-in, while, repeat-while, stride
- Allows you to execute blocks of code repeatedly
- **labeling loops**

```
outerLoop: for i in 1...3 {
    for j in 1...3 {
        if i * j == 6 {
            break outerLoop // Exit the outer loop
        }
        print("\(i) * \(j) = \(i * j)")
    }
}
```

- **Iterating through arrays and dictionaries & multiple iterators**

```
let names = ["Alice", "Bob", "Charlie"]
for (index, name) in names.enumerated() {
    print("\(index): \(name)")
}

// multiple iterators
let numbers = [1, 2, 3]
let letters = ["a", "b", "c"]
for (number, letter) in zip(numbers, letters) {
    print("\(number) - \(letter)")
}
```

Conditionals

- Conditionals include if, if-else, switch, guard
- *Covered in above examples*
- **Ternary control operator**

```
let number = 10
let result = number > 0 ? "Positive" : "Non-positive"
print(result) // Output: Positive
```

- **Pattern matching * (Cool)**

```
let point = (2, 2)
switch point {
    case (0, 0):
        print("Origin")
    case (_, 0):
        print("On x-axis")
    case (0, _):
        print("On y-axis")
    case (-2...2, -2...2):
        print("Inside the square")
    default:
        print("Outside the square")
}
```

- **Value binding in switch statments * (Cool)**

```
let somePoint = (1, 1)
switch somePoint {
    case (let x, 0):
        print("On the x-axis with x = \(x)")
    case (0, let y):
        print("On the y-axis with y = \(y)")
```

```
    case let (x, y):
        print("Somewhere else at (\(x), \(y))")
}
```

Strings

- **Creating a string**

```
let testString = "Hello World"
```

- You can also call other strings, withing strings

```
let otherString = "test"
let mainString = "I can't wait to take this \(otherString)"
// output: I can't wait to take this test
```

- You can also concatenate strings

```
let firstName = "Jaden"
let lastName = "Johnson"
let fullName = firstName + " " + lastName
print(fullName) // output: Jaden Johnson
```

- String things
  - **.count:** finds the length of the string
  - **Can index strings**

```
let name = "Jaden"
print(name[2])

// can also compare strings and do += to add things to strings
```

Dictionaries

- A key-value pairing type structure

```
var person: [String: Any] = ["name": "John", "age": 30, "isStude
```

- You can pre initialize values into a dictionary

```
let keys = ["100-432112", "200-532874", "202-546549", "104-1098
let values = ["Wind in the Willows", "Tale of Two Cities",
"Sense and Sensibility", "Shutter Island"]
let bookDict = Dictionary(uniqueKeysWithValues: zip(keys, values

// can use print(bookDict.count) to get dictionary item count


OTHER DICTIONARY FUNCTIONS:
Accessing Dictionaries
• print(bookDict["200-532874"]) <- by using a key
• print(bookDict["999-546547", default: "Book not\found"]) <- us

// can update using .updateValue()
// can add by:
bookDict[key] = value
// can remove using .removeValue()

//iterate via
for (bookid, title) in bookDict {
    print("Book ID: \(bookid) Title: \(title)")
}
```

Arrays

- Can only hold values of the same type
- Ex:

```
var nameArray = ["Jaden", "Joe", "Jameson"]
let otherArray: [Int] = [1, 2, 3]

// can also initialize with a starting size

var variableName = [String] (repeating: “My String”, count: 10)
// can also add arrays
// ex: let array3 = array1 + array2
```

- can use .isEmpty to check if array is empty (returns true or false)
- can access array as per usual with myArray[1], myArray[0]
  - **special functions:**
  - **.shuffled()** & **.randomElement()**
  - **.remove()(at : <location>)** & **.removeLast()**
- can iterative with for each

```
treeArray.forEach { tree in
    print(tree)
}
```

- **Mixed arrays**

```
let mixedArray: [Any] = ["Test word", 1, 3.5]
```

Closures

- easy way to pass around functionality in code

```
let greet = {
    print("Hello, World!")
}
greet() // Output: Hello, World!
```

- **Closures** let you reuse code and also allows you to be flexible with code that you have already made
- This also encapsulates some of your code so that it doesn’t show as much of the backend when you are trying to work on more complex environments

---

Guard

- Works **DIFFERENT** than if|if else
- Only activates when a condition **DOESN’T** hit

```
// no guard
func voteEligibility() {
    var age = 42
if age >= 18 {
    print("Eligible to vote")
}
else {
    print("Not eligible to vote")
}
}
voteEligibility()

//guard
func voteEligibility() {
var age = 42
guard age >= 18 else {
    print("Not Eligible to vote")
```

```
return
}
    print("Eligible to vote")
}
voteEligibility()
```

Tuples

- tuples hold multiple different types of data

```
let person = ("John", 30, true)
```

- you can also print out and utilize the various parts of a tuple

```
let (name, age, isStudent) = person
print(name)       // Output: John
print(age)        // Output: 30
print(isStudent)  // Output: true

// you can also name tuple elements

let person = (name: "John", age: 30, isStudent: true)
print(person.name)       // Output: John
print(person.age)        // Output: 30
print(person.isStudent)  // Output: true
```

- Helpful because unlike arrays, tuples can carry multiple different types of values, and can also have each separate part of the tuple be utilized

---

Enumerations

- Essentially allows you to group data
- Can be done by creating a bunch of “cases”

```
enum CompassDirection {
    case north
    case south
    case east
    case west
}

let direction: CompassDirection = .north

// can also use these with SWITCH statements

switch direction {
    case .north:
        print("Heading North")
    case .south:
        print("Heading South")
    case .east:
        print("Heading East")
    case .west:
        print("Heading West")
}
```

Other

- Error types
- Can define errors using enumerators

```
For example:
```

```
enum FileTransferError: Error {
case noConnection
case lowBandwidth
case fileNotFound
}

* Example from notes

• Functions can only throw errors using the throws keyword
func transferFile() throws {
}
• Functions with that thrown errors can also return a results:
func transferFile() throws -> Bool {
}
```

- This is similar to try and catch from other various coding languages