Cheat Sheet V2

Variables

```
var mutableVar = "I can change"
let immutableConstant = "I cannot change"
```

Functions

Basic Function

```
func functionName(param1: Type, param2: Type) ->
ReturnType {
    // function body
    return value
}

// Calling the function
let result = functionName(param1: value1, param2:
value2)
```

Implicit Return (for single-expression functions)

```
func square(value: Int) -> Int {
    value * value
}

let squaredValue = square(value: 5)  // 25
```

Using Tuples (multiple return values)

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    // ... find min and max
```

```
    return (minValue, maxValue)
}

let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("Minimum is \(bounds.min) and Maximum is \
(bounds.max)")
```

In-Out Parameters (modify external variables)

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

var someInt = 3
var anotherInt = 107
swapTwoInts(a: &someInt, b: &anotherInt)
print("someInt is now \(someInt), and anotherInt is now
\(anotherInt)")  // 107, 3
```

- Only variables can be passed as `inout`. Constants and literals cannot be passed due to their immutability.
- when passing a variable to an `inout` parameter, prepend the variable with an `&` to indicate it can be modified by the function.

Operators

Arithmetic Operators

- `+`, `-`, `*`, `/`, `%`

Comparison Operators

- `==`, `!=`, `<`, `>`, `<=`, `>=`

Logical Operators

- `&&` (AND), `||` (OR), `!` (NOT)

Compound Assignment Operators

- `+=`, `-=`, `*=`, `/=`, `%=`, etc.

```
var a = 5
a += 3  // a = 8
```

Range Operators

- Closed Range Operator (`...`): Defines a range that runs from a to b, and includes both a and b.

```
for idx in 1...5 {
    print(idx)  // 1, 2, 3, 4, 5
}
```

- Half-Open Range Operator (`..<`): Defines a range that runs from a to b but does not include b.

```
for idx in 1..<5 {
    print(idx)  // 1, 2, 3, 4
}
```

Ternary Conditional Operator (`? :`)

- Shorthand for a simple `if-else` statement.

```
let condition = true
let result = condition ? "True value" : "False
value"  // result = "True value"
```

Nil Coalescing Operator (`??`)

- A shorthand for unwrapping an optional with a default value if the optional is `nil`.

```
let optionalValue: Int? = nil
let defaultValue = 42
let value = optionalValue ?? defaultValue  // value
= 42
```

Classes

```
class ClassName {
    var property: Type
    init(param: Type) {
        self.property = param
    }
    func method() {
        // ...
    }
}
```

- Type method: called within the class to get the maximum allowable balance

```
Class func getMaxBalance ()
-> Float {
  return 10000.0
}
```

- Instance methods: called to display the account balance

```
func displayBalance ()
{
  print ("Number \
  (accountNumber)")

print ("Current balance is \
   (accountBalance)")
}
```

```
init (number: Int, balance: Float)
{
    accountNumber = number
    accountBalance = balance
}
```

    - can also do deinit
- inheritance

Instances

```
let instance = ClassName(param: value)
```

Control Flow

```
// If-Else
if condition {
    // code
} else if anotherCondition {
    // code
} else {
    // code
}

// Switch
switch variable {
case value1:
    // code
case value2:
    // code
default:
    // code
}
```

1. For-In Loop

- Iterates over sequences like arrays, dictionaries, ranges, and strings.

```
for item in collection {
    // code
}
```

2. For Loop (`From:to:by` equivalent)

- This style of loop isn't natively supported in Swift like it is in some languages. However, the `stride(from:to:by:)` function achieves similar behavior.

```
for i in stride(from: startValue, to: endValue, by:
stepValue) {
    // code
}
```

3. While Loop

- Evaluates the condition before the loop executes.

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

4. Repeat-While Loop

- Evaluates the condition after the loop body executes.

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

5. Break

- Exits the entire control flow statement immediately.

```
for number in numbers {
    if number < 0 {
        break
    }
}
```

6. Continue

- Stops the current iteration and moves on to the next one.

```
for number in numbers {
    if number < 0 {
        continue
    }
    // code for positive numbers
}
```

7. Ranges

- Closed Range (`...`): a...b (includes both `a` and `b`)
- Half-Open Range (`..<`): a..<b (includes `a` but not `b`)

```
for i in 1...5 {  // 1, 2, 3, 4, 5
    print(i)
}
```

8. Switch

- A multiway branch statement. Swift's `switch` is very powerful and can match various types.

```
switch valueToCheck {
case value1:
    // code
case value2:
    // code
default:
    // code
}
```

9. Where in Switch

- Used to check for additional conditions.

```
switch someValue {
case let x where x > 5:
    print("Greater than 5")
default:
    print("Not greater than 5")
}
```

10. Switch with Tuples

- Allows checking multiple values.

```
let somePoint = (1, 2)
switch somePoint {
case (0, 0):
    print("At the origin")
case (_, 0):
    print("On the x-axis")
case (0, _):
    print("On the y-axis")
case let (x, y) where x == y:
    print("On the line x == y")
default:
```

```
    print("Somewhere else")
}
```

Swift's control flow, especially its `switch` statement, is very expressive and can match and evaluate a wide range of conditions. Always remember to cover all possible values in a `switch`—if you're not sure about some, you can use the `default` case as a catch-all.

Loops

```
// For-In
for item in collection {
    // code
}

// While
while condition {
    // code
}

// Repeat-While
repeat {
    // code
} while condition
```

Conditionals

- See Control Flow

Strings

```
var str = "Hello, World"
```

Dictionaries

```
var dict: [KeyType: ValueType] = ["key1": value1,
"key2": value2]
```

Arrays

```
var array: [Type] = [item1, item2, item3]
```

Closures

```
let closure: (Type) -> ReturnType = { (param: Type) ->
ReturnType in
    // code
    return value
}
```

Guard

```
func someFunction() {
    guard condition else {
        return
    }
    // continue if condition is true
}
```

Tuples

```
let tuple = (item1, item2)
```

Enumerations

```
enum EnumName {
    case case1
```

```
    case case2(value: Type)
}
```

Inheritance

In Swift, a class can inherit (or "subclass") from another class, allowing it to inherit the properties, methods, and other characteristics of the superclass. This is a fundamental aspect of object-oriented programming.

Base (Superclass)

```
class Animal {
    var name: String
    init(name: String) {
        self.name = name
    }
    func speak() {
        print("Some generic animal sound")
    }
}
```

Derived (Subclass)

- Inherits from `Animal` in this case.

```
class Dog: Animal {
    var breed: String

    // Subclass specific initializer
    init(name: String, breed: String) {
        self.breed = breed
        super.init(name: name)  // Calling the
superclass's initializer
    }
```

```
    // Method Overriding
    override func speak() {
        print("\(name) says Woof!")
    }
}
```

Usage

```
let dog = Dog(name: "Buddy", breed: "Golden Retriever")
dog.speak()  // Buddy says Woof!
```

Key Points:

- A subclass can have its own properties and methods in addition to the inherited ones.
- The `override` keyword is used to indicate that a subclass's method is meant to override a method declared in its superclass.
- A subclass can call methods and access properties of its superclass using the `super` keyword.
- In Swift, classes can inherit from only one superclass (single inheritance). If you want to adopt multiple behaviors, consider using protocols (similar to interfaces in other languages).