# Swift 5 Cheatsheet

## Table Of Contents



## Variables

Use `var` for variables that can change ("mutable") and `let` for constants that can't change ("non-mutable"). Variable, constant – it's in the name!

*Integers* are "whole" numbers, i.e. numbers without a fractional component.

```
var meaningOfLife: Int = 42
```

*Floats* are decimal-point numbers, i.e. numbers with a fractional component. They're single precision, with as little as 6 decimal digits.

```
var phi: Float = 1.618
```

*Doubles* are floating point numbers with double precision, i.e. numbers with a fractional component. Doubles are preferred over floats.

```
let pi: Double = 3.14159265359
```

A *String* is a sequence of characters, like text.

```
var message: String = "Hello World!"
```

You use *Booleans* for logical operations. A boolean can be either `true` or `false`. We have George Boole (1815–1864) to thank for its name.

```
var isLoggedIn: Bool = false
```

You can assign the result of an expression to a variable, like this:

```
var result: Int = 1 + 2
```

An *expression* is programmer lingo for putting stuff like variables, operators, constants, functions, etc. together that the Swift compiler can "evaluate". Like this:

```
let a = 3
let b = 4
let c = a + b
```

Swift can determine the *type* ( `Int`, `Double`, `String`, etc.) of a variable on its own. This is called *type inference*. In this example, the type of `foo` is inferred to be `String`.

```
var foo = "bar"
```

**Fun Fact:** "Foobar" is a placeholder name that programmers use to denote variable names, functions, etc. when its exact name isn't that important. For instance, when you show example code in a Swift cheatsheet. But… *foobar* is confusing, so let's not use it in tutorials, courses and programming books!)

Variables can be *optional*, which means it either contains a value or it's `nil`. Optionals make coding Swift safer and more productive. Here's an optional string:

```
var optionalMessage: String?
```

# Functions

*Functions* are containers of Swift code. They have input and output. You often use functions to create abstractions in your code, and to make your code reusable.

Here's an example of a function:

```
func greetUser(name: String, bySaying greeting:String = "Hello") -> String
{
    return "\(greeting), \(name)"
}
```

This function has two *parameters* called `name` and `greeting`, both of type `String`. The second parameter `greeting` has an *argument label* `bySaying`. The return type of the function is `String`, and its code is written between those squiggly brackets.

You call the function like this:

```
let message = greetUser(name: "Reinder", bySaying: "Good Morning")
```

The cool thing about functions is that you code them once, and then call them thousands of times. It's (almost) always better to be lazy than tired!

See how the function gets two "input" arguments and one "output" value? See how the function uses the argument label in the previous example? See how the result of the function is assigned to `message`? Good.

When using documentation, or in courses, you'll often see a special function syntax. Like this: `greetUser(name:bySaying:)`. See how it leaves some stuff out? This makes it easier to talk about the function.

## Classes, Objects, Properties

Classes are basic building blocks for apps. They can contain functions, sometimes called *methods*, and variables, called *properties*.

```
class Office: Building, Constructable
{
    var address: String = "1 Infinite Loop"
    var phone: String?

    @IBOutlet weak var submitButton:UIButton?

    lazy var articles:String = {
        return Database.getArticles()
    }()

    override init()
    {
        address = "1 Probability Drive"
    }

    func startWorking(_ time:String, withWorkers workers:Int)
```

```
    {
        print("Starting working at time \(time) with workers \(workers)")
    }
}
```

The class definition starts with `class`, then the class name `Office`, then the `Building` class it *inherits* from, and then the `Constructable` *protocol* it conforms to. Inheritance, protocols, all that stuff, is part of *Object-Oriented Programming*.

Properties are variables that belong to a class instance. This class has 4 of them: `address`, `phone`, the outlet `submitButton` and the lazy computed property `articles`. Properties can have a number of attributes, like `weak` and `@IBOutlet`.

The function `init()` is *overridden* from the superclass `Building`. The class `Office` is a *subclass* of `Building`, so it inherits all functions and properties that `Building` has.

# Control Flow

## Conditionals

This is an `if`-statement, or *conditional*. You use them to make decisions based on logic.

```
if isActive
{
    print("This user is ACTIVE!")
} else {
    print("Inactive user...")
}
```

You can combine multiple conditionals with the if-elseif-else syntax, like this:

```
var user:String = "Bob"

if user == "Alice" && isActive
{
    print("Alice is active!")
}
else if user == "Bob" && !isActive
{
    print("Bob is lazy!")
}
else
{
    print("When all else fails. Alice is inactive, or Bob is active, or the user is ne
ither Bob nor Alice.")
}
```

**Fun Fact:** Like *foobar*, Alice and Bob are fictional characters that are used as placeholder names. They were invented in a 1978 paper about cryptography.

See that `&&`? That's the *Logical AND* operator. You use it to create logical expressions that can be evaluated to `true` or `false`. Like this:

```
if user == "Deep Thought" || meaningOfLife == 42
{
    print("The user is Deep Thought, or the meaning of life is 42...")
}
```

Conditionals can be challenging to comprehend. Like this:

```
if c < a && b + c == a
{
    print("Will this ever happen?")
}
```

## Loops

Loops repeat stuff. It's that easy. Like this:

```
for i in 1...5 {
    print(i)
}
```

This prints `1` to `5`, including `5`! You can also use the *half-open range operator* `a..<b` to loop from `a` to `b` *not including* `b`. Like this:

```
for i in 1..<4 {
    print(i)
}
// Output: 1 2 3
```

When you don't know how many times a loop needs to run *exactly*, you can use a `while` loop. Like this:

```
while b <= 60 && b > 0
{
    print(b)
    b -= 1
}
```

## Switch

A `switch` statement takes a value and compares it against one of several *cases*. It's similar to the `if-else` `if-else` conditional, and it's an elegant way of dealing with multiple states.

An example:

```
let weather = ...

switch weather
{
    case .rain:
        print("Bring a raincoat!")
    case .clear, .sunny:
        print("Don't forget your sunglasses.")
    case .overcast:
        print("It's really depressing.")
    case .tsunami, .earthquake:
        print("OH NO! BIG WAVE!")
    default:
        print("Expect the best, prepare for the worst.")
}
```

In Swift, `switch` statements don't have an implicit *fall-through*, but you can use `fallthrough` explicitly. Every case needs to have at least one line of code in it.

The `switch` statement only executes one case. As such, you don't have to use a `break` explicitly to end a case. You can use it to break out of a case, though.

The `switch` cases need to be *exhaustive*, or you need to provide a `default` case. For instance, when you're working with an `enum`, you'll need to incorporate every value in the enumeration.

You can also use Swift *ranges* to match interval for numbers, use tuples to match partial values, and use Swift's `where` keyword to check for additional conditions.

## Strings

Strings are pretty cool. Here's an example:

```
var jobTitle: String = "iOS App Developer"
```

Inside a string, you can use *string interpolation* to string together multiple strings. Like this:

```
var hello = "Hello, \(jobTitle)"
// Output: Hello, iOS App Developer
```

You can also turn an `Int` into a `String`:

```
let number = 42
let numberAsString = "\(number)"
```

And vice-versa:

```
let number = "42"
let numberAsInt = Int(number)
```

## Optionals

*Optionals* can either be `nil` or contain a value. You **must** always *unwrap* an optional before you can use it.

This is Bill. Bill is an optional.

```
var bill: String? = nil
```

You can unwrap `bill` in a number of ways. First, this is *optional binding*.

```
if let definiteBill = bill {
    print(definiteBill)
}
```

In this example, you bind the non-optional value from `bill` to `definiteBill` *but only when* `bill` *is not* `nil`. It's like asking: "Is it not `nil`?" OK, if not, then assign it to this constant and execute that stuff between the squiggly brackets.

You can also use *force-unwrapping* to unwrap an optional. Like this:

```
var droid: String? = "R2D2"

if droid != nil {
    print("This is not the droid you're looking for: \(droid!)")
}
```

See how that `droid` is force-unwrapped with the exclamation mark `!`? You should keep in mind that if `droid` is `nil` when you force-unwrap it, your app will crash.

You can also use *optional chaining* to work your way through a number of optionals. This saves you from coding too much optional binding blocks. Like this:

```
view?.button?.title = "BOOYAH!"
```

In this code, `view`, `button` and `title` are all optionals. When `view` is `nil`, the code "stops" before `button`, so the `button` property is never accessed.

One last thing... the *nil-coalescing operator*. You can use it to provide a default value when an expression results in `nil`. Like this:

```
var meaningOfLife = deepThought.think() ?? 42
```

See that `??`. When `deepThought.think()` returns `nil`, the variable `meaningOfLife` is `42`. When that function returns a value, it's assigned to `meaningOfLife`.

## Arrays

Arrays are a collection type. Think of it as a variable that can hold multiple values, like a closet that can contain multiple drawers. Arrays always have *numerical* index values. Arrays always contain elements of the same type.

```
var hitchhikers = ["Ford", "Arthur", "Zaphod", "Trillian"]
```

You can add items to the array:

```
hitchhikers += ["Marvin"]
```

You can get items from the array with *subscript syntax*:

```
let arthur = hitchhikers[1]
```

Remember that arrays are *zero-index*, so the index number of the first element is `0` (and not `1`).

You can iterate arrays, like this:

```
for name in hitchhikers {
    print(name)
}
```

## Dictionaries

Dictionaries are also collection types. The items in a dictionary consists of key-value pairs. Unlike arrays, you can set your own key type. Like this:

```
var score = [
    "Fry": 10,
    "Leela": 29,
    "Bender": 1,
    "Zoidberg": 0
]
```

What's the type of this dictionary? It's `[String: Int]`. Just like with arrays, you can use *subscript syntax* to get the value for a key:

```
print(score["Zoidberg"])
// Output: 0
```

You can also iterate a dictionary, like this:

```
for (name, points) in score
{
    print("\(name) has \(points) points");
}
```

## Closures

With *closures* you can pass around blocks of code, like functions, as if they are variables. You use them, for instance, by passing a callback to a lengthy task. When the task ends, the callback – a closure – is executed.

You define a closure like this:

```
let authenticate = { (name: String, userLevel: Int) -> Bool in
    return (name == "Bob" || name == "Alice") && userLevel > 3
}
```

You call the closure like this:

```
authenticate("Bob", 7)
```

If we had a user interface for authenticating a user, then we could pass the closure as a callback like this:

```
let loginVC = MyLoginViewController(withAuthCallback: authenticate)
```

Another use case for closures is multi-threading with Grand Central Dispatch. Like this:

```
DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + .seconds(60)) {
    // Dodge this!
}
```

In the above example, the last argument of `asyncAfter(deadline:execute:)` is a closure. It uses the *trailing closure* syntax. When a closure is the last argument of a function call, you can write it after the function call parentheses and omit the argument label.

# Guard and Defer

## Guard

The `guard` statement helps you to return functions early. It's a conditional, and when it isn't met you need to exit the function with `return`.

Like this:

```
func loadTweets(forUserID userID: Int) {
    guard userID > 0 else {
        return
    }

    // Load the tweets...
}
```

You can read that as: *"Guard that the User ID is greater than zero, or else, exit this function"*. Guard is especially powerful when you have multiple conditions that should return the function.

## Defer

With `defer` you can define a code block that's executed when your function returns. The `defer` statement is similar to `guard`, because it also helps with the flow of your code.

Like this:

```
func saveFile(withData data: Data) {

    let filePointer = openFile("../example.txt")

    defer {
        closeFile(filePointer)
    }

    if filePointer.size > 0 {
        return
    }

    if data.size > 512 {
        return
    }

    writeFile(filePointer, withData: data)
}
```

In the example code you're opening a file and writing some data to it. As a rule, you need to close the file pointer before exiting the function.

The file isn't written to when two conditions aren't met. You have to close the file at those points. Without the `defer` statement, you would have written `closeFile(_:)` twice.

Thanks to `defer`, the file is always closed when the function returns.

## Generics

In Swift your variables are *strong typed*. When you set the type of animals your farm can contain to `Duck`, you can't change that later on. With *generics* however, you can!

Like this:

```
func insertAnimal<T>(_ animal: T, inFarm farm: Farm)
{
    // Insert `animal` in `farm`
}
```

This is a *generic function*. It uses a *placeholder type* called `T` instead of an actual type name, like `String`.

If you want to insert ducks, cows, birds and chickens in your farm, you can now do that with one function instead of 4.

## Tuples

With *tuples* you get two (or more) variables for one. They help you structure your code better. Like this:

```
let coffee = ("Cappuccino", 3.99)
```

You can now get the price of the coffee like this:

```
let (name, price) = coffee
print(price)
// Output: 3.99
```

When you need just the name, you can do this:

```
let (name, _) = coffee
print(name)
// Output: Cappuccino
```

You can also name the elements of a tuple, like this:

```
let flight = (code: "XJ601", heading: "North", passengers: 216)
print(flight.heading)
// Output: North
```

## Error Handling

Errors in Swift can be thrown, and should be caught. You can define an error type like this:

```
enum CreditCardError: Error {
    case insufficientFunds
    case issuerDeclined
    case invalidCVC
}
```

When you code a function that can throw errors, you have to mark its function definition with `throws`. Like this:

```
func processPayment(creditcard: String) throws {
    ...
```

Inside the function, you can then throw an error like this:

```
throw CreditCardError.insufficientFunds
```

When you *use* a function that can throw errors, you have to wrap it in a `do-try-catch` block. Like this:

```
do {
    try processPayment(creditcard: "1234.1234")
}
catch let error {
    print(error)
}
```

In the example above, the `processPayment(creditcard:)` function is marked with the `try` keyword. When an error occurs, the `catch` block is executed.

## Resources

No cheatsheet is complete without a list of resources with more information. Wanna see how deep the rabbit hole really goes?

- Swift Language Guide
- Swift Evolution
- Swift Standard Library
- Apple Developer Documentation
- https://github.com/vsouza/awesome-ios
- https://github.com/matteocrippa/awesome-swift
- http://online.swiftplayground.run/