**100 days of Swift**

**Variables**

Swift is a safe variable language, which means that each var must have one specific type.

To interpolate variables, we can do **var** str = "Pi is equal to \(pi)" , where **var** pi == 3.141. To make variables constants, use **let.**

We can be explicit about our typing, with either **var** or **let**, do the following: **let** album: String = "Reputation" or **let** height: Double = 1.78.

It is preferable to use constants **(let)** as much as possible.

Floats occupy lesser space than Doubles.

**Complex data types**

Arrays in Swift are zero based. To be explicit in its declaration use **let** beatles = [String] [“John”, “Paul”, “George”].

Sets are unordered and cannot contain duplicates, whereas arrays retain their order and can contain duplicates. Sets offer O(1) access time to any element **let** colors = Set [“red", "blue", "green"]

Tuples have three characteristics that make them special: You cannot add or remove items from a tuple; you can’t change the type of items on a tuple; and you need to access them by using numerical positions or naming them **var** person = (name: "Taylor Swift", age: 50)

and access their values by person.0 or person.first. Tuples can also be created as **var** fibonacci = (1, 1, 2, 3, 5, 8).

Dictionaries work just the same as in other languages, they are just declared a bit different:

**let** heights = [

"Taylor Swift": 1.78,

"Justin Bieber": 1.90

]

Default dictionaries also exist here, as in python, just add the default keyword as in heights ["Invalid Invalid", default: -1 ]. This is useful since we can use it to add a special return value, that is not **nil.** We can create an empty dictionary as **var** teams = [String: String](), empty arrays as **var** results = [Int]() and empty sets as **var** words = Set<String>().

An enum is a set of named values, struct is structured data type, and of course a class allows us to create objects with all POO related stuff, use enums like

**enum** Result {

**case** success

**case** failure

}

**let** result1 = Result.success

Enums can also store associated values attached to each case { **case** running(destination: String) } which can be filled used as **let** talking = Activity.talking(volume: 10).

Enums can have an associated raw value, just change the declaration to **enum** Planet: Int {} and fill the values as **case** mercury = 1, or let Swift do it automatically. To get a value from this enum, call it as **let** earth = Planet(rawValue: 2) these are useful for example, to parse information to a server in a way that is readable.

Raw values and associated values are similar, but raw values can be determined automatically and are limited to one raw value per case.

**Operators and conditions**

Swift won’t be able to add an Int variable to a Double variable as in many other languages, this happens because Swift likes to be extremely predictable. Doubles and Ints in Swift take the same amount of space in memory.

Swift supports operator overloading, which is a way to define what an operator does depending on the values it holds. We can make our own Enums comparable as follows:

**enum** Sizes: Comparable {

**case** small

**case** medium

**case** large

} and use them as **let** first = Sizes.small and **let** second = Sizes.large to compare them as print(first < second) where we will get true as a result from, since small appears before large.

Ifs in Swift can be used with or without parenthesis **if** firstCard + secondCard == 21. The ternary operator, works with 3 values, and is quite intuitive to use print(firstCard == secondCard ? "Cards are the same" : "Cards are different"), (compare ? trueResult : falseResult).

Switch statements are structured as

**switch** weather {

**case** "rain":

print("Bring an umbrella")

**case** "sunny":

print("Wear sunscreen")

**default**:

print("Enjoy your day!")

} where all break statements are already there by default, if we want to ignore them, use the **fallthrough** keyword. Switchs are recommended for pattern matching.

Swift gives us two ways of making ranges: the **..<** and **…** operators. The half-open range operator, **..<**, creates ranges up to but excluding the final value, and the closed range operator, **…**, creates ranges up to and including the final value. That is the only difference. “1 to 4” means 1, 2, and 3, but “1 through 4” means 1, 2, 3, and 4. We can print arrays as print(beatles[1...])

**Loops**

For and whiles looks have the same syntax that Python, but with { } for the body. You can declare counts outside **let** range = 1...3 and use them as **for** num **in** range. Do-whiles are also available on Swift, in the form of

**repeat** {

print(number)

number += 1

} **while** number <= 20 it is useful to use it to DRY (don’t repeat yourself).

To exit multiple loops at the same time, just put a label on the outer loop as outerLoop: **for** i **in** 1...10 and when breaking, use **break** outerLoop. To skip the current iteration, use the **continue** keyword.

When doing for loops, the value is assigned to a temporary constant, which cannot be changed. In while loops, you cannot use just a var if it is not Boolean, you must put a comparator.

**Functions**

To declare a function in Swift do:

**func** square(number: Int) -> Int {

**return** number \* number // or just number \* number

} where we omit the return statement if everything is in a single expression.

To return multiple values from a function we can use a tuple as in:

**func** getUser() -> (first: String, last: String) {

(first: "Taylor", last: "Swift")

}

Swift lets us provide two names for each parameter: one to be used externally when calling the function, and one to be used internally inside the function, for example in **func** sayHello(to name: String) we would use to externally and name internally.

To omit parameters label, put an underscore before as in: **func** greet(**\_** person: String){}

For default/optional parameters **func** greet(**\_** person: String, nicely: Bool = **true**) {}

Variadic functions accept any number of parameters, declare them as **func** square(numbers: Int...) and call it as square(numbers: 1, 2, 3, 4, 5)

If we want to validate the input of a function before processing, we can use the **throw** keyword. Declare it as **func** checkPassword(**\_** password: String) **throws** -> Bool { } and call it with the following considerations: **do** starts a section of code that might cause problems, **try** is used before every function that might throw an error and **catch** will handle errors gracefully.

All parameters passed into a Swift function are constants, so you can’t change them. If you want, you can pass in one or more parameters as inout, which means they can be changed inside your function, and those changes reflect in the original value outside the function (as passing by reference). Declare them as **func** doubleInPlace(number: **inout** Int){ } and call them as doubleInPlace(number: &num).

**Closures**

Closures gives us the capacity of storing procedures into a variable, they are useful for situations we say: “here’s some work I want you to do at some point, but not necessarily now.”, as in functional programming. Unlike functions, closures put their parameters inside the opening brace. Closures can accept parameters, list them inside a parenthesis within the brackets. Remember closures cannot use external parameters. Do not forget about the keyword **in**. Closures also return values, just write them before the **in** word.

**let** driving = { (place: String ) -> String **in**

**return** "I am driving my car to \(place)"

}

Closures can be used as parameters in functions, which basically is passing a function as a parameter, for example the following function would be able to receive a closure that has no parameters and returns nothing, as:

**func** travel(action: () -> Void) {

print("I'm getting ready to go.")

action()

print("I arrived!")

}

If the last parameter of the function is a closure, we can make use of the trailing closure syntax, we would call the above travel function as

travel {

print("I am driving my car")

}

As we said, closures can also accept parameters:

|  |  |
| --- | --- |
| This function receives a closure, that receives a string as a parameter  **func** travel(action: (String) -> Void) {  print("I'm getting ready to go.")  action("London")  print("I arrived!")  } | We then call the function with the trailing closure syntax  travel { (place: String) -> Void **in**  print("I'm going to \(place) in my car")  }  Note that the Void keywork is no needed, is optional for any return type. |

Closures can get shortened quite much, for example, the above travel closure can be reduced to

travel {

print("I'm going to \($0) in my car")

} where we removed the out type, the return keyword and the in parameter (by putting a $0).

You can get closures return from functions, it has a weird syntax because it has two arrows, the first one specifies the function return value and the second one the closure return value. They help answering the I need a function to call, but I don’t know what that function is.

// Read as: func travel returns a closure that accepts a string, and the closure returns void

**func** travel() -> (String) -> Void {

// This is the closure to be returned

**return** {

print("I am going to \($0)")

}

}

**let** result = travel()

result("London")

In the following function, the var counter will be accessible for the closure and will be global, that means, each time the closure gets called, it increases += its past value + 1.

**func** travel() -> (String) -> Void {

**var** counter = 1

**return** {

print("\(counter). I'm going to \($0)")

counter += 1

}

} Remember closures share & save their values within them.

**Structs**

Structs let us create our own data types out of several small types. A tuple is effectively just a struct without a name, like an anonymous struct. There are two properties: stored properties which are just declared vars, while computed properties are assigned on runtime.

**struct** Sport {

**var** name: String

**var** isOlympicSport: Bool

**var** olympicStatus: String {

**if** isOlympicSport {

**return** "\(name) is an Olympic sport"

} **else** {

**return** "\(name) is not an Olympic sport"

}

}

Property observers notify us when a property has changed, declare them as:

**struct** Progress {

**var** task: String // They only work with vars, not lets

**var** amount: Int {

**didSet** {

print("\(task) is now \(amount)% complete")

}

}

} **didSet** will be called each time the value changes. **willSet** is called before a property changes

Structs can have functions inside them, that can use the properties of the struct as desired. Functions inside the structs are called methods. The difference between methods and funcs is that methods belong to a type such as structs, enums and classes, while functions do not.

**struct** Person {

**var** clothes: String {

**willSet** {

updateUI(msg: "I'm changing from \(clothes) to \(newValue)")

}

**didSet** {

updateUI(msg: "I just changed from \(oldValue) to \(clothes)")

}

}

}

Swift won’t let you write methods that change properties unless you specifically request it. That is where we use mutating methods, where you put the mutating vars inside the mutating func.

**struct** Person {

**var** name: String

**mutating** **func** makeAnonymous() {

name = "Anonymous"

}

}

**Initializers**

Memberwise initializers are those that are mandatory, either defined when creating or called when initialized. If you call them on **init** , then you cannot pass that argument by label, but you can pass items to the initializer as **init**(itemHeight: Double, itemWidth: Double) and use those labels on the constructor **let** drawers = Cabinet(itemHeight: 1.4, itemWidth: 1.0)

**struct** User {

**var** username: String

**init**() {

username = "Anonymous"

print("Creating a new user")

}

} As soon as you add a custom initializer for your struct, the default memberwise initializer goes away. Though an extension can be added as:

**extension** User {

**init**() {

**self**.name = "Anonymous"

print("Creating an anonymous employee…")

}

} Which will call the memberwise init if not provided as argument. All vars must be provided an argument before call the initializer. Remember that to reference **self**.name helps us distinguish which vars are ours and which do not.

Swift lets you create properties only when they are needed, which saves compile time, just add the keyword lazy to the property as in **lazy** **var** familyTree = FamilyTree() and family tree will only be initialized when it is first called. Unlike computed properties, the result is stored.

Structs also allow to have static properties and methods, which will be shared by all struct instances. Declare them as **static** **var** guysCount = 0 and call them as Guy.guysCount.

Access control is available in Swift. Just add **private** before the var declaration.

**Classes**

Classes are very similar to structs, but they have a few key important differences. Frist, classes never come with member initializers, which means, you must always create your own initializer.

Classes can extend from other classes, in other words, inherit. Just add them after their name as in Poodle: Dog, you can then call the parents initializers as **super**.init(name: name, breed: "Poodle"). To override functions from inherited classes, just add the **override** **func** makeNoise(){ } in the child class.

If you want to disallow other developers from building their own class based on yours, use **final** keyword before the **class** word. Swift supports inheriting from only one class.

Copies of structs are unique (changing one won’t change another), whereas copies of classes actually point to the same shared data, this is when using the = operator, as in **var** singerCopy = singer. This because classes are reference types and structs are value types. This is good in most situations, i.e., you have a set of users copied all over your app, and if you change a property of one you would want to change all of them.

Classes have destructors, also called, deinitializers, they can be called as **deinit** { ... } This gets run when a class instance gets destroyed, its job is to tell us when a class was destroyed. Structs don’t have deinitializers because each struct has its own copy of its data, so nothing special needs to happen when it is destroyed.

If you have a constant struct with a variable property, it cannot be changed because the struct itself is constant, for classes, that var CAN change, and you do not need the **mutating** keyword, in fact, that keyword is not allowed in classes. To prevent a variable from changing, just declare it as **let.**

**Protocols**

Protocols let us define how structs, classes, and enums ought to work: what methods they should have, and what properties they should have. To declare a protocol just do

**protocol** Identifiable {

**var** id: String { **get** **set** }

} which then can be either applied to a struct as **struct** User: Identifiable or function parameters as **func** displayID(id: Identifiable){ } so Swift can enforce their rules for us. Protocols cannot implement methods, just extensions.

One protocol can inherit from another, and unlike classes, multiple inheritance is supported, call them as **protocol** Doctor: MakesDiagnoses, PrescribesMedicine { }

Extensions allow us to add methods to existing types, e.g., we could add an extension to type so it has a squared() method

**extension** Int {

**func** squared() -> Int {

**return** **self**.**self**

}

} Extensions do not allow stored properties, just computed ones. If extensions have methods that change vars, they must be marked as **mutating**. Protocols affect one data type at a time, extensions can solve that as follows:

**extension** Collection {

**func** summarize() {

print("There are \(count) of us:")

**for** name **in** **self** {

print(name)

}

}

} this extension will affect all data types that have that the Collection protocol (sets, arrays, etc)

Protocol-oriented programming is crafting your code around protocols and protocol extensions.

|  |  |
| --- | --- |
| // extend from the protocol  **extension** Identifiable {  // provide default implementation for identify()  **func** identify() {  print("My ID is \(id).")  }  } | **struct** MyUser: Identifiable {  **var** id: String  }  **let** twostraws = MyUser(id: "twostraws")  twostraws.identify() |

In protocol-oriented programming (POP) we prefer to build functionality by composing protocols (“this new struct conforms to protocols X, Y, and Z”), whereas in object-oriented programming (OOP) we prefer to build functionality through class inheritance.

**Optionals**

Optionals do not hold any value, it just represents the null on swift. Declare them as **var** age: Int? = **nil** it adds the variable the option to be **nil.** To check if an optional is nil, or has a value, you can do **if** **let** unwrapped = age { … } and the value will be assigned to unwrapped. This can also be done with functions **if** **let** username = getUsername() { } , where the function was declared as **func** getUsername() -> String? { … }

We can also use **guard** **let**, which works similar to the **if** **let**, but the key main difference is that you can still use them after the **guard** finishes. See below:

**func** greet(**\_** name: String?) {

**guard** **let** unwrapped = name **else** {

print("You didn't provide a name!")

**return**

}

print("Hello, \(unwrapped)!")

} If it fails to unwrap, is quits the function, loop or condition.

Use **if** **let** if you just want to unwrap some optionals, but prefer **guard** **let** if you’re specifically checking that conditions are correct before continuing.

To force unwrapping, which happens when you are sure a value is not there and convert the value from an optional type to a non-optional type. Use the ! as **let** num = Int(str) to assign a default value to variable if is **nil** do **let** year = birthYear ?? 1. Guard can also be used without the **let**, if no assignation is needed **guard** page >= 1 **else** { }

Implicitly unwrapped optionals might contain a value, or they might be **nil** however, unlike regular optionals you don’t need to unwrap them in order to use them, use them as **let** age: Int! = **nil**

Optional chaining is possible, which basically proceeds if value asked is not **nil** otherwise, it ignores whatever is left **let** beetle = names.first?.uppercased()

There is also an optional **try**? which will halt if the function called threw an error, it is also possible to use **try**! When we know the function won’t fail.

Failable initializers is an initializer that might work or might not work. You can write these in your own structs and classes by using **init?()** rather than **init()** and return nil if something goes wrong, this will allow your return value to be **nil.** And then initialize as **let** person: Person? = Person(id: "abc")

Typecast is basically converting one object into another object type. To do it, call **if** **let** dog = pet **as**? Dog { }

**Capture Lists: Weak, strong and unowned references**

Capture lists come before a closure’s parameter list in your code, and capture values from the environment as either strong, weak, or unowned. We use them mainly to avoid strong reference cycles – aka retain cycles.

Strong capturing means the closure will capture any external values that are inside the closure to make sure they never get destroyed. What it does is basically to preserve properties of a function even after called, so its closure is a safe call. This is used by default.

Weak capturing makes two things: first, weakly captured values aren’t kept alive by the closure, so they might be destroyed and set to **nil**; as a result, these weakly captured values are always optional in swift. See code below.

func sing() -> () -> Void {

let taylor = Singer()

let singing = { [weak taylor] in

taylor?.playSong()

return

}

return singing

}

Unowned capturing allows values to become nil at any point in the future, but you can work with them as if they are always going to be there. The list looks [unowned taylor].

When to use which?

1. If you know for sure your captured value will never go away while the closure has any chance of being called, you can use **unowned**.
2. If you have a strong reference cycle situation – where thing A owns thing B and thing B owns thing A – then one of the two should use **weak** capturing. This should usually be whichever of the two will be destroyed first.
3. If there’s no chance of a strong reference cycle you can use strong capturing.

**Threads**

When you call async() GCD (General Central Dispatch) will work with a system of queues which have different QoS levels, below listed is their importance:

1. User Interactive: this is the highest priority background thread and should be used when you want a background thread to do work that is important to keep your user interface working. This priority will ask the system to dedicate nearly all available CPU time to you to get the job done as quickly as possible.
2. User Initiated: this should be used to execute tasks requested by the user that they are now waiting for in order to continue using your app. It's not as important as user interactive work but it is important because you're keeping the user waiting.
3. The Utility queue: this should be used for long-running tasks that the user is aware of, but not necessarily desperate for now.
4. The Background queue: this is for long-running tasks that the user isn't actively aware of, or at least doesn't care about its progress or when it completes.

There is another way to use GCD, called performSelector(), which has an inBackground: parameter and onMainThread: parameter. You just pass the name of a method to run, and it will do it so wherever you point out. It makes code easier because we don’t want to worry about closure capturing.

**Codable vs NSCoding**

There are three primary differences between the two solutions:

1. The **Codable** system works on both classes and structs. We made **Person** a class because **NSCoding** only works with classes, but if you didn’t care about Objective-C compatibility you could make it a struct and use **Codable** instead.
2. When we implemented **NSCoding** in the previous chapter we had to write **encode()** and **init()** calls ourself. With **Codable** this isn’t needed unless you need more precise control - it does the work for you.
3. When you encode data using **Codable** you can save to the same format that **NSCoding** uses if you want, but a much more pleasant option is JSON – **Codable** reads and writes JSON natively.