**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.

**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** lets you 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).