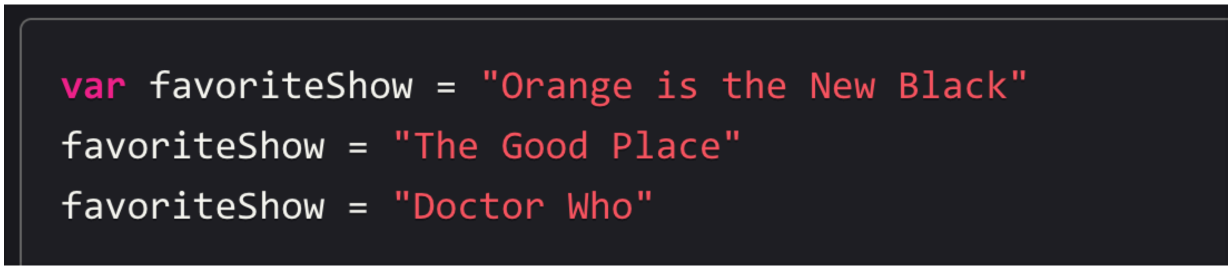
SWIFT INTRO : DATA TYPES

✓ VAR

Once you create a variable using **var**, you can change it as often as you want without using **var** again.

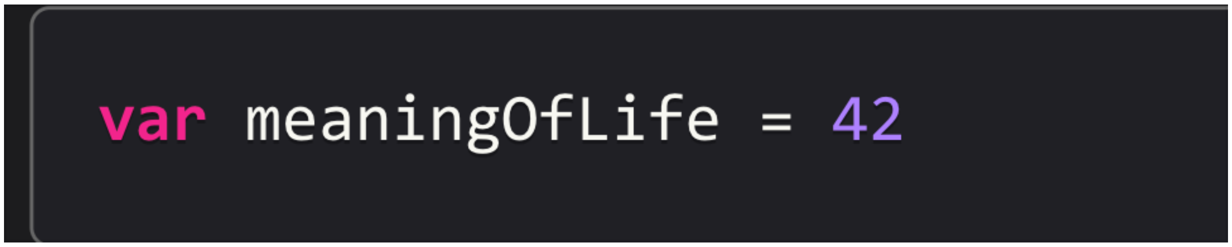


```swift
var favoriteShow = "Orange is the New Black"
favoriteShow = "The Good Place"
favoriteShow = "Doctor Who"
```

"create a new variable called **favoriteShow** and give it the value Orange is the New Black." Lines 2 and 3 don't have **var** in there, so they modify the existing value rather than creating a new variable.

✓ String/Int & Type safe language

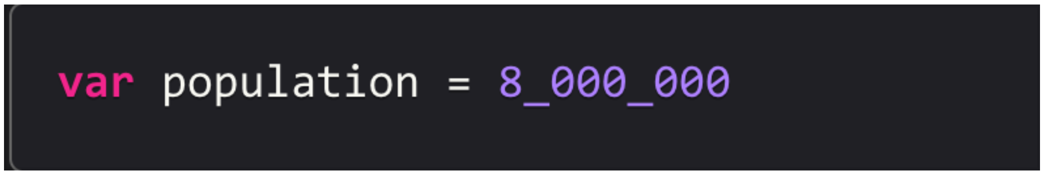
When you create a variable Swift can figure out what type the variable is based on what kind of data you assign to it, and from then on that variable will always have that one specific type.



```swift
var meaningOfLife = 42
```

Because we assigned 42 as the initial value of **meaningOfLife**, Swift will assign it the type *integer* – a whole number. It's a variable, which means we can change its value as often as we need to, but we *can't* change its type: it will always be an integer.

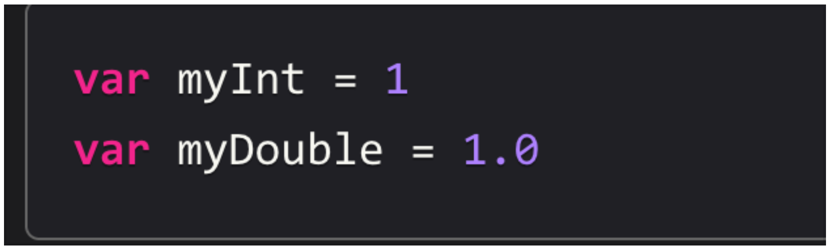
If you have large numbers, Swift lets you use underscores as thousands separators – they don't change the number, but they do make it easier to read. For example:



```swift
var population = 8_000_000
```

✓ Doubles

When creating a numeric variable, Swift decides whether to consider it an integer or a double based on whether you include a decimal point.

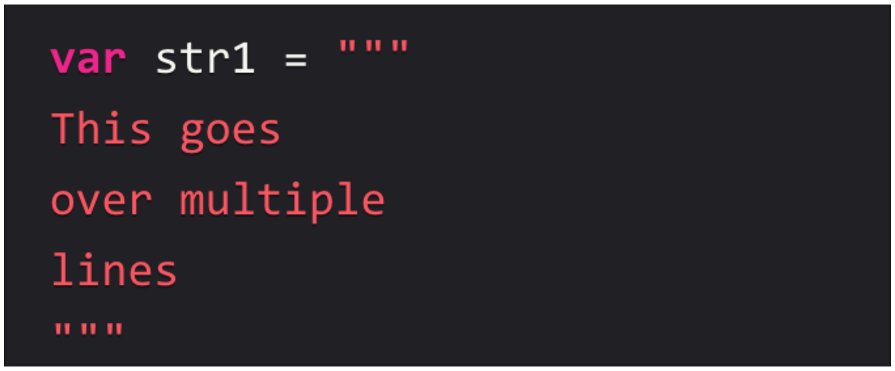


```swift
var myInt = 1
var myDouble = 1.0
```

** we can't write **var total = myInt + myDouble.** It isn't allowed!

✓ Multi-line String

If you want multi-line strings you need slightly different syntax: start and end with three double quote marks, like this:



```swift
var str1 = """
This goes
over multiple
lines
"""
```

NOTE: **the **opening and closing triple must be on their own line by themselves**, but opening and closing line breaks won't be included in your final string.
If you only want multi-line strings to format your code neatly, and you don't want those line breaks to actually be in your string, end each line with a \

```swift
var str2 = """
This goes \
over multiple \
lines
"""
```

String interpolation

String interpolation - a way of injecting custom data into strings at runtime: it replaces one or more parts of a string with data provided by us.
String interpolation allows you to create strings from other variables and constants, placing their values inside your string.

```swift
var city = "Cardiff"
var message = "Welcome to \(city)!"
```

NOTE: Swift is **capable of placing any kind of data inside string interpolation**. The result might not always be useful, but for all of Swift's basic types – strings, integers, Booleans, etc – the results are great.

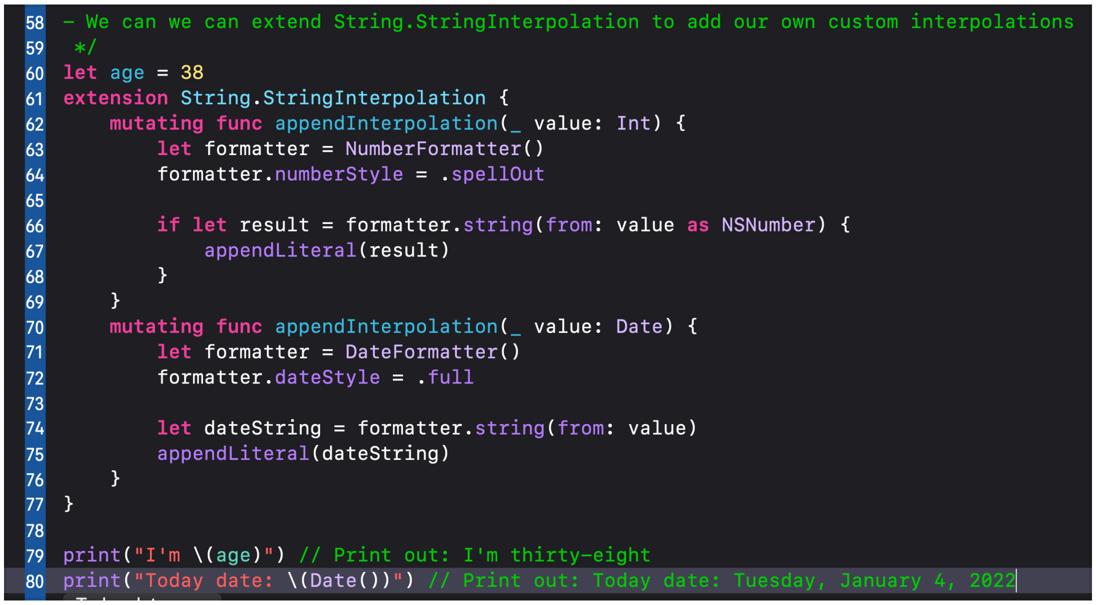
- Swift 2.1 where we gained the ability to use string literals in interpolations, like this:

```swift
print("Hi, \(user ?? "Anonymous")")
```

- We can extend String.StringInterpolation to custom interpolation (swift 5.0+)

(Check xcode sample)

ref link : https://www.hackingwithswift.com/articles/178/super-powered-string-interpolation-in-swift-5-0
https://www.hackingwithswift.com/articles/163/how-to-use-custom-string-interpolation-in-swift



```
- We can we can extend String.StringInterpolation to add our own custom interpolations
 */
let age = 38
extension String.StringInterpolation {
    mutating func appendInterpolation(_ value: Int) {
        let formatter = NumberFormatter()
        formatter.numberStyle = .spellOut

        if let result = formatter.string(from: value as NSNumber) {
            appendLiteral(result)
        }
    }
    mutating func appendInterpolation(_ value: Date) {
        let formatter = DateFormatter()
        formatter.dateStyle = .full

        let dateString = formatter.string(from: value)
        appendLiteral(dateString)
    }
}

print("I'm \(age)") // Print out: I'm thirty-eight
print("Today date: \(Date())") // Print out: Today date: Tuesday, January 4, 2022
```

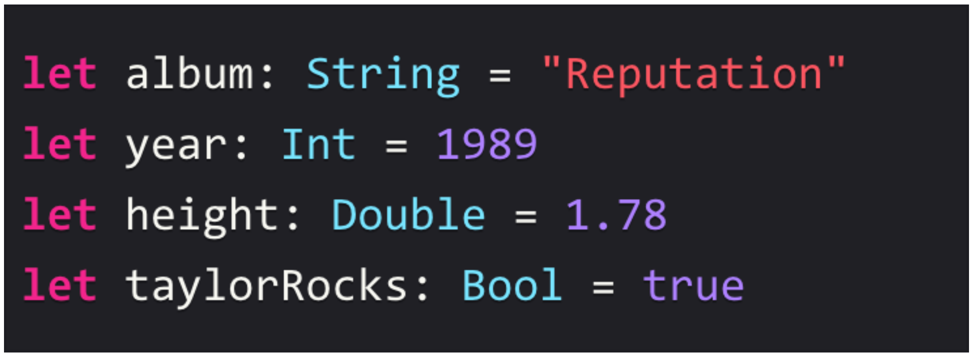
✓ Constants

The **let** keyword creates *constants*, which are values that can be set once and never again

```
let taylor = "swift"
```

✓ Type Annotation

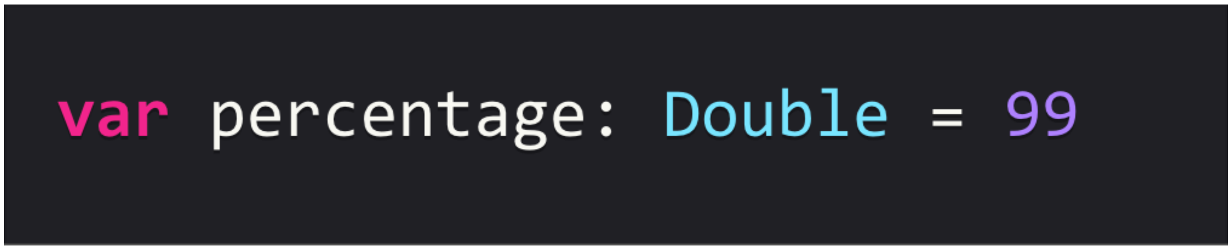
If you want you can be explicit about the type of your data rather than relying on Swift's type inference, like this:



```swift
let album: String = "Reputation"
let year: Int = 1989
let height: Double = 1.78
let taylorRocks: Bool = true
```

*Reason to have type annotation:

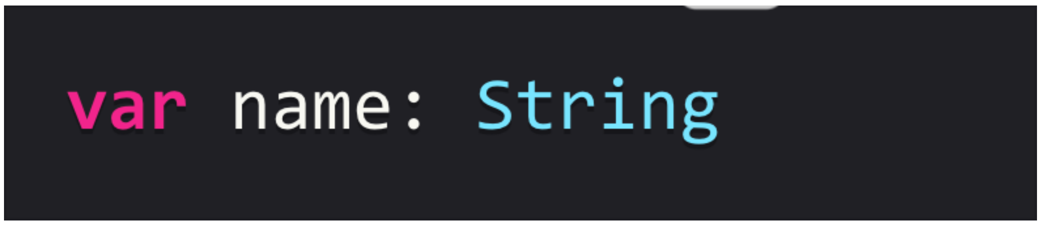
1. In advanced code, Swift can’t know that ahead of time the type, so you’ll need to tell it.
2. Casting value to specific type . EX:



```swift
var percentage: Double = 99
```

That makes **percentage** a double with the value of 99.0. Yes, we have assigned an integer to it, but our type annotation makes it clear that the actual data type we want is double.

3. You want to tell Swift that a variable is going to exist, but you don’t want to set its value just yet.

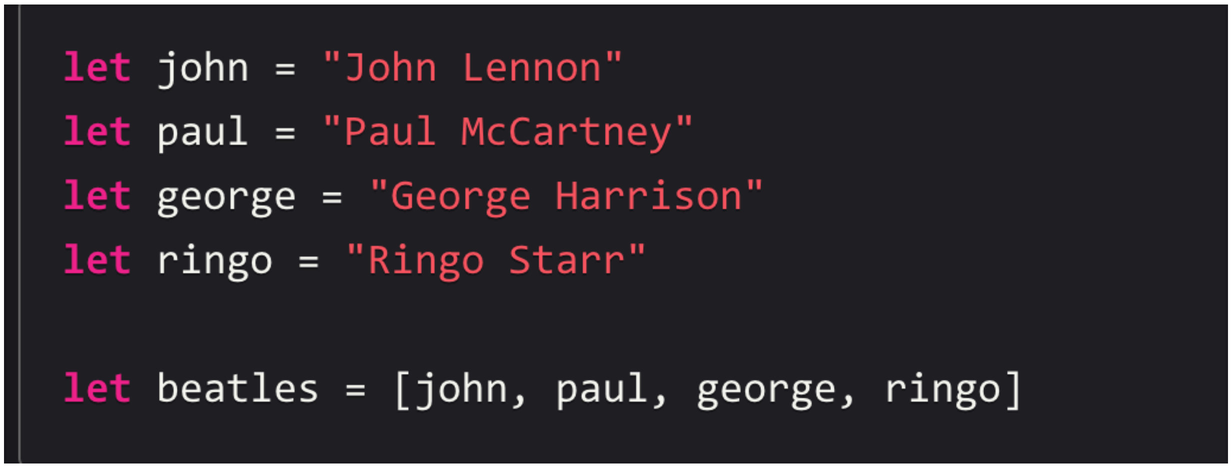


```swift
var name: String
```

You can then assign a string to **name** later on, but you can’t assign a different type because Swift knows it would be invalid.

✓ Array

Array positions count from 0



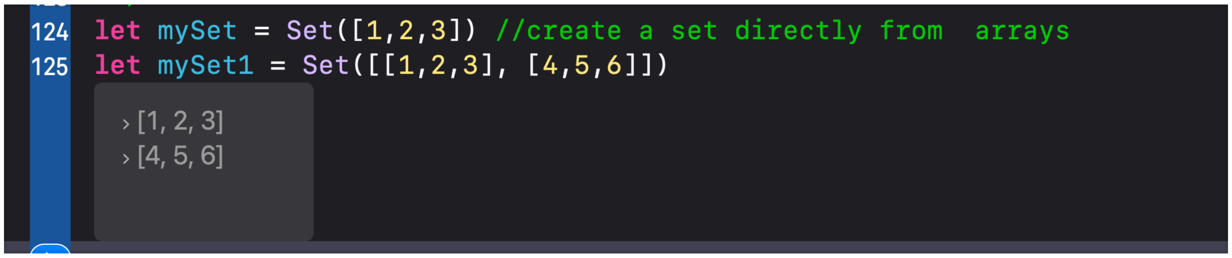
```swift
let john = "John Lennon"
let paul = "Paul McCartney"
let george = "George Harrison"
let ringo = "Ringo Starr"

let beatles = [john, paul, george, ringo]
```

Note: If you're using type annotations, arrays are written in brackets: **[String]**, **[Int]**, **[Double]**, and **[Bool]**.

✓ Set

Sets are collections of values just like arrays, except they have two differences:

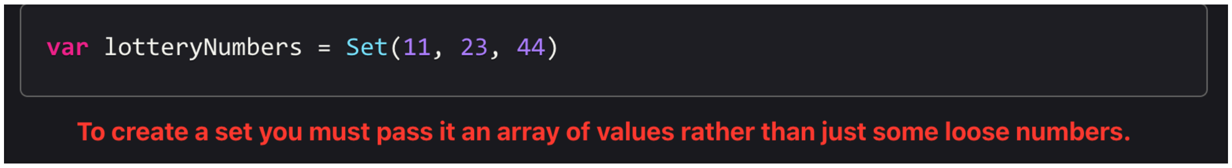
1. Items **aren't stored in any order**; they are stored in what is effectively a random order.
2. Item **can't appear twice in a set**; all items must be **unique.**



```swift
let mySet = Set([1,2,3]) //create a set directly from  arrays
let mySet1 = Set([[1,2,3], [4,5,6]])
```

```
› [1, 2, 3]
› [4, 5, 6]
```

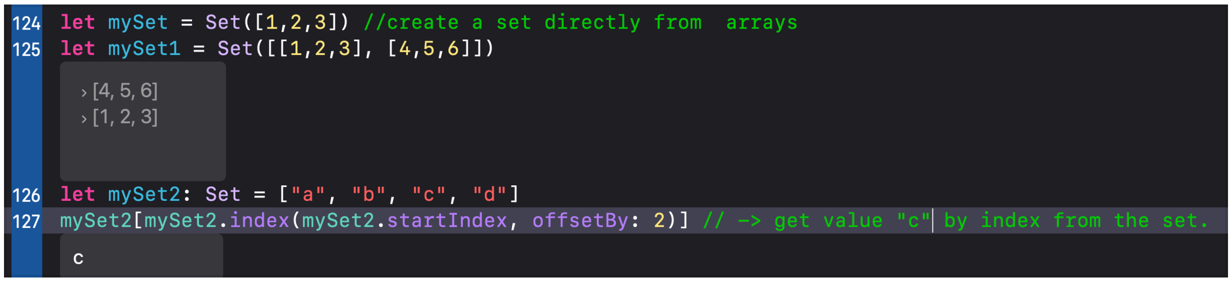
Below is a wrong set defining



```swift
var lotteryNumbers = Set(11, 23, 44)
```

To create a set you must pass it an array of values rather than just some loose numbers.

**NOTE: Because they are unordered, you can't read values from a set using numerical positions like you can with arrays.



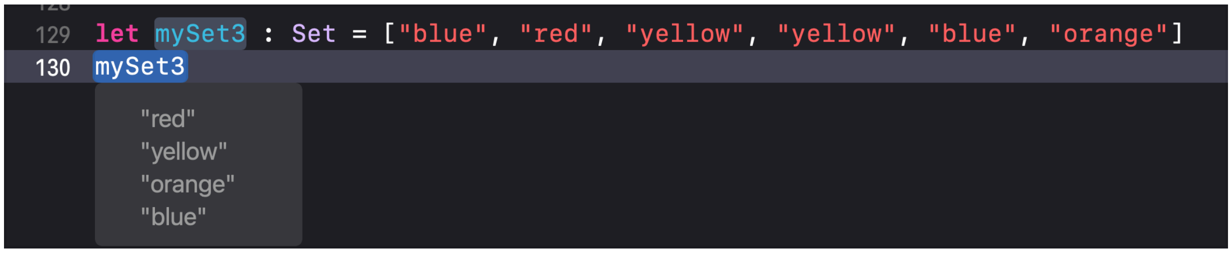
```
let mySet = Set([1,2,3]) //create a set directly from  arrays
let mySet1 = Set([[1,2,3], [4,5,6]])
```

```
› [4, 5, 6]
› [1, 2, 3]
```

```
let mySet2: Set = ["a", "b", "c", "d"]
mySet2[mySet2.index(mySet2.startIndex, offsetBy: 2)] // -> get value "c" by index from the set.
```

```
c
```

If you try to insert a duplicate item into a set, the duplicates get ignored. For example:



```
let mySet3 : Set = ["blue", "red", "yellow", "yellow", "blue", "orange"]
mySet3
```

```
"red"
"yellow"
"orange"
"blue"
```

– Using insert(), remove(),...methods to handle set

```
// using insert method
numbers.insert(32)
```

```
// remove Java from a set
let removedValue = languages.remove("Java")
```

`sorted()`	sorts set elements
`forEach()`	performs the specified actions on each element
`contains()`	searches the specified element in a set
`randomElement()`	returns a random element from the set
`firstIndex()`	returns the index of the given element

– SET VS ARRAY

Similarities	Differences (SET)	(ARRAY)

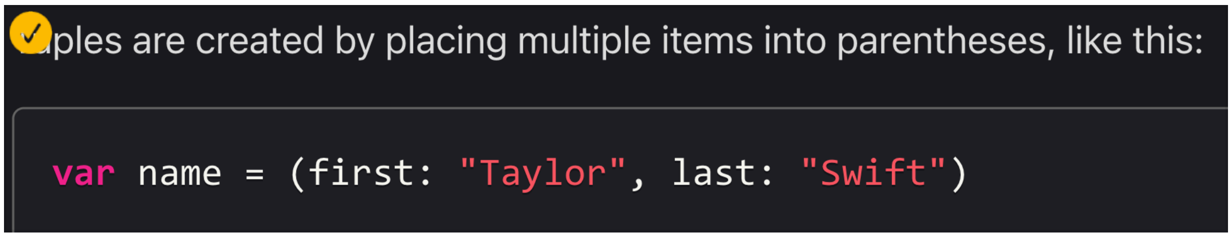
Both are collections of data - holding multiple values inside a single variable	Sets are unordered and cannot contain duplicates	arrays retain their order and *can* contain duplicates

✓ Tupples

Tuples allow you to store several values together in a single value. That might sound like arrays, but tuples are different:

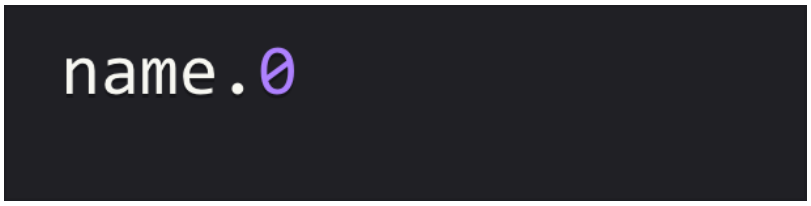
1. You **can’t add or remove items** from a tuple; they are fixed in size.
2. You **can’t change the type** of items in a tuple; they always have the same types they were created with.
3. You can access items in a tuple using numerical positions or by naming them, but Swift won’t let you read numbers or names that don’t exist.

**NOTES: You can change the values inside a tuple after you create it, but not the *types* of values


✓ples are created by placing multiple items into parentheses, like this:

```swift
var name = (first: "Taylor", last: "Swift")
```

- Access tupples using numerical positions starting from 0



```swift
name.0
```

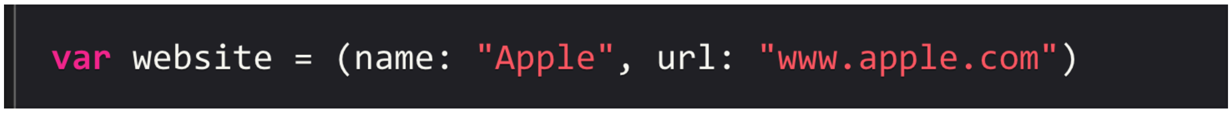
- Access items using their names

```
name.first
```

- TUPPLES VS ARRAY

Similarities	Differences (TUPPLE)	(ARRAY)
Both are collections of data - holding multiple values inside a single variable	tuples hold a *fixed set* of things that can't be changed	arrays can have items added to them indefinitely
	Pros: Each value is specifically created by you, so as well as providing a name you also provide a *type.* *EX: var person = (name: "Paul", age: 40, isMarried: true)*	In array, you can't combine a string, an integer, and a Boolean in a single value

EX:
it's two string items, **name** and **url**, nothing more. Therefore, If we wanted to add to that the date we last visited the site, we couldn't!



```
var website = (name: "Apple", url: "www.apple.com")
```

Because they are specific and named, Swift lets us read them back as **website.name** and **website.url**.

✓ Array , Set, Tupple Use Cases

- TUPPLE: specific, fixed collection of related values where each item has a **precise position or name**

```swift
let address = (house: 555, street: "Taylor Swift Avenue", city: "Nashville")
```

- SET: a collection of **values that must be unique** or you need to be able to check whether a specific item is in there extremely quickly

```swift
let set = Set(["aardvark", "astronaut", "azalea"])
```

- ARRAY: a collection of values that **can contain duplicates**, or the order of your items matters

```swift
let pythons = ["Eric", "Graham", "John", "Michael", "Terry", "Terry"]
```

✓ Dictionaries:

Use a colon to separate the value you want to store (e.g. 1.78) from the identifier you want to store it under (e.g. "Taylor Swift").

```swift
let heights = [
    "Taylor Swift": 1.78,
    "Ed Sheeran": 1.73
]
```

Similarities	Differences (DICTIONARY)	(ARRAY)

Both are collections of data - holding multiple values inside a single variable	dictionaries let us choose a “key” that identifies the item we want to add, whereas arrays just add each item sequentially.	
dictionaries start and end with brackets and each item is separated with a comma.	**Note:** When using type annotations, dictionaries are written in brackets with a colon between your identifier and value types. For example, **[String: Double]** and **[String : String]**.	
	Dictionaries don’t store our items using an index, but instead they optimize the way they store items for fast retrieval.	

	CONS: you can’t be guaranteed that a key in a dictionary exists. This is why reading a value from a dictionary might send back nothing – If you try to read a value from a dictionary using a key that doesn’t exist, Swift will send you back **nil** – nothing at all.	

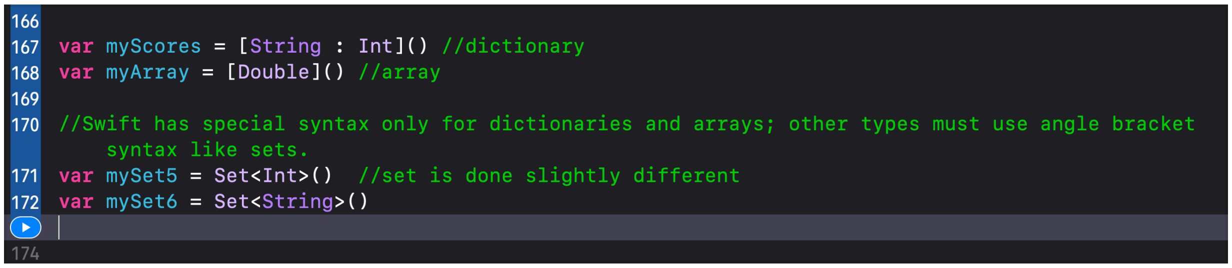
These identifiers are called *keys*, and you can use them to read data back out of the dictionary:

```
heights["Taylor Swift"]
```

– Dictionary Default Value:
We can provide the dictionary with a default value to use if we request a missing key.

```
favoriteIceCream["Charlotte", default: "Unknown"]
```

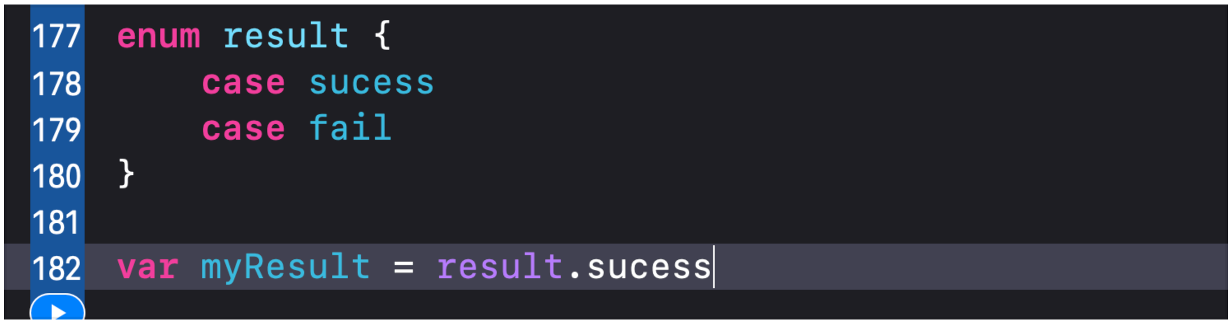
✓ Empty Collections:
Arrays, sets, and dictionaries are called *collections*, because they collect values together in one place.
If you want to **create an *empty* collection** just write its type followed by **opening and closing parentheses.**



```
var myScores = [String : Int]() //dictionary
var myArray = [Double]() //array

//Swift has special syntax only for dictionaries and arrays; other types must use angle bracket
    syntax like sets.
var mySet5 = Set<Int>()  //set is done slightly different
var mySet6 = Set<String>()
```

✅ Enumerations

A way of defining groups of related values in a way that makes them easier to use. This stops you from accidentally using different strings each time.



```
enum result {
    case sucess
    case fail
}

var myResult = result.sucess
```

– Enum Associated Value:

Enums can also store ***associated* values attached to each case**. This lets you attach additional information to your enums so they can represent more nuanced data.

PROS:

Associated values **can be attached to every case, or only some cases.**

Plus, each case **can have as many associated values as you want**, as long as each one has a type.

```swift
enum Activity {
    case bored
    case running(destination: String)
    case talking(topic: String)
    case singing(volume: Int)
}
```

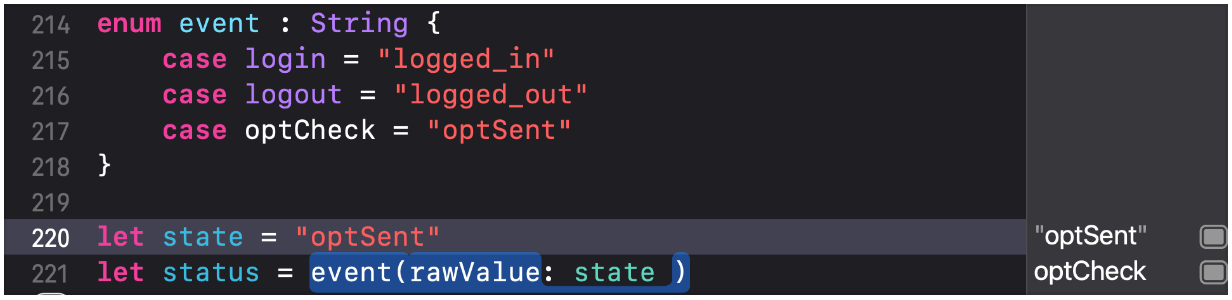
Now we can be more precise – we can say that someone is talking about football:

```swift
let talking = Activity.talking(topic: "football")
```

– Enum raw values:

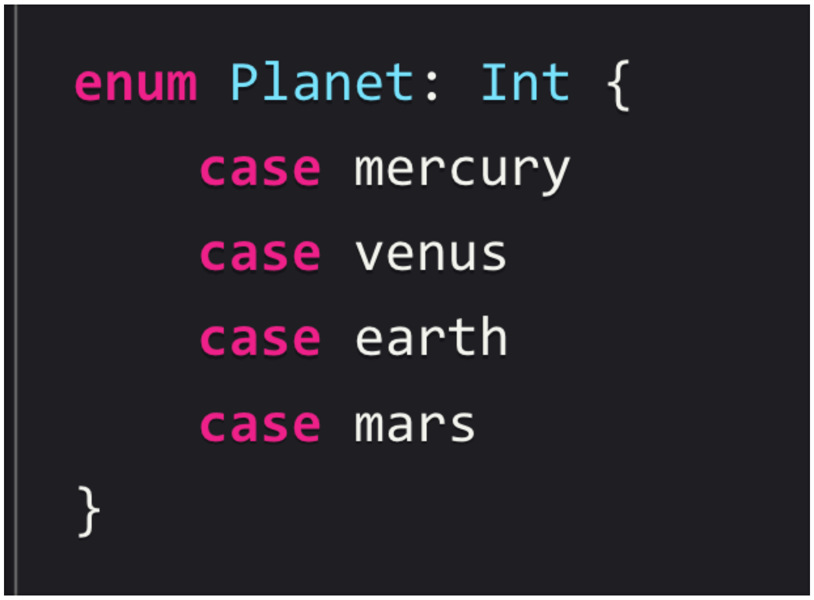
Ref link: https://www.avanderlee.com/swift/enumerations/

You can attach raw values to enums so they can be created from integers or strings



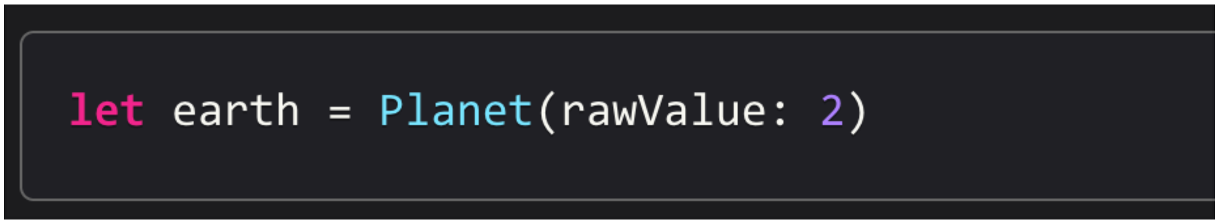
```swift
enum event : String {
    case login = "logged_in"
    case logout = "logged_out"
    case optCheck = "optSent"
}

let state = "optSent"
let status = event(rawValue: state )
```

```
"optSent"
optCheck
```



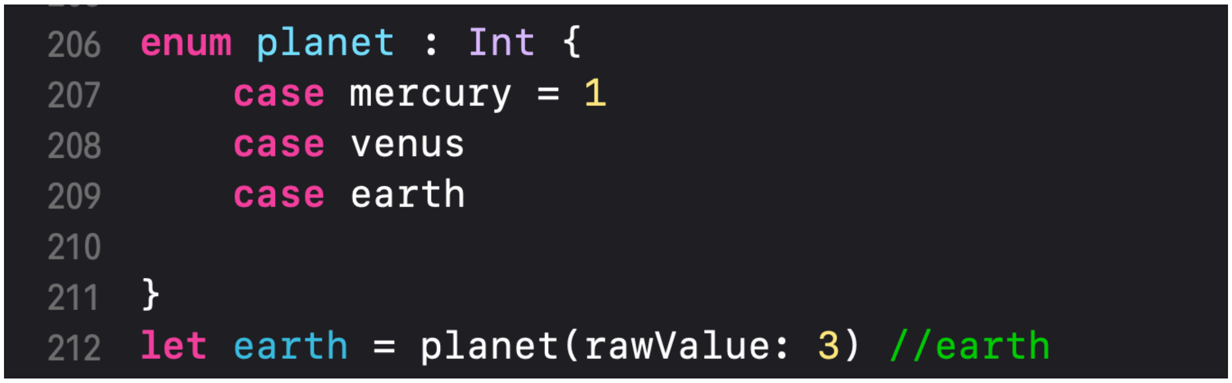
```swift
enum Planet: Int {
     case mercury
     case venus
     case earth
     case mars
}
```

+ Swift will automatically assign each of those a number starting from 0, and you can use that number to create an instance of the appropriate enum case.
For example, **earth** will be given the number 2, so you can write this:



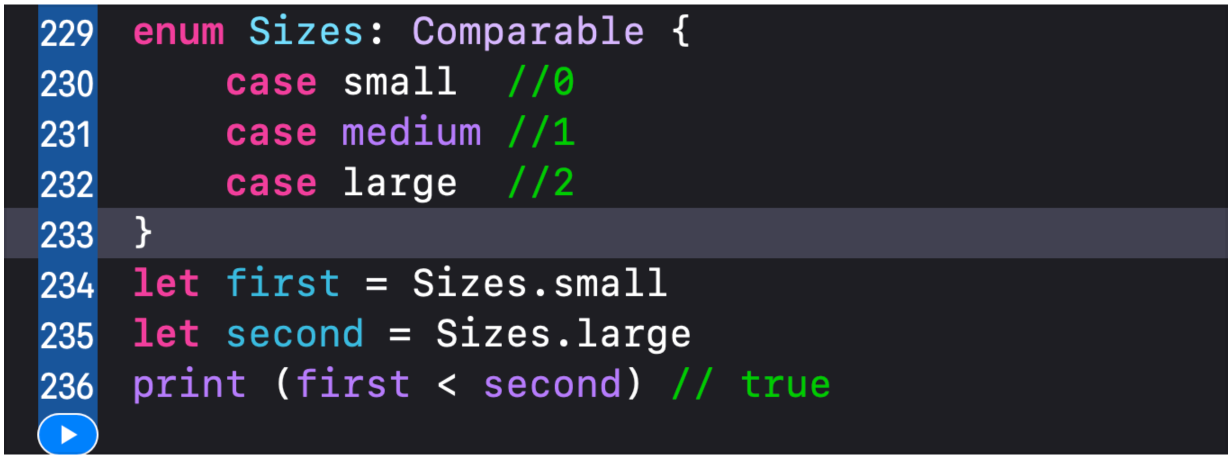
```swift
let earth = Planet(rawValue: 2)
```

+ If you want, you can assign one or more cases a specific value, and Swift will generate the rest.



```swift
enum planet : Int {
    case mercury = 1
    case venus
    case earth

}
let earth = planet(rawValue: 3) //earth
```

+ Comparable enum



```
enum Sizes: Comparable {
     case small  //0
     case medium //1
     case large  //2
}
let first = Sizes.small
let second = Sizes.large
print (first < second) // true
```