A.STEFANCHUK IT STEP

SWIFT INTRO

BRIEF HISTORY

- Swift was developed by Apple to replace old and clumsy Objective-C and released in 2014 year
- Designed to be extremely safe, fast and clean
- Current version - Swift 5.3
- It is open source and widely supported by community

COOL STUFF

- Optionals
- JSON support without any mapping libraries
- Functional programming (map, filter, etc.)
- Closures
- Powerful enums and structs
- Protocol oriented paradigm

GENERAL

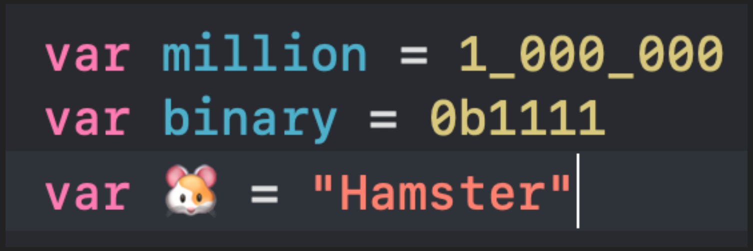
- print(“Hello world”) - is whole program
- print(“Something”) - is our debugger “on minimals”
- Code is written in global scope -> no need for main()
- No semicolons needed at the end of line
- You can experiment with code in Playground

PRIMITIVE TYPES

- Int (+unsigned integer like UInt16)
- Double
- Float
- Bool
- String

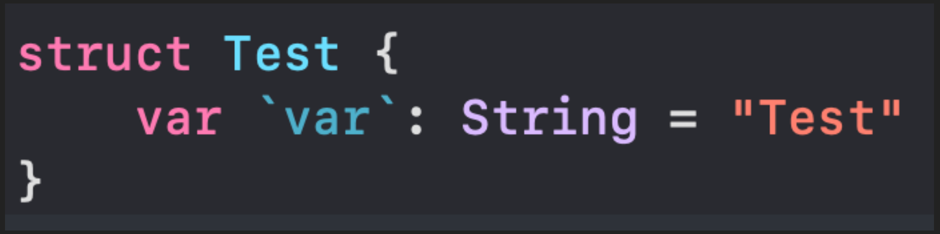
All those types are structs!

JUST BECAUSE WE CAN

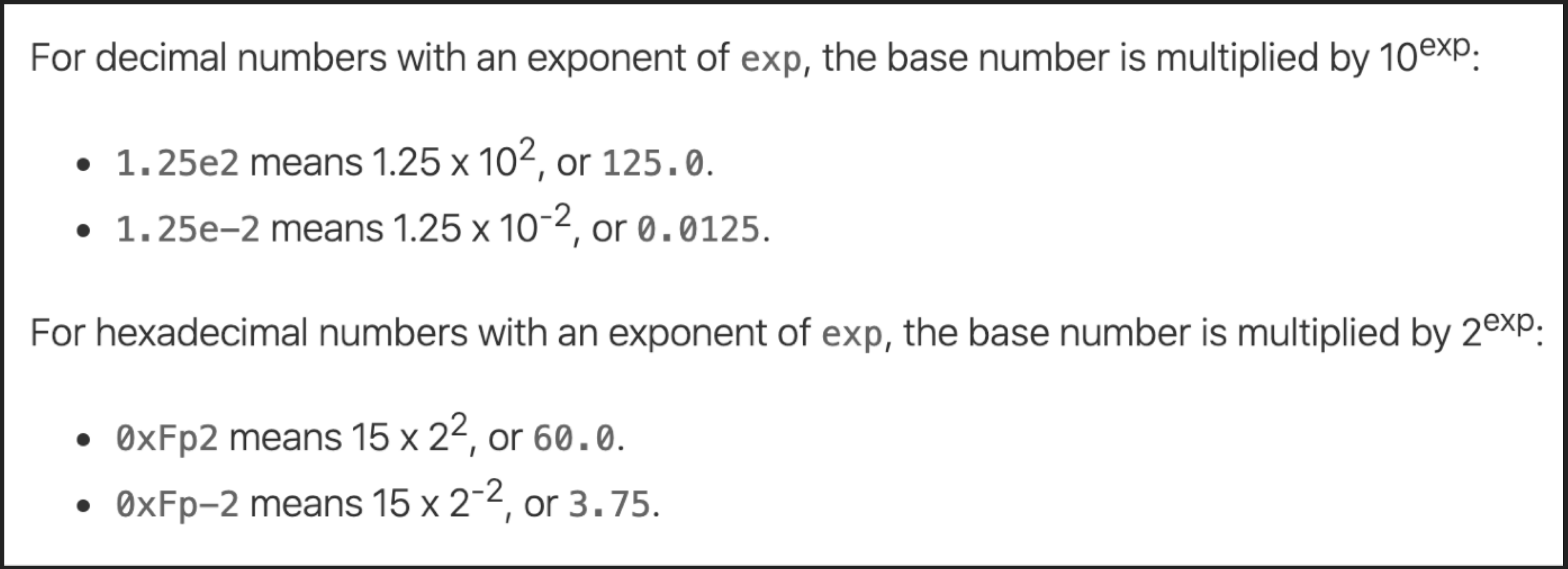


```
var million = 1_000_000
var binary = 0b1111
var 🐹 = "Hamster"
```

Don't use emojis in your code ;)



```
struct Test {
    var `var`: String = "Test"
}
```



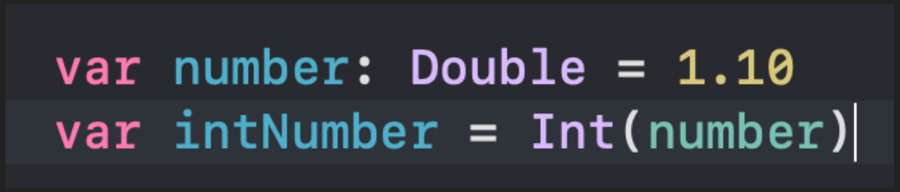
For decimal numbers with an exponent of `exp`, the base number is multiplied by 10^{exp}:

- `1.25e2` means 1.25 x 10^{2}, or `125.0`.
- `1.25e-2` means 1.25 x 10^{-2}, or `0.0125`.

For hexadecimal numbers with an exponent of `exp`, the base number is multiplied by 2^{exp}:

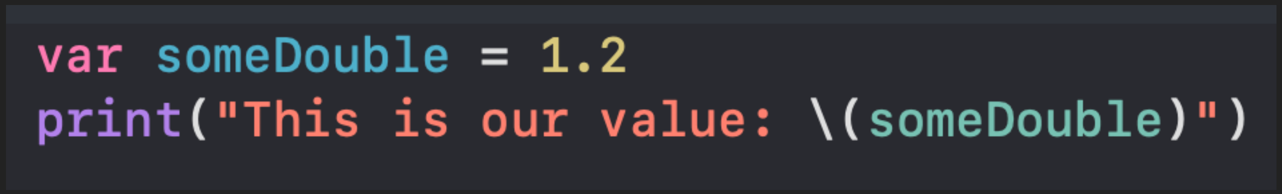
- `0xFp2` means 15 x 2^{2}, or `60.0`.
- `0xFp-2` means 15 x 2^{-2}, or `3.75`.

TYPE CONVERSION



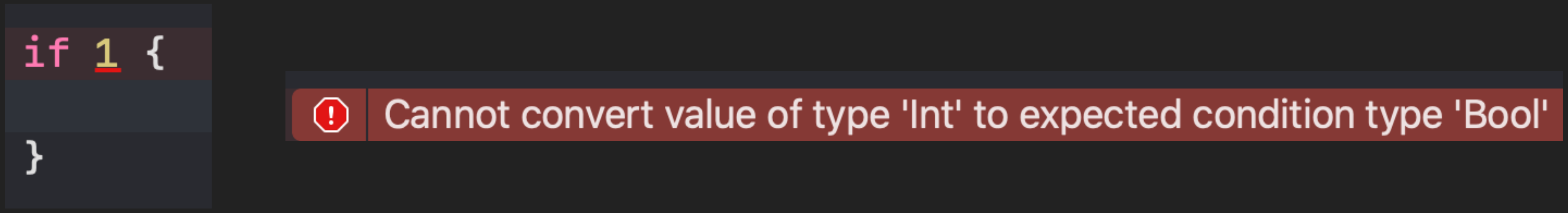
```
var number: Double = 1.10
var intNumber = Int(number)
```

STRING INTERPOLATION



```
var someDouble = 1.2
print("This is our value: \(someDouble)")
```

ONLY BOOL FOR CONDITIONS

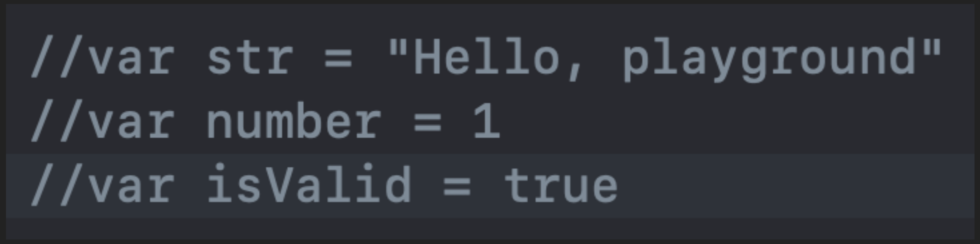


```
if 1 {

}
```

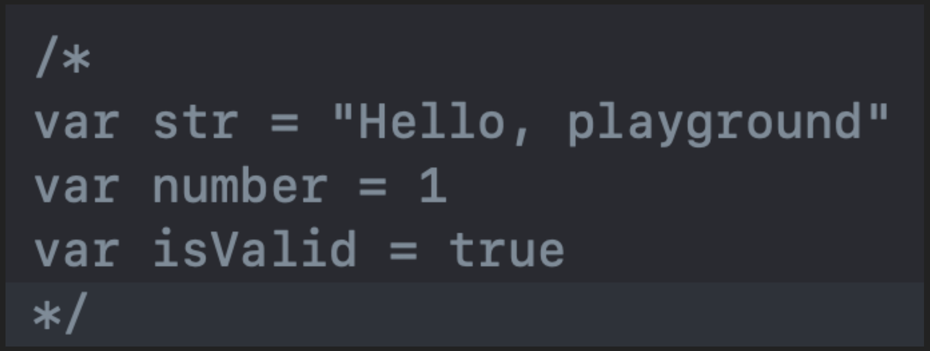
Cannot convert value of type 'Int' to expected condition type 'Bool'

COMMENTS



```
//var str = "Hello, playground"
//var number = 1
//var isValid = true
```

Don't leave commented code ;)

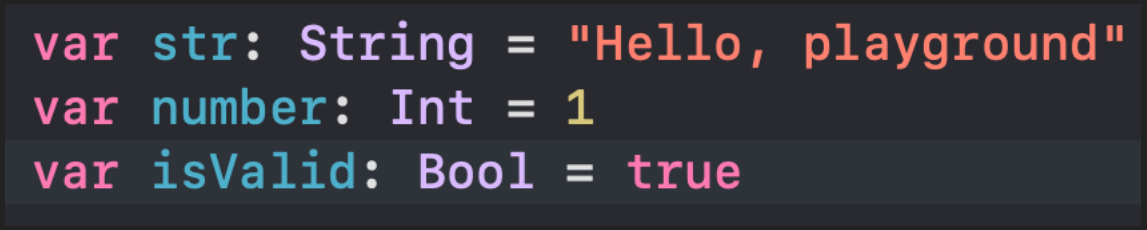


```
/*
var str = "Hello, playground"
var number = 1
var isValid = true
*/
```

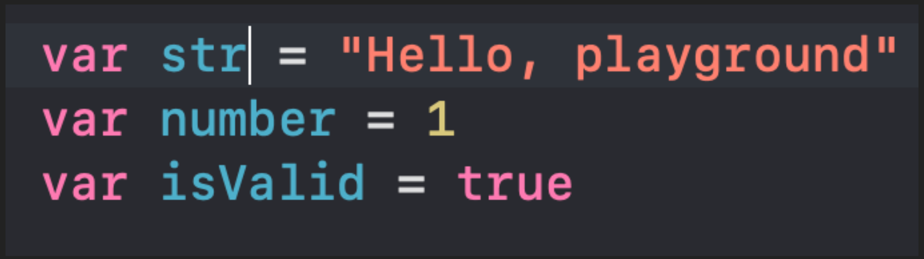
Only if it's documentation

TYPE INFERENCE

- Swift defines all the types at compile time and that’s one of reasons why it is so safe (you don’t get runtime types error, you get them at compile time)



```
var str: String = "Hello, playground"
var number: Int = 1
var isValid: Bool = true
```

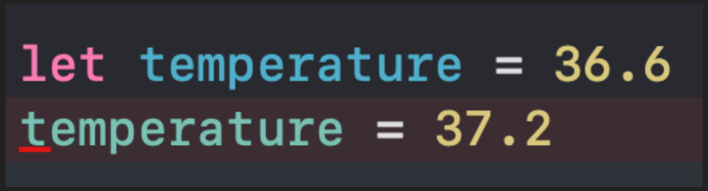


```
var str = "Hello, playground"
var number = 1
var isValid = true
```

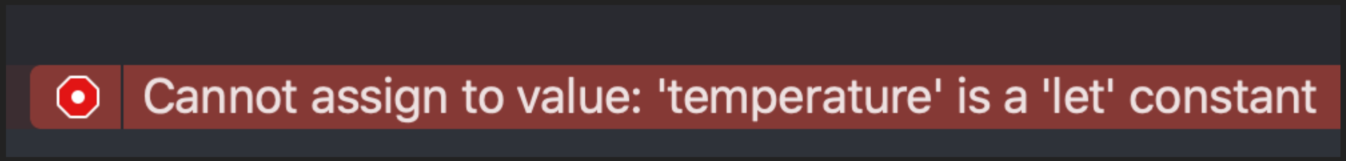
Same result, less code

VAR VS LET

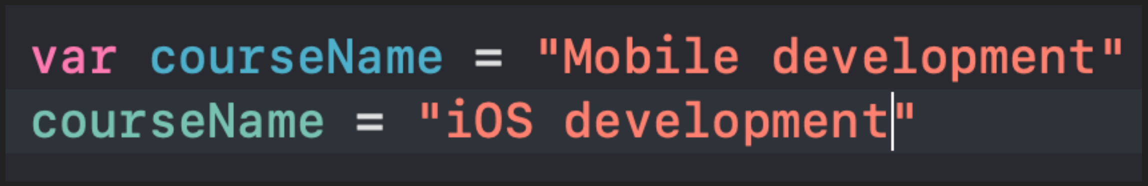
- Everything that can be mutated is variable (var)
- Data that won’t change over time is constant (let)



```
let temperature = 36.6
temperature = 37.2
```



Cannot assign to value: 'temperature' is a 'let' constant



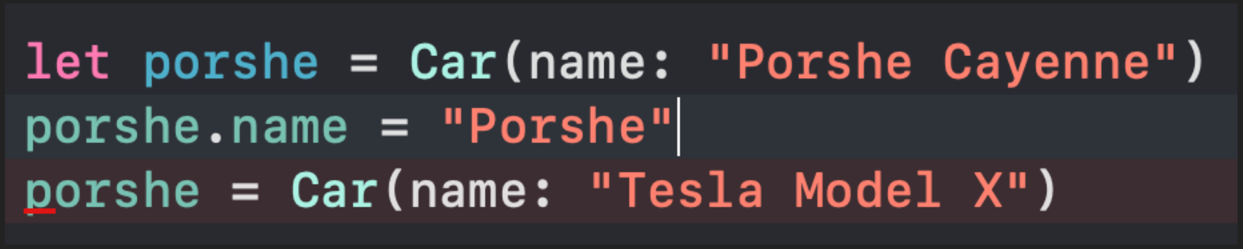
```
var courseName = "Mobile development"
courseName = "iOS development"
```

✓

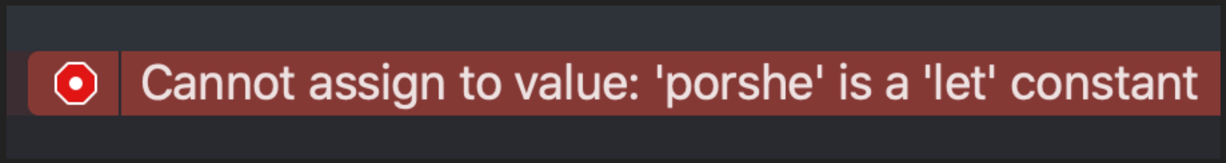
Good practice: make everything let first, then change to var if needed

VAR VS LET (FOR REFERENCE TYPES)

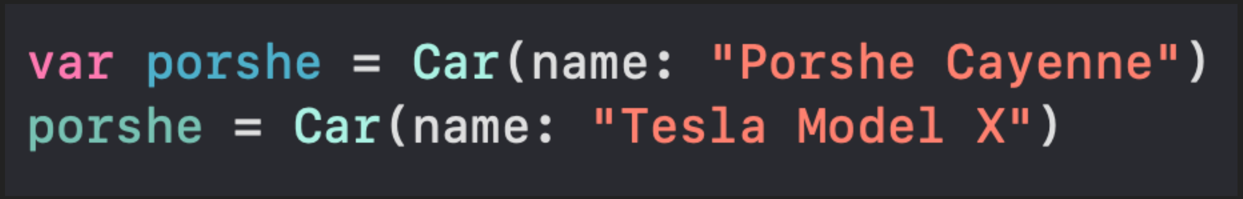
- Works differently for classes as it affects variable or constant reference to object



```
let porshe = Car(name: "Porshe Cayenne")
porshe.name = "Porshe"
porshe = Car(name: "Tesla Model X")
```



Cannot assign to value: 'porshe' is a 'let' constant

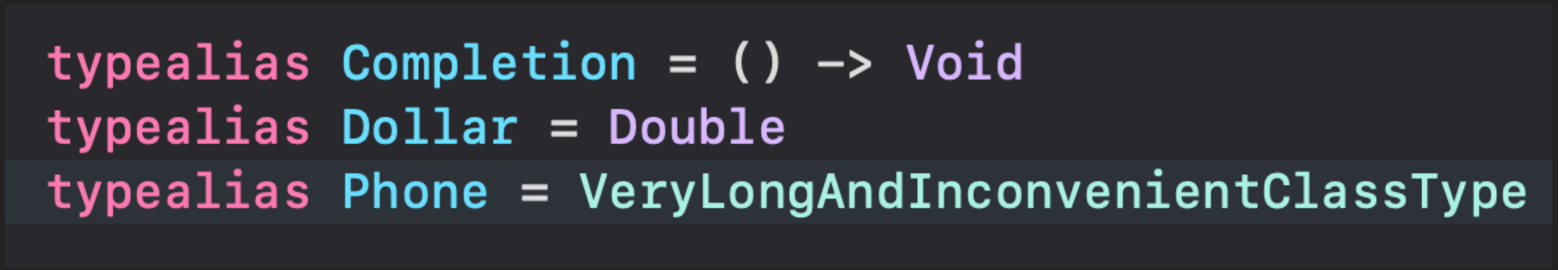


```
var porshe = Car(name: "Porshe Cayenne")
porshe = Car(name: "Tesla Model X")
```

✓

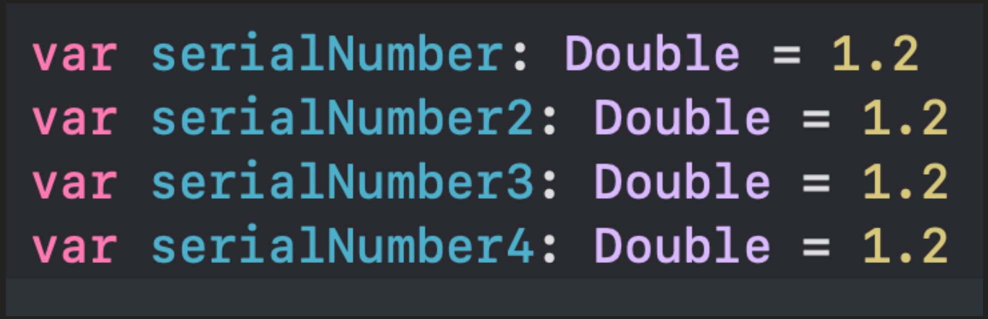
TYPEALIAS

- Need to rename something to match your logic or domain



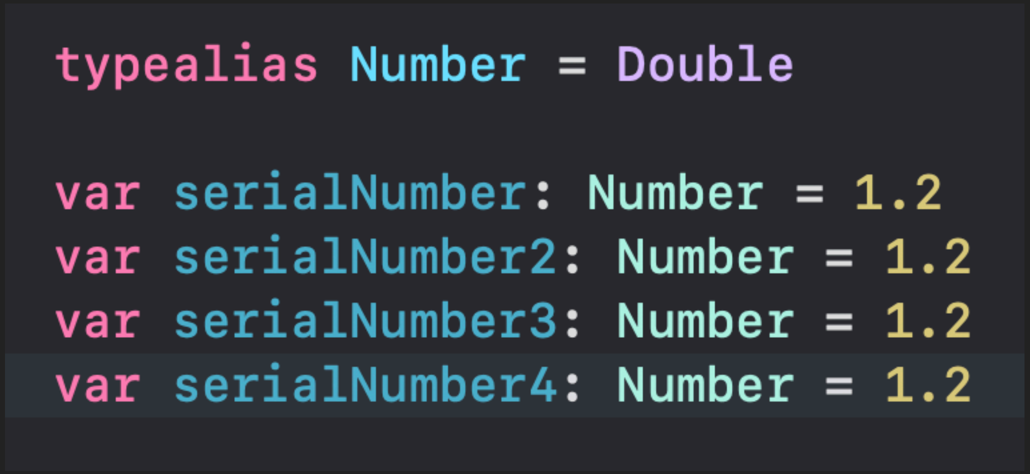
```
typealias Completion = () -> Void
typealias Dollar = Double
typealias Phone = VeryLongAndInconvenientClassType
```

- We are not sure if we need to use Double or Float somewhere



```
var serialNumber: Double = 1.2
var serialNumber2: Double = 1.2
var serialNumber3: Double = 1.2
var serialNumber4: Double = 1.2
```

Change 4 times



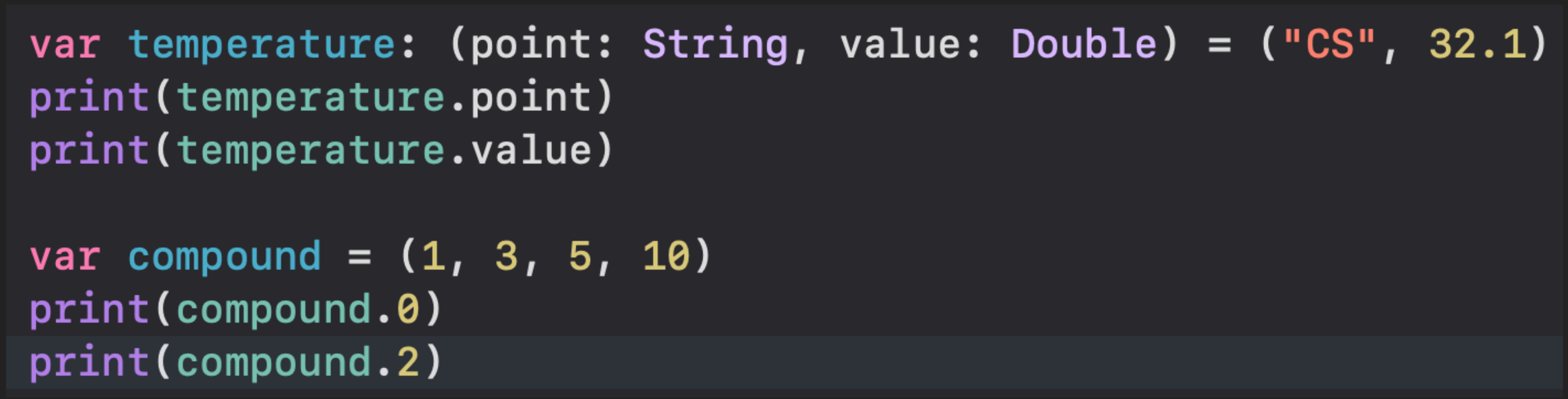
```
typealias Number = Double

var serialNumber: Number = 1.2
var serialNumber2: Number = 1.2
var serialNumber3: Number = 1.2
var serialNumber4: Number = 1.2
```

Change once

TUPLE

- Allow to group more than one value together

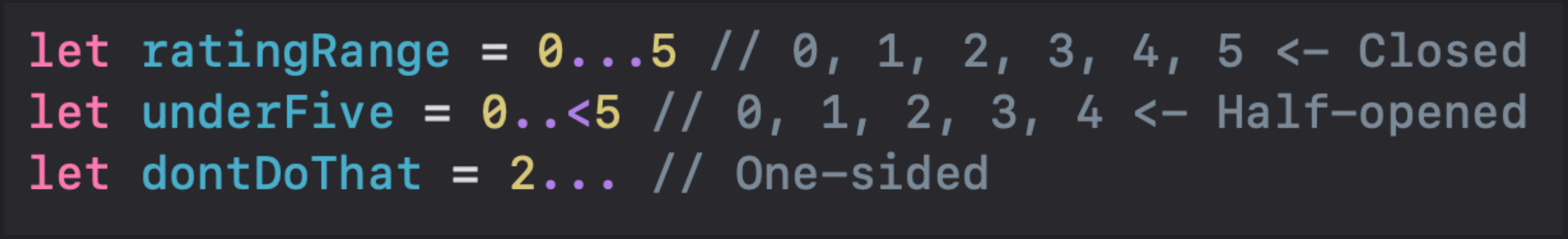


```
var temperature: (point: String, value: Double) = ("CS", 32.1)
print(temperature.point)
print(temperature.value)

var compound = (1, 3, 5, 10)
print(compound.0)
print(compound.2)
```

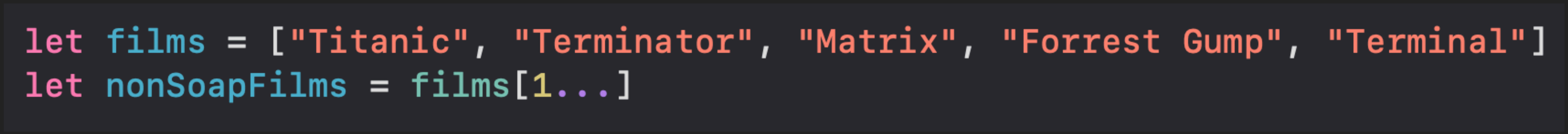
Practical when we need to return more than 1 value from function and there is no need to create separate class for it

RANGES

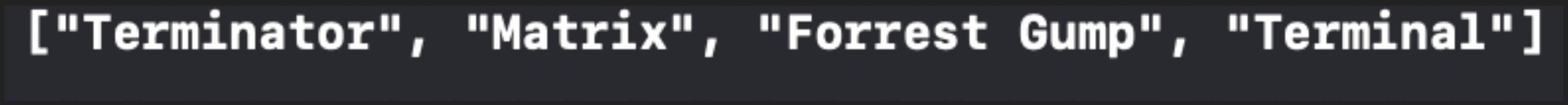


```
let ratingRange = 0...5 // 0, 1, 2, 3, 4, 5 <- Closed
let underFive = 0..<5 // 0, 1, 2, 3, 4 <- Half-opened
let dontDoThat = 2... // One-sided
```

One sided ranges can when need to do with some elements in array and it should be done from some index



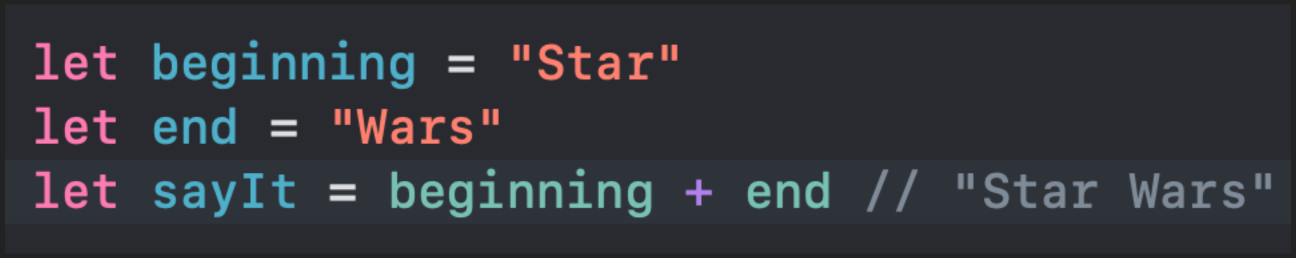
```
let films = ["Titanic", "Terminator", "Matrix", "Forrest Gump", "Terminal"]
let nonSoapFilms = films[1...]
```



```
["Terminator", "Matrix", "Forrest Gump", "Terminal"]
```

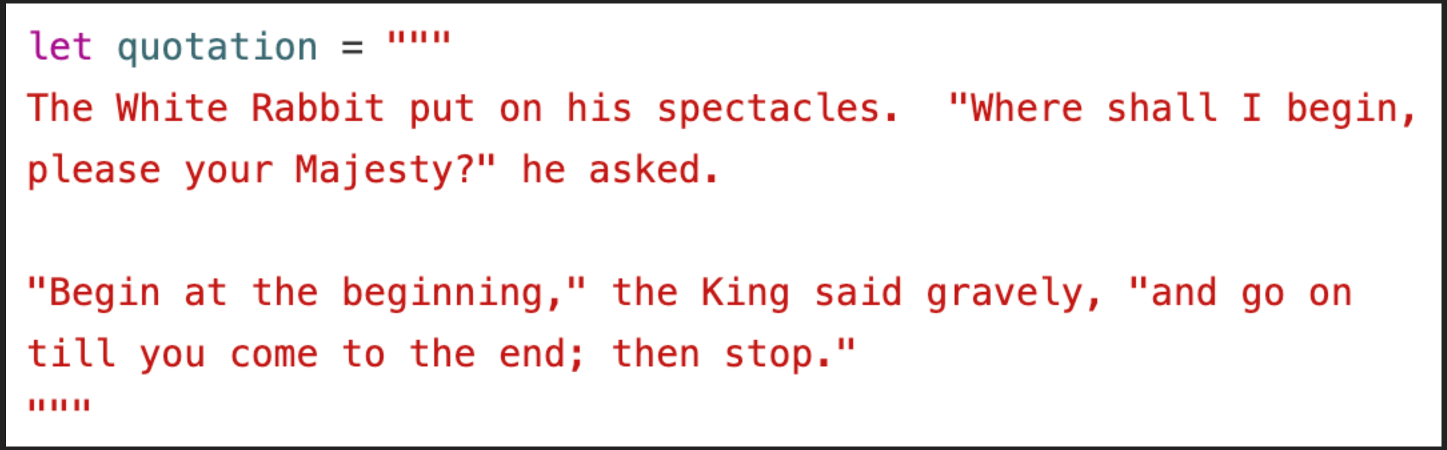
STRINGS

- Collection of characters
- Composed of encoding- independent Unicode characters
- Can be created with just +



```swift
let beginning = "Star"
let end = "Wars"
let sayIt = beginning + end // "Star Wars"
```

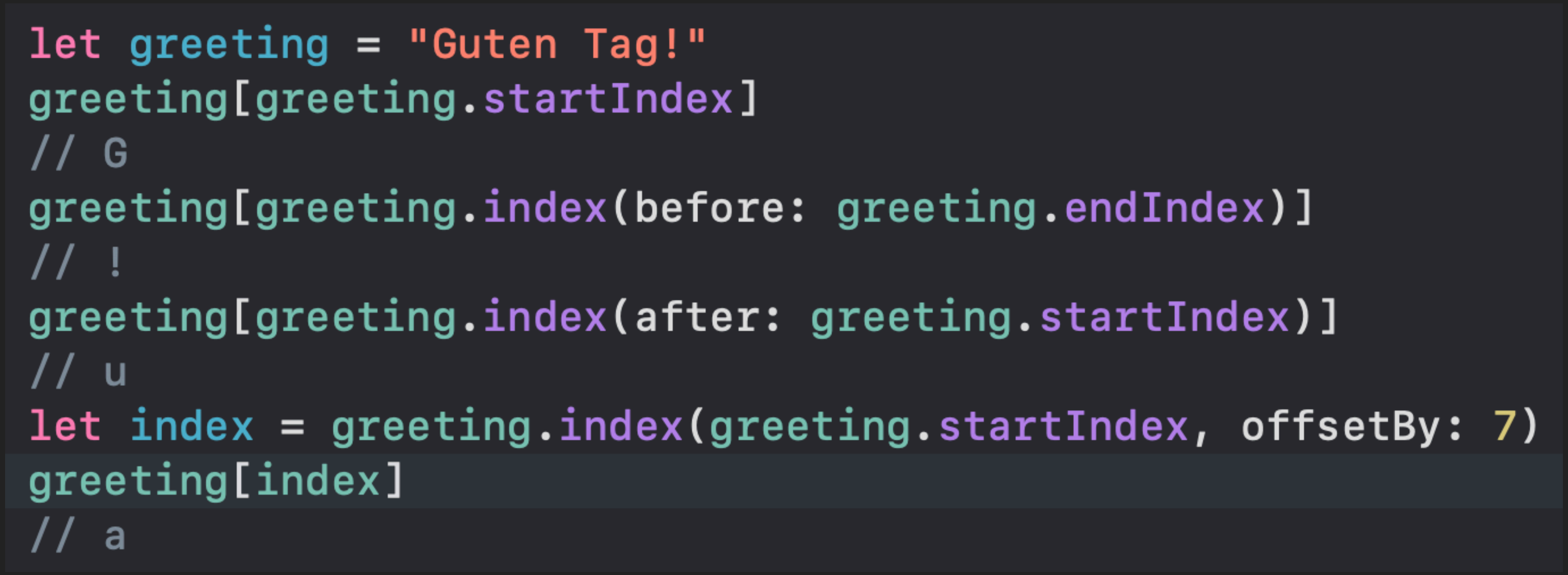
- Multiline string literal
- Formatting is saved



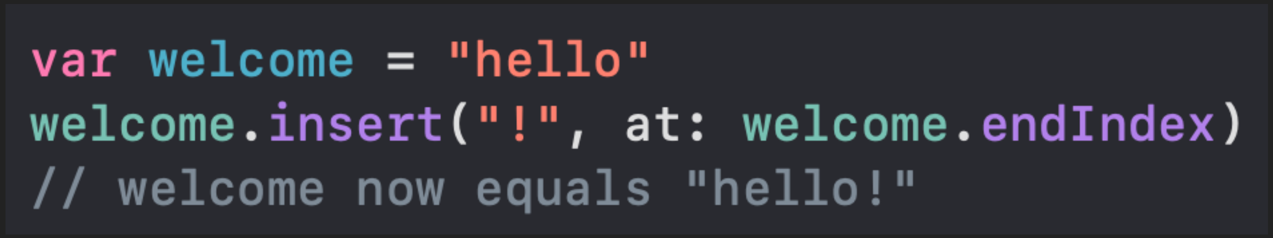
```swift
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
```

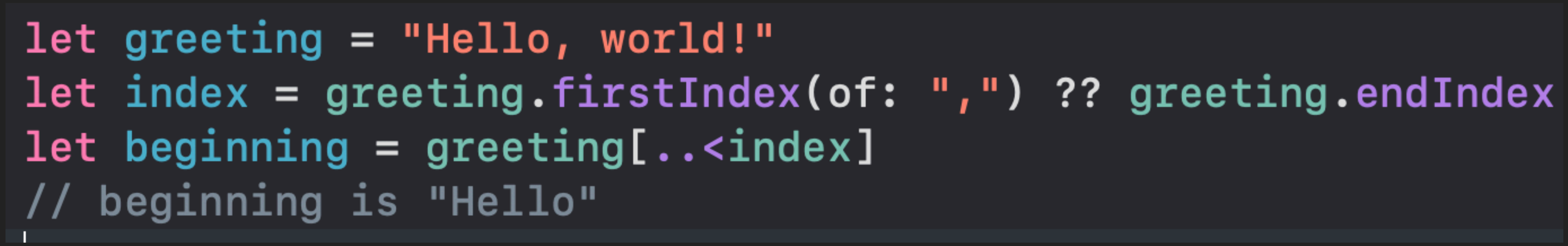
STRINGS - ARE ACCESSED VIA INDEXES



```swift
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
// G
greeting[greeting.index(before: greeting.endIndex)]
// !
greeting[greeting.index(after: greeting.startIndex)]
// u
let index = greeting.index(greeting.startIndex, offsetBy: 7)
greeting[index]
// a
```



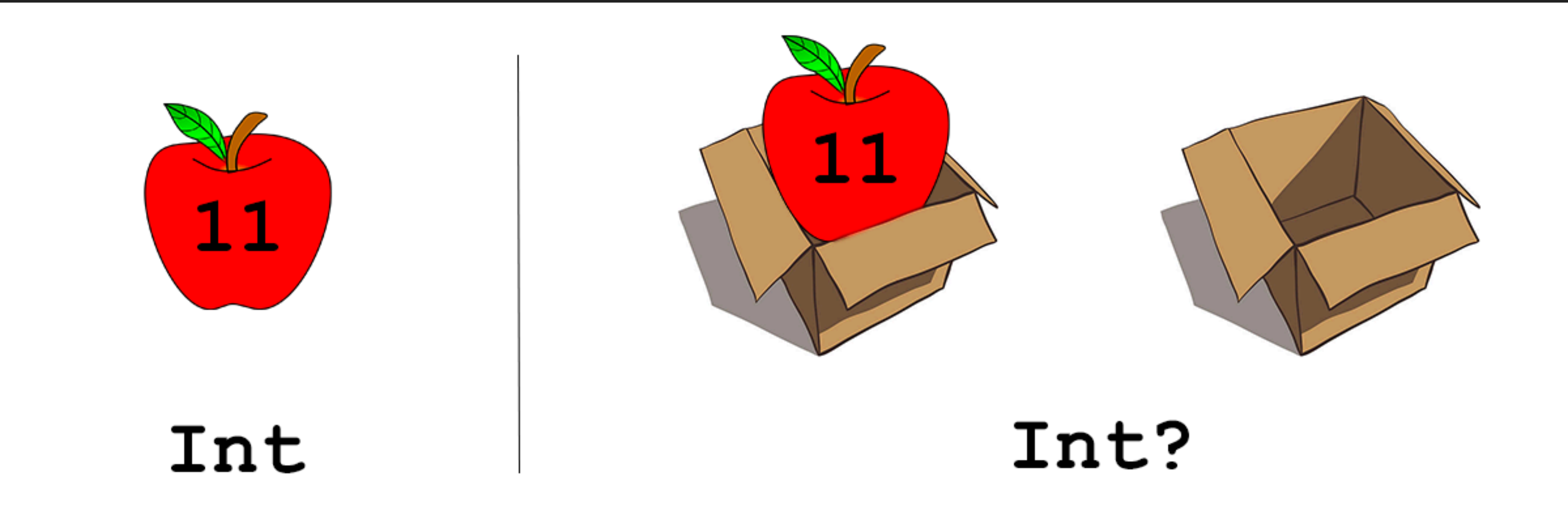
```swift
var welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
// welcome now equals "hello!"
```



```swift
let greeting = "Hello, world!"
let index = greeting.firstIndex(of: ",") ?? greeting.endIndex
let beginning = greeting[..<index]
// beginning is "Hello"
```

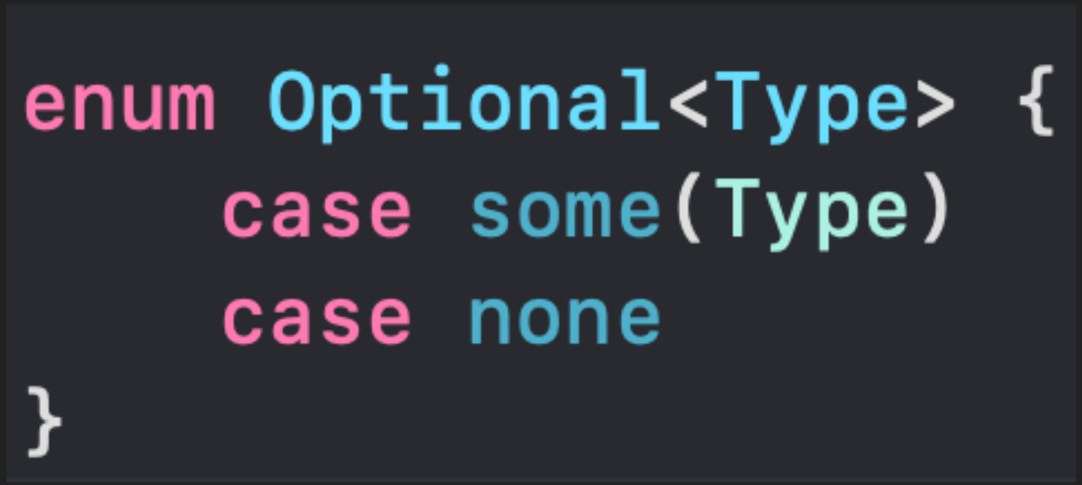
And lot more other functions

OPTIONALS

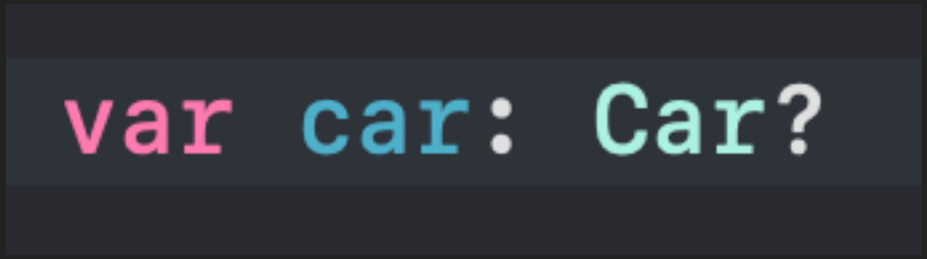


OPTIONALS

- Handles absence of value
- Optional type has some value or don’t have it (nil)
- Every type can be optional by adding ? to it (Int?)



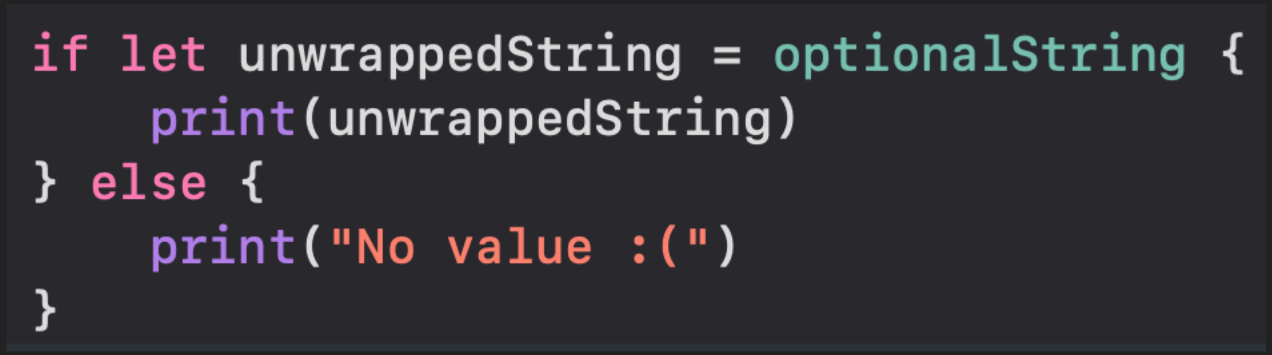
```
enum Optional<Type> {
    case some(Type)
    case none
}
```



```
var car: Car?
```

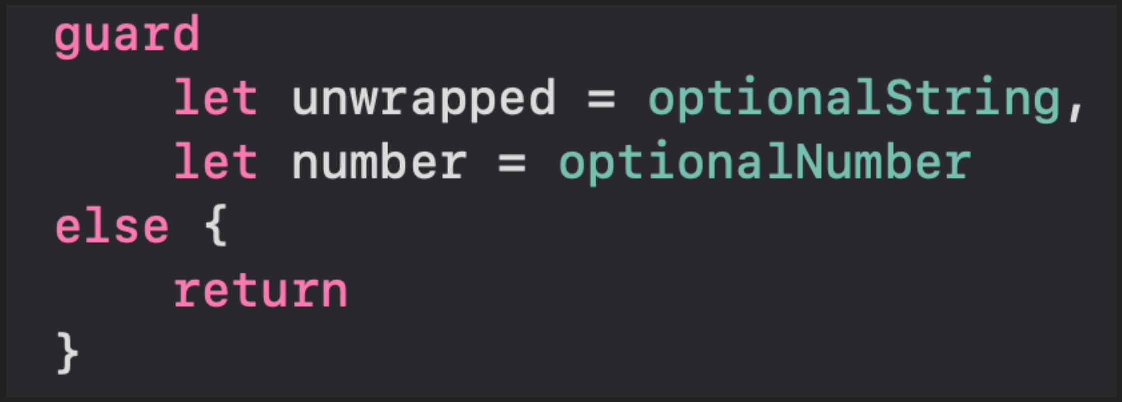
OPTIONALS UNWRAPPING

- Before using optional we should unwrap it:
- 1. if let



```
if let unwrappedString = optionalString {
    print(unwrappedString)
} else {
    print("No value :(")
}
```

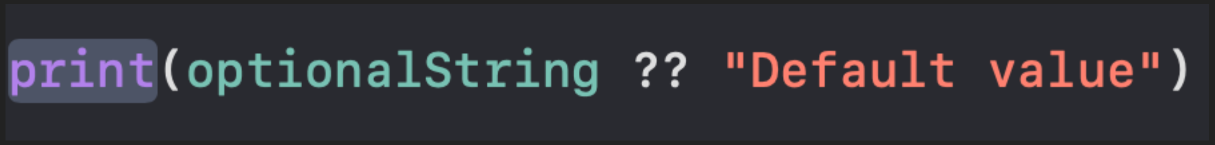
- 2. guard



```
guard
    let unwrapped = optionalString,
    let number = optionalNumber
else {
    return
}
```

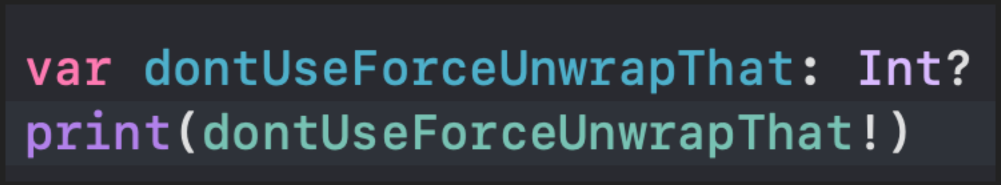
OPTIONALS UNWRAPPING

- 3. nil coalesting operator - ??



```
print(optionalString ?? "Default value")
```

- 4. Force unwrapping

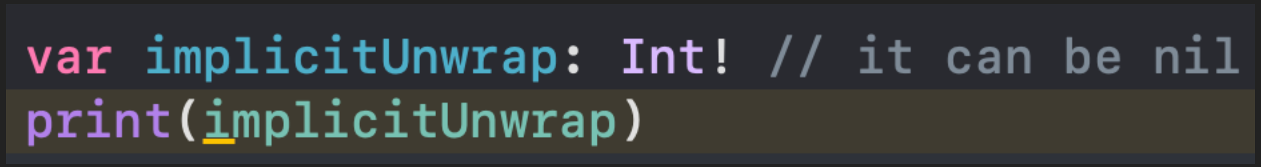


```
var dontUseForceUnwrapThat: Int?
print(dontUseForceUnwrapThat!)
```

You should be really sure
to do that, so avoid this

OPTIONALS UNWRAPPING

- 5. Implicitly unwrapped optional



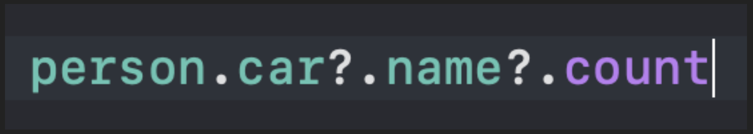
```
var implicitUnwrap: Int! // it can be nil
print(implicitUnwrap)
```

Such value is optional, but
you don’t have to unwrap
it every time

Be careful with that!

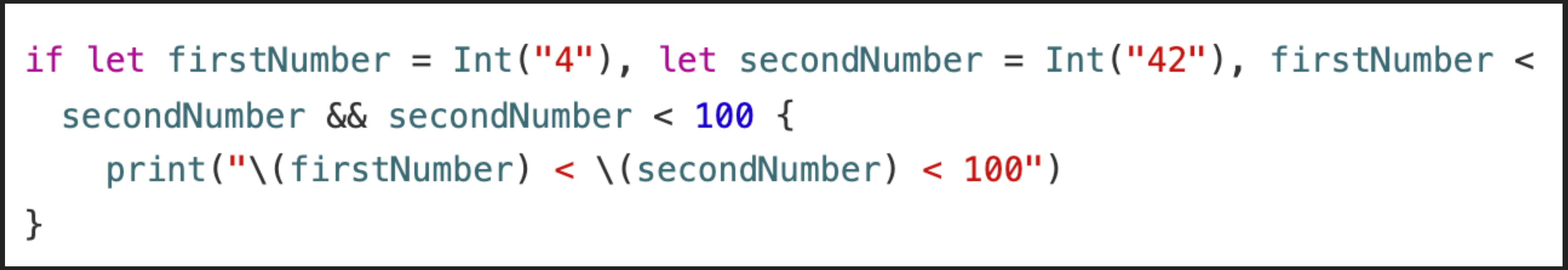
OPTIONAL CHAINING

- 5. Implicitly unwrapped optional



```
person.car?.name?.count
```

- If some value with ? Is nil -> drops operation and goes to next line of code



```
if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber <
  secondNumber && secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
```

- firstNumber & secondNumber already unwrapped

SUMMARY

- Swift has **modern** and **clear** syntax
- **Primitive types** are Int, Bool, Double, Float, String
- **Type inference** - Swift automatically detect types
- **var** is for variables, **let** is for constants
- All **variables** are **initialized** before use
- **Typealias** allows to give another names to types
- **Tuples** can group few values together
- **Optionals** handle absence of value

BEST PRACTICES

- Don’t use emojis in your code
- Don’t write non necessary code - ; () type specification
- Don’t leave commented pieces of code
- Use let first, then change it to var if needed
- Typealiases can make your code nice with right use
- Don’t use !(force unwrap) with optionals to prevent crashes