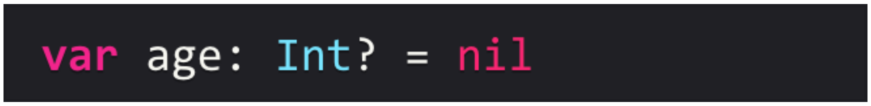
SWIFT INTRO :
OPTIONALS & UNWRAPPING & TYPECASTING

✓ Handling Missing Data (Optionals)

- You can make ***optionals*** out of any type. They allow us to represent the absence of some data
- To make a type optional, add a question mark after it.

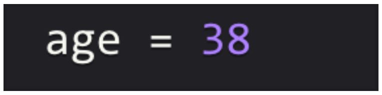
EX:

An optional integer might have a number like 0 or 40, but it might have no value at all – it might literally be missing, which is **nil** in Swift.



```
var age: Int? = nil
```

That doesn't hold any number – it holds nothing. But if we later learn that **age**, we can use it:



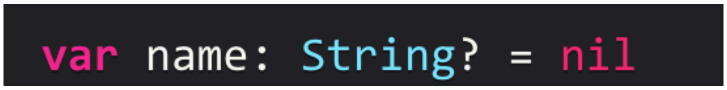
```
age = 38
```

watch more: https://www.youtube.com/watch?v=7a7As0uNWOQ

✓ Unwrapping Optional

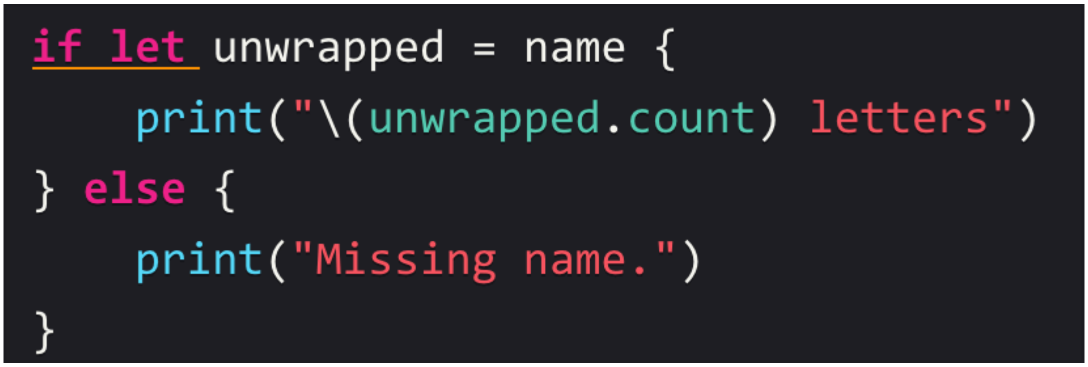
- **IF-LET UNWRAPPING**:
- A common way of unwrapping optionals is with **if let** syntax, which unwraps with a condition.
- If there was a value inside the optional then you can use it, but if there wasn't, the condition fails.

EX:
we have an optional string



```
var name: String? = nil
```

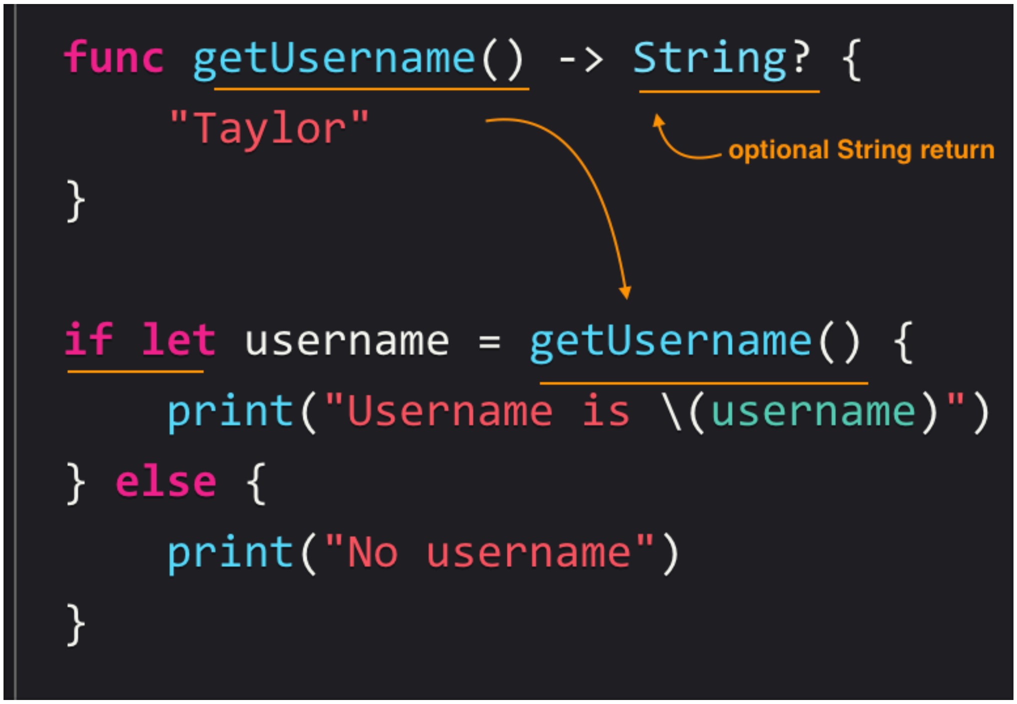
A real string has a **count** property that stores how many letters it has, but this is **nil** – it's empty memory, not a string, so it doesn't have a **count**. => Therefore we need to wrap 'name'



```swift
if let unwrapped = name {
    print("\(unwrapped.count) letters")
} else {
    print("Missing name.")
}
```

+ If **name** holds a string, it will be put inside **unwrapped** as a regular **String** and we can read its **count** property inside the condition.
+ If **name** is empty, the **else** code will be run.

Another example :



```swift
func getUsername() -> String? {
    "Taylor"
}

if let username = getUsername() {
    print("Username is \(username)")
} else {
    print("No username")
}
```

optional String return

- **GUARD-LET UNWRAPPING**:

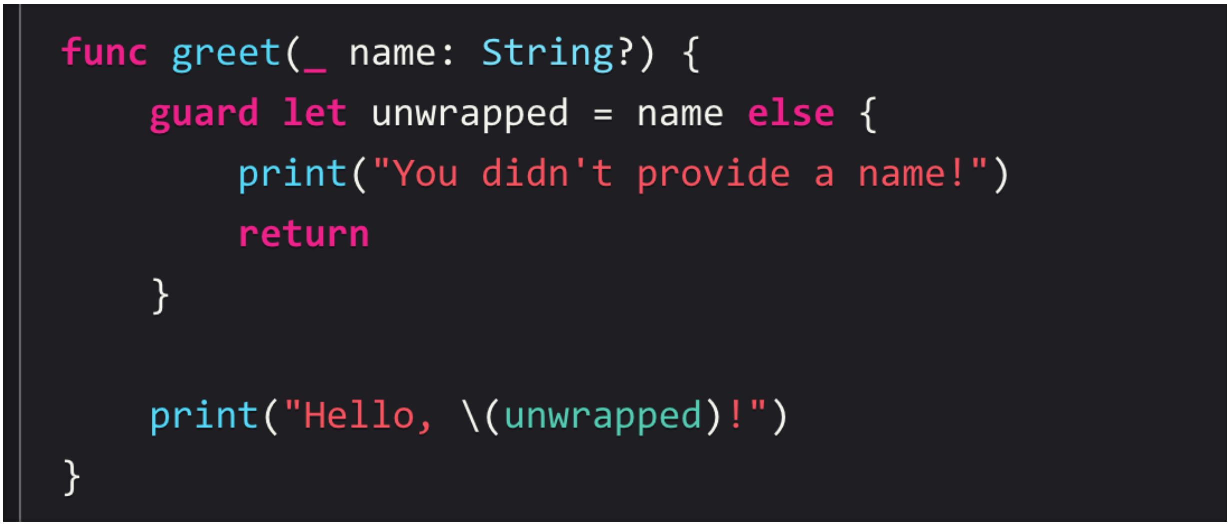
– **guard let** will unwrap an optional for you, but if it finds **nil** inside it expects you to exit the function, loop, or condition you used it in.

PROS:

– The major difference between **if let** and **guard let** is that your unwrapped optional <u>remains usable</u> after the **guard** code.

- Using **guard let** lets you deal with problems at the start of your functions, then exit immediately.

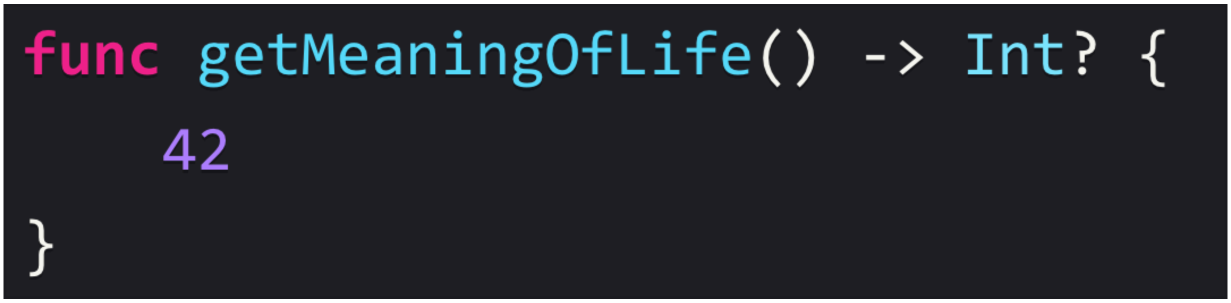
EX:
+ If there's nothing inside name, it will print a message and exit.
+ We stay around after the **guard** finishes, we print the unwrapped string at the end of the function:



```swift
func greet(_ name: String?) {
    guard let unwrapped = name else {
        print("You didn't provide a name!")
        return
    }

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

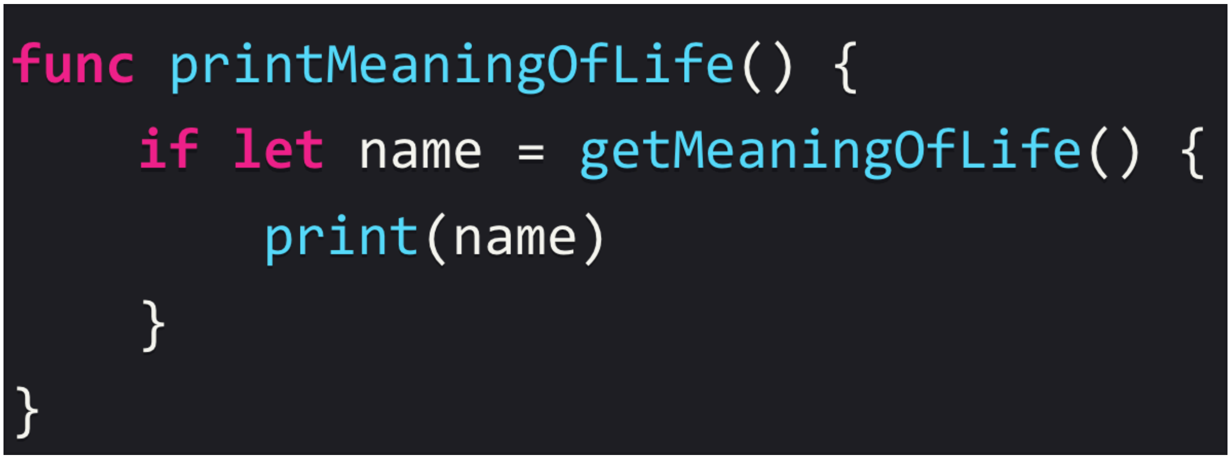
- **IF-LET vs GUARD-LET UNWRAPPING**:

EX:



```swift
func getMeaningOfLife() -> Int? {
    42
}
```

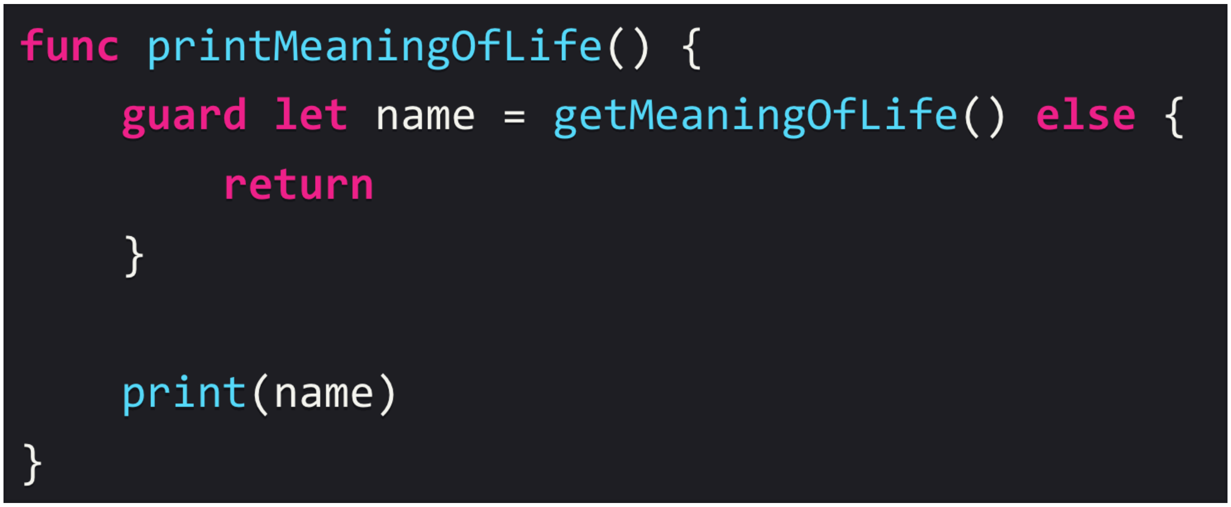
Using if-let



```
func printMeaningOfLife() {
    if let name = getMeaningOfLife() {
        print(name)
    }
}
```

-> the result of **getMeaningOfLife()** will only be printed if it returned an integer rather than nil.

Using guard-let

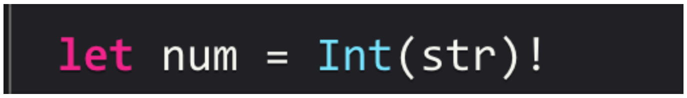


```
func printMeaningOfLife() {
    guard let name = getMeaningOfLife() else {
        return
    }

    print(name)
}
```

guard requires that we exit the current scope when it's used, which in this case means we must return from the function if it fails.

✓ Force Unwrapping (!)

- If you know for *sure* that a value isn't nil, you can force unwrap the result by writing **!**
- Swift will immediately unwrap the optional and make **num** a regular **Int** rather than an **Int?**. But if you're *wrong,* your code will crash.

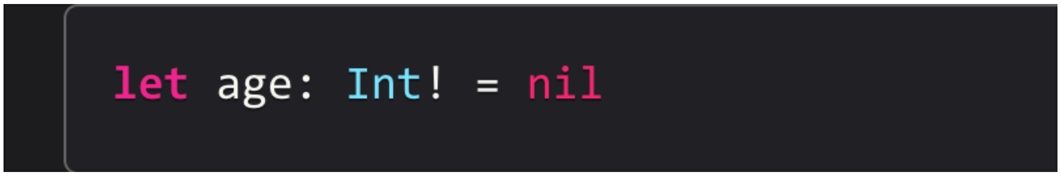


```
let num = Int(str)!
```

✓ Implicitly unwrapped optional

- Implicitly unwrapped optionals are created by adding an exclamation mark after your type name

Implicitly unwrapped optional	regular optionals
SAME:	might contain a value or they might be **nil**.
DIFFERENT:	
you don’t need to unwrap them in order to use.	you need to unwrap them in order to use
if you attempt to use an implicitly unwrapped optional and it’s actually nil, your code will just crash	



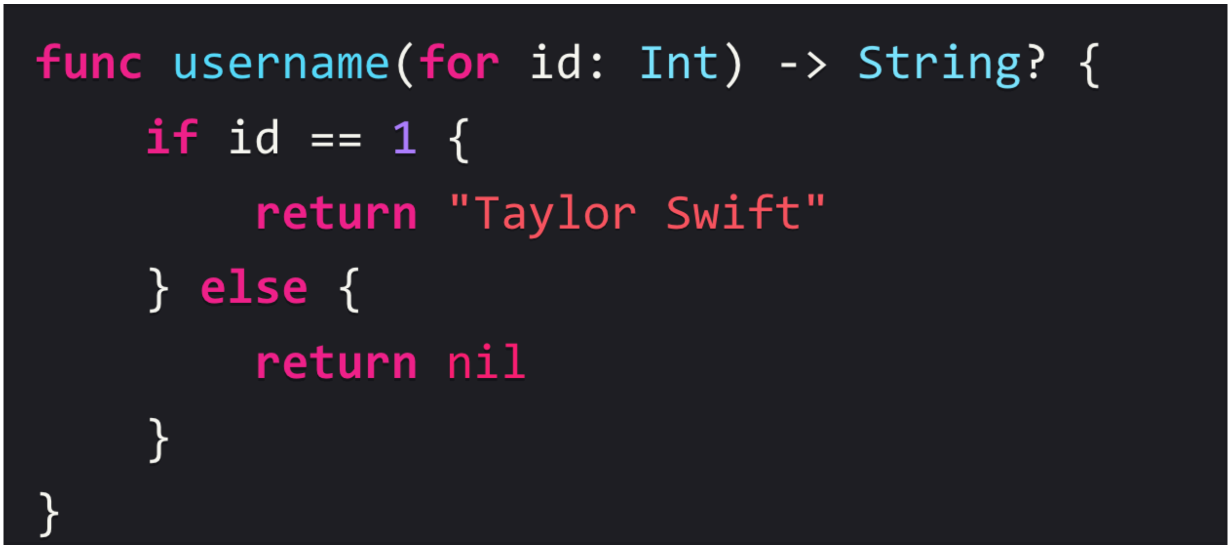
```
let age: Int! = nil
```

REASON TO USE:
Implicitly unwrapped optionals exist because sometimes a variable will start life as nil, but will always have a value before you need to use it. Because you know they will have a value by the time you need them

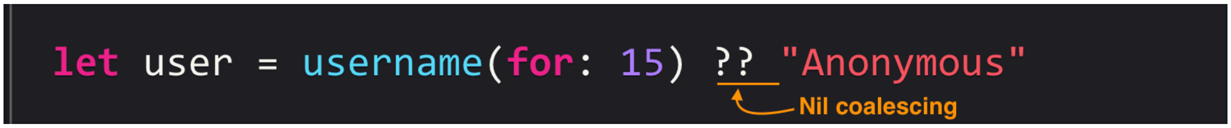
✓ Nil coalescing

- Nil coalescing lets us attempt to unwrap an optional, but provide a default value if the optional contains nil.

=> it will either be the value from inside the optional or the default value used as a backup.

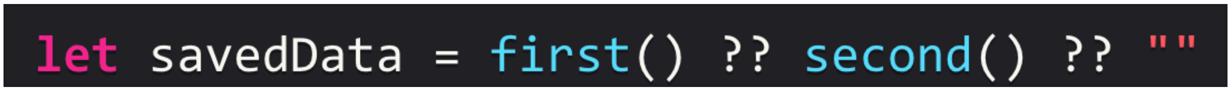


```swift
func username(for id: Int) -> String? {
    if id == 1 {
        return "Taylor Swift"
    } else {
        return nil
    }
}
```



```swift
let user = username(for: 15) ?? "Anonymous"
```

- Chain nil coalescing (not common)



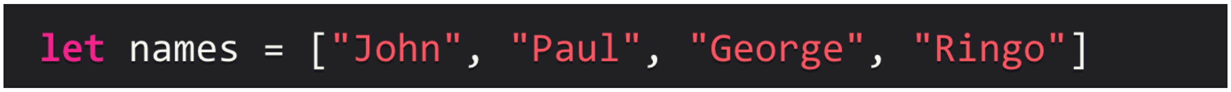
```swift
let savedData = first() ?? second() ?? ""
```

That will attempt to run **first()**, and if that returns nil attempt to run **second()**, and if *that* returns nil then it will use an empty string.

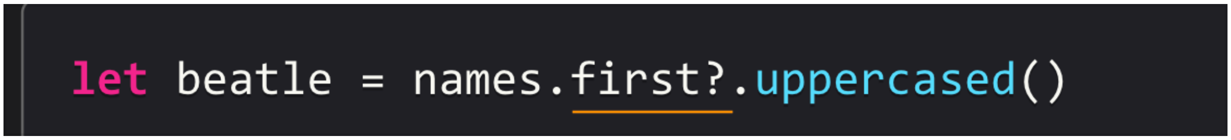
✓ Optional Chaining

- If you want to access something like **a.b.c** and **b** is optional, you can write a question mark after it to enable *optional chaining*: **a.b?.c**.
- When that code is run, Swift will check whether **b** has a value, and if it's **nil** the rest of the line will be ignored

EX:



```swift
let names = ["John", "Paul", "George", "Ringo"]
```

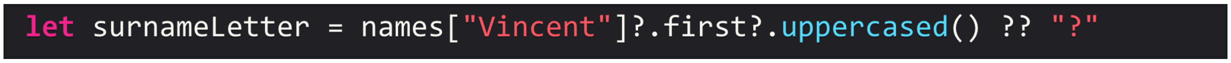


```swift
let beatle = names.first?.uppercased()
```

if **first** returns **nil** then Swift won't try to uppercase it, and will set **beatle** to **nil** immediately.

Another example:

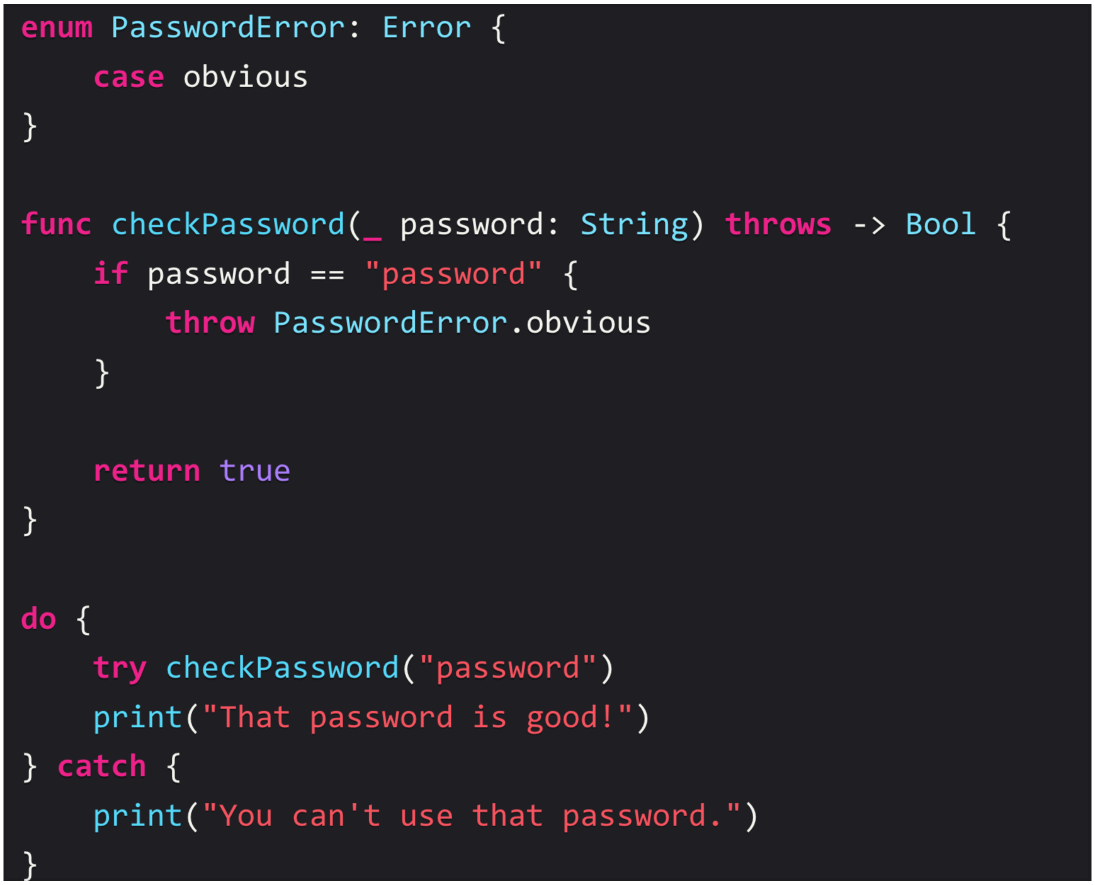
we could automatically return “?” if we were unable to read the first letter of someone’s surname:



```
let surnameLetter = names["Vincent"]?.first?.uppercased() ?? "?"
```

✅ Optional Try

Normal do-try-catch:

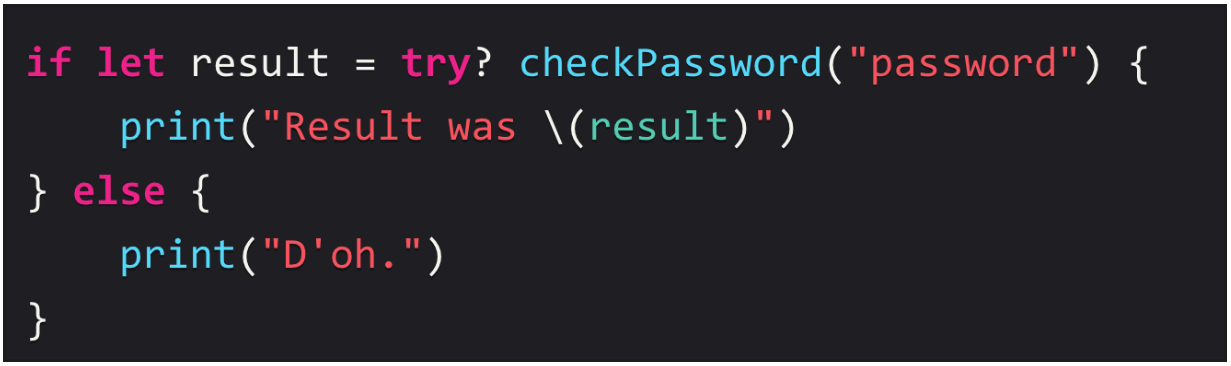


```
enum PasswordError: Error {
    case obvious
}

func checkPassword(_ password: String) throws -> Bool {
    if password == "password" {
        throw PasswordError.obvious
    }

    return true
}

do {
    try checkPassword("password")
    print("That password is good!")
} catch {
    print("You can't use that password.")
}
```

There are 2 alternatives to **try:**

1. **try?**

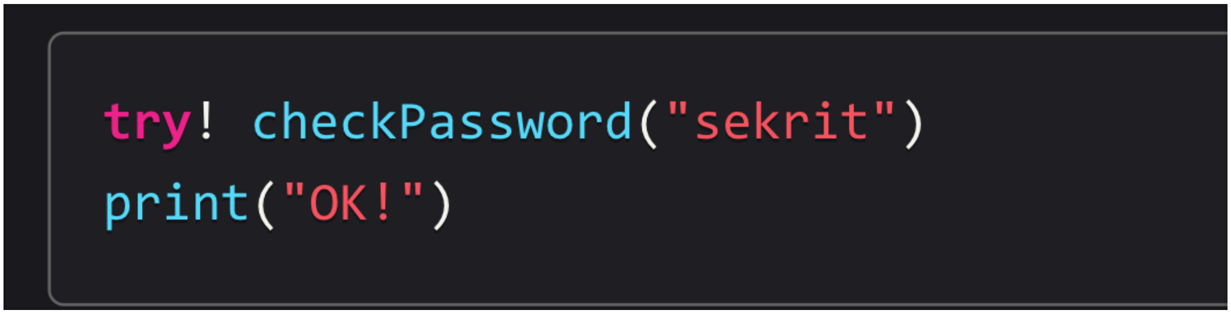
+ Changes throwing functions into functions that return an optional.
+ If the function succeeds, its return value will be an optional containing whatever you would normally have received back,
+ If it fails the return value will be an optional set to nil.



```swift
if let result = try? checkPassword("password") {
    print("Result was \(result)")
} else {
    print("D'oh.")
}
```

2. try!

you can use when you know for sure that the function will not fail. If the function *does* throw an error, your code will crash.



```swift
try! checkPassword("sekrit")
print("OK!")
```

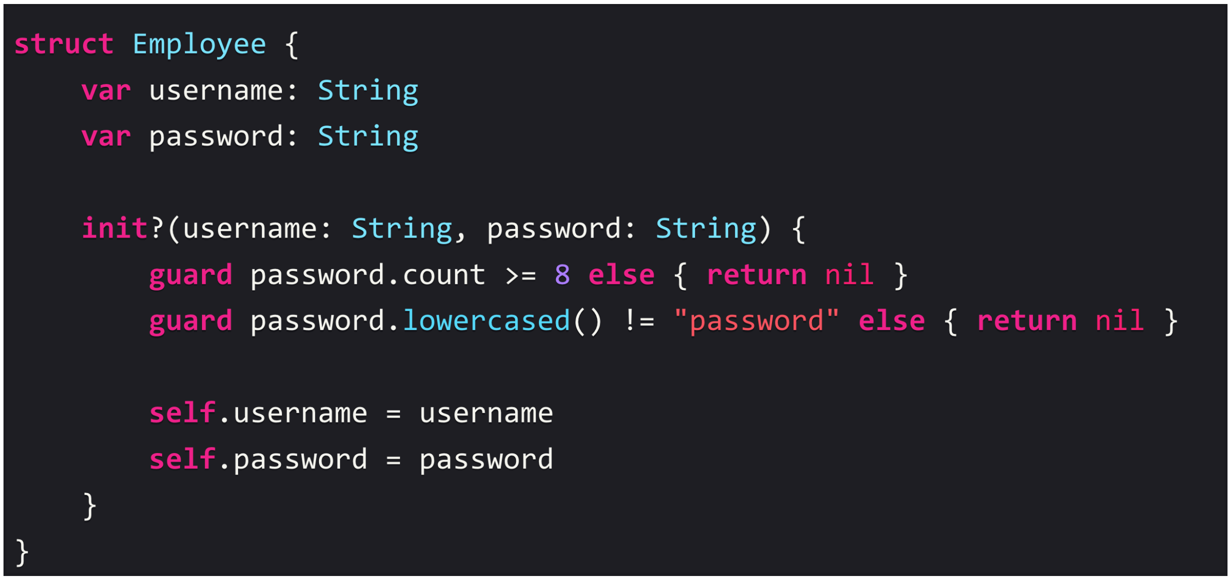
- Usage:

If you want to run a function and care only that it succeeds or fails – you don’t need to distinguish between the various reasons why it might fail – then using optional try

✓ Failable Initializer

- A *failable initializer*: an initializer that might work or might not.
- You can write these in your own structs and classes by using **init?()** rather than **init()**, and return **nil** if something goes wrong.
- The return value will then be an optional of your type, for you to unwrap however you want.
- Failable initializers give us the opportunity to back out of an object’s creation if validation checks fail.

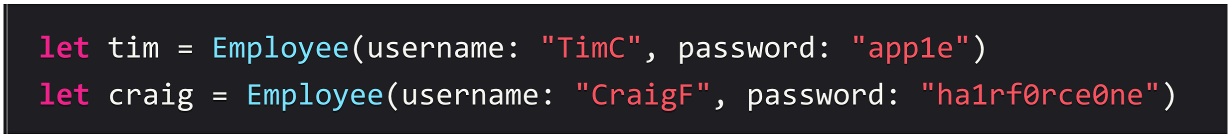
EX:



```swift
struct Employee {
    var username: String
    var password: String

    init?(username: String, password: String) {
        guard password.count >= 8 else { return nil }
        guard password.lowercased() != "password" else { return nil }

        self.username = username
        self.password = password
    }
}
```

Employee struct that has a failable initializer with two checks:

1. passwords be at least 8 characters
2. not be the string “password”.



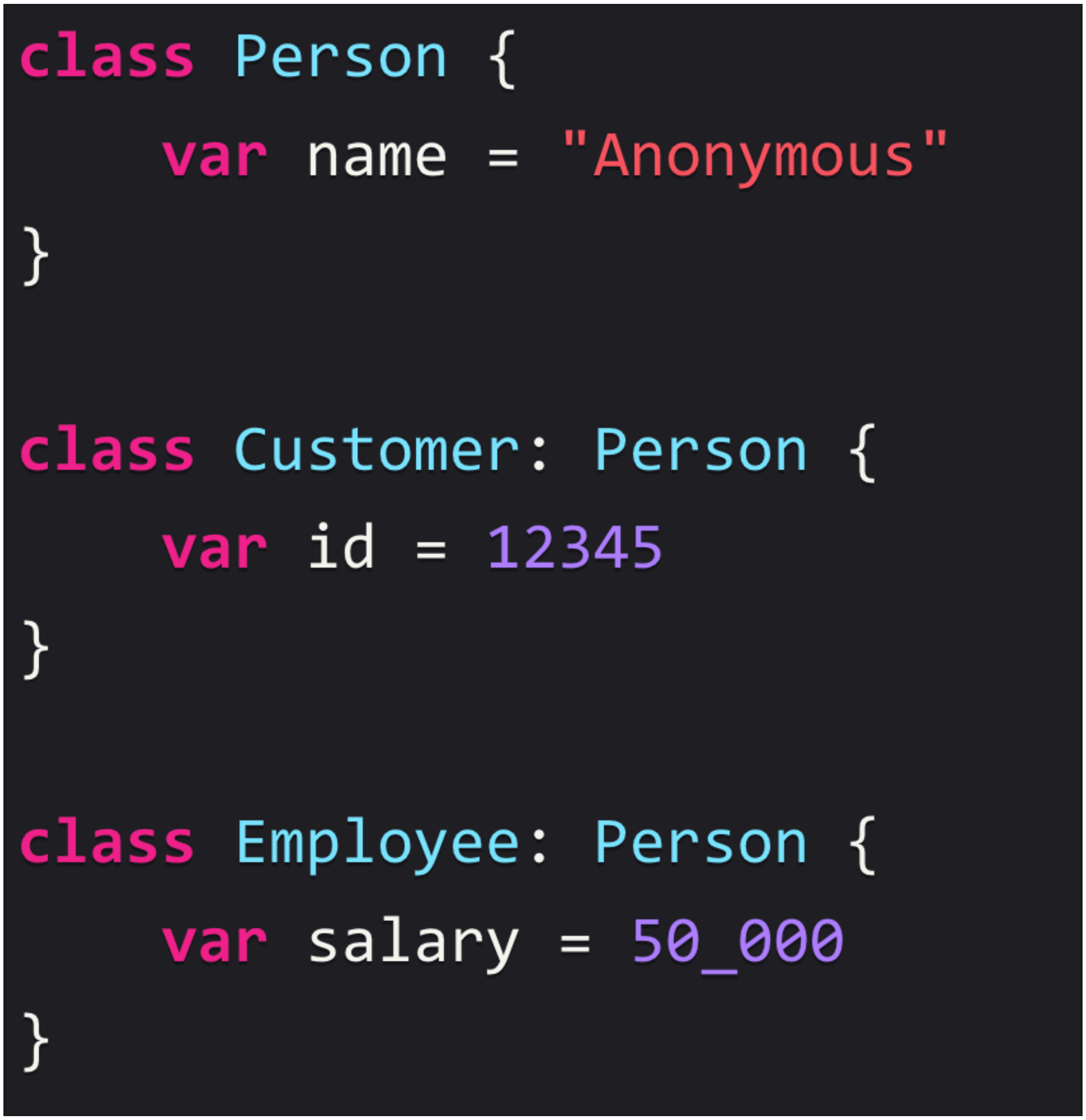
```swift
let tim = Employee(username: "TimC", password: "app1e")
let craig = Employee(username: "CraigF", password: "ha1rf0rce0ne")
```

The first of those (tim) will be an optional set to nil because the password is too short, but the second (craig) will be an optional set to a valid **User** instance.

✓ TypeCasting

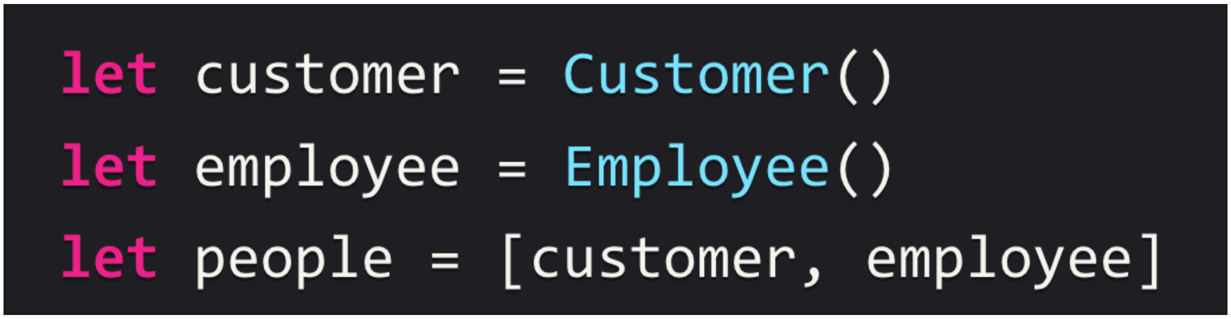
- uses a keyword called **as?**, which returns an optional: it will be **nil** if the typecast failed, or a converted type otherwise.
- helpful when working with protocols and class inheritance.

EX:



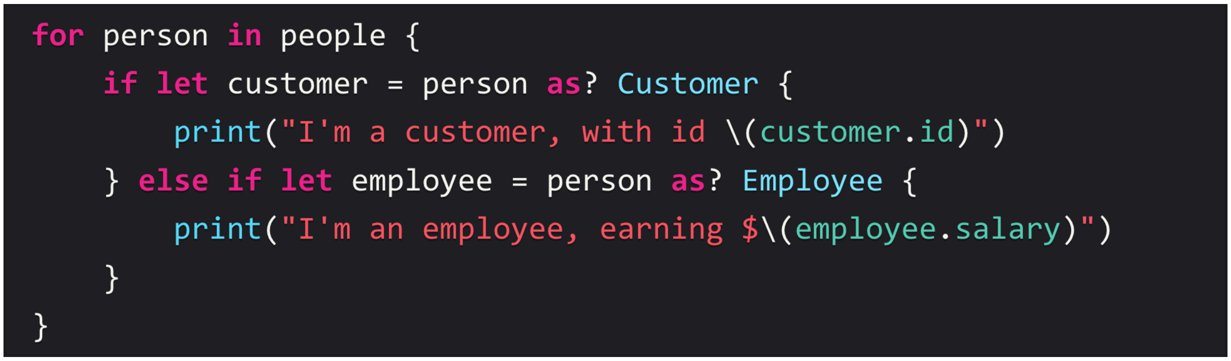
```swift
class Person {
    var name = "Anonymous"
}

class Customer: Person {
    var id = 12345
}

class Employee: Person {
    var salary = 50_000
}
```

We have an 'people' array of **Person** type because both **Customer** and **Employee** inherit from **Person**



```swift
let customer = Customer()
let employee = Employee()
let people = [customer, employee]
```

So, if we loop over **people** we'll only be able to access the **name** of each item in the array.
-> We need to typecast to get id if it's **Customer type** or get salary if it's **Employee type**



```swift
for person in people {
    if let customer = person as? Customer {
        print("I'm a customer, with id \(customer.id)")
    } else if let employee = person as? Employee {
        print("I'm an employee, earning $\(employee.salary)")
    }
}
```

-> We attempts to convert **person** first to **Customer** and then to **Employee**.
-> If either test passes, we can then use the extra properties that belong to that class, as well as the **name** property from the parent class.