A.STEFANCHUK IT STEP

SWIFT FINALE

EXTENSIONS

- Allow us to extend our classes, structs and enums

EXTENSIONS

- Allow us to extend our classes, structs and enums
- Very convenient to keep code in logic groups

EXTENSIONS

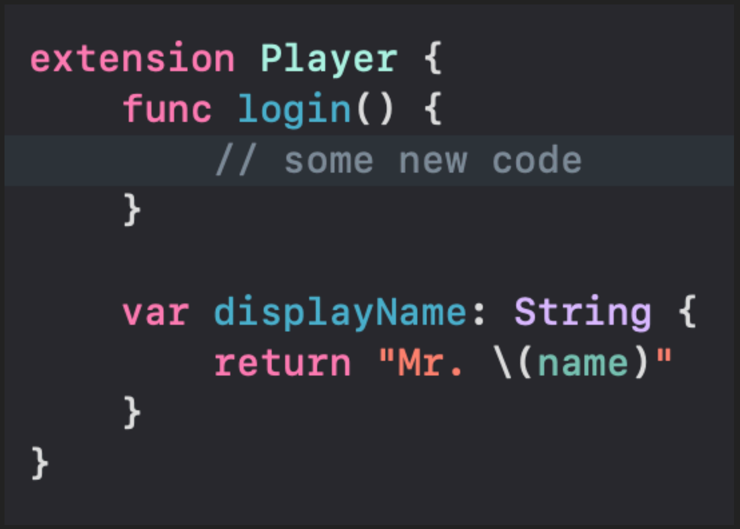
- Allow us to extend our classes, structs and enums
- Very convenient to keep code in logic groups
- Can be used to conform protocols

EXTENSIONS

- Allow us to extend our classes, structs and enums
- Very convenient to keep code in logic groups
- Can be used to conform protocols
- Can add new methods, computed properties, inits, subscripts, nested types

EXTENSIONS

- Allow us to extend our classes, structs and enums
- Very convenient to keep code in logic groups
- Can be used to conform protocols
- Can add new methods, computed properties, inits, subscripts, nested types
- Can't add new stored properties!



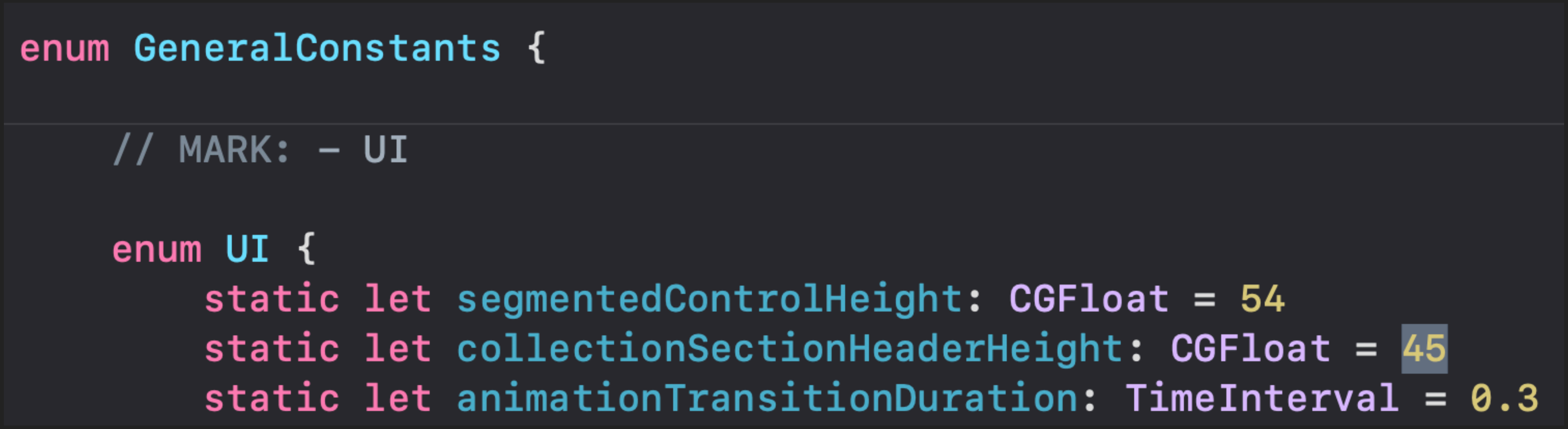
```swift
extension Player {
    func login() {
        // some new code
    }

    var displayName: String {
        return "Mr. \(name)"
    }
}
```

NESTED TYPES

- You can nest types inside each other to make more convenient structure of code or name spacing

NESTED TYPES

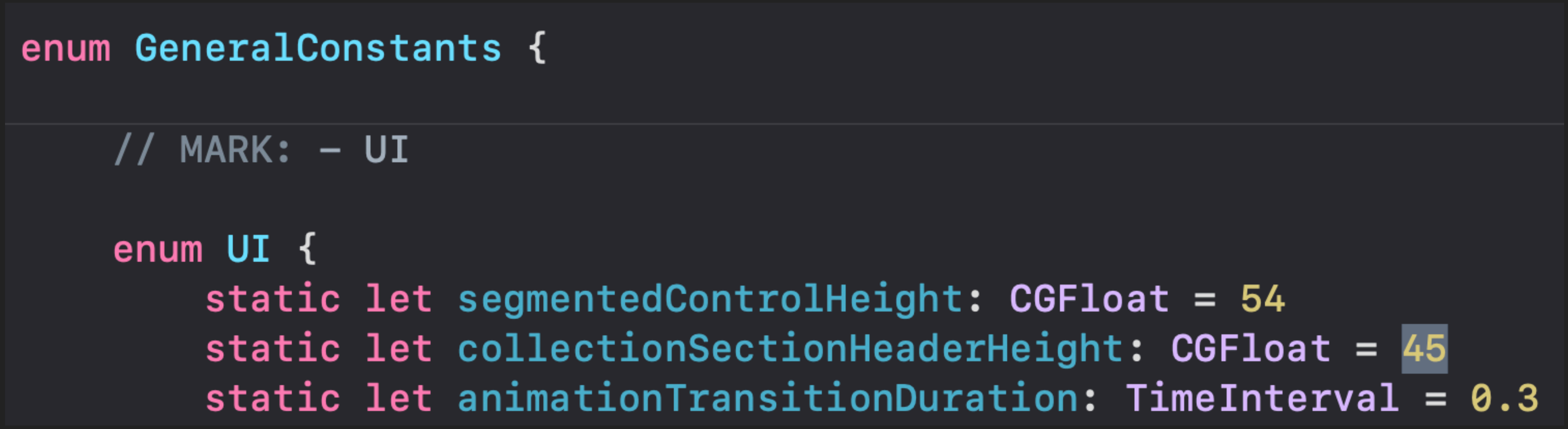
- You can nest types inside each other to make more convenient structure of code or name spacing



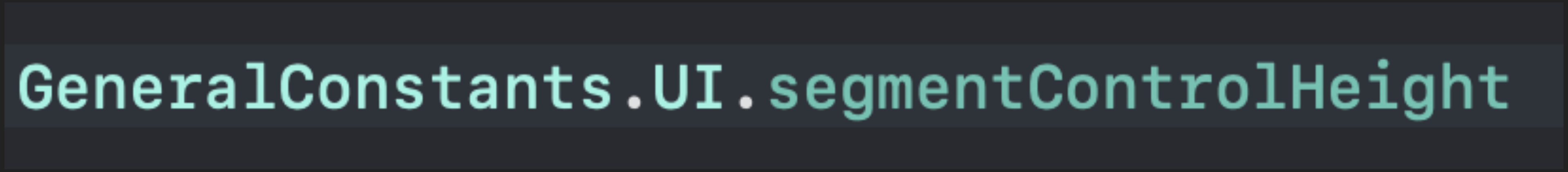
```swift
enum GeneralConstants {

    // MARK: - UI

    enum UI {
        static let segmentedControlHeight: CGFloat = 54
        static let collectionSectionHeaderHeight: CGFloat = 45
        static let animationTransitionDuration: TimeInterval = 0.3
```

NESTED TYPES

- You can nest types inside each other to make more convenient structure of code or name spacing



```swift
enum GeneralConstants {

    // MARK: - UI

    enum UI {
        static let segmentedControlHeight: CGFloat = 54
        static let collectionSectionHeaderHeight: CGFloat = 45
        static let animationTransitionDuration: TimeInterval = 0.3
```



```swift
GeneralConstants.UI.segmentControlHeight
```

INIT

- There are a lot of rules, but you should remember:
- You can have as much inits as needed

INIT

- There are a lot of rules, but you should remember:
- You can have as much inits as needed
- Every var should have value after init()

INIT

- There are a lot of rules, but you should remember:
- You can have as much inits as needed
- Every var should have value after init()
- Super.init() should come first

INIT

- There are a lot of rules, but you should remember:
- You can have as much inits as needed
- Every var should have value after init()
- Super.init() should come first
- Convenience init() should call some of main init()

INIT

- There are a lot of rules, but you should remember:
- You can have as much inits as needed
- Every var should have value after init()
- Super.init() should come first
- Convenience init() should call some of main init()
- init?() - is failable init that can return nil

INIT

- There are a lot of rules, but you should remember:
- You can have as much inits as needed
- Every var should have value after init()
- Super.init() should come first
- Convenience init() should call some of main init()
- init?() - is failable init that can return nil
- required init() should be implemented in subclasses

DEINIT

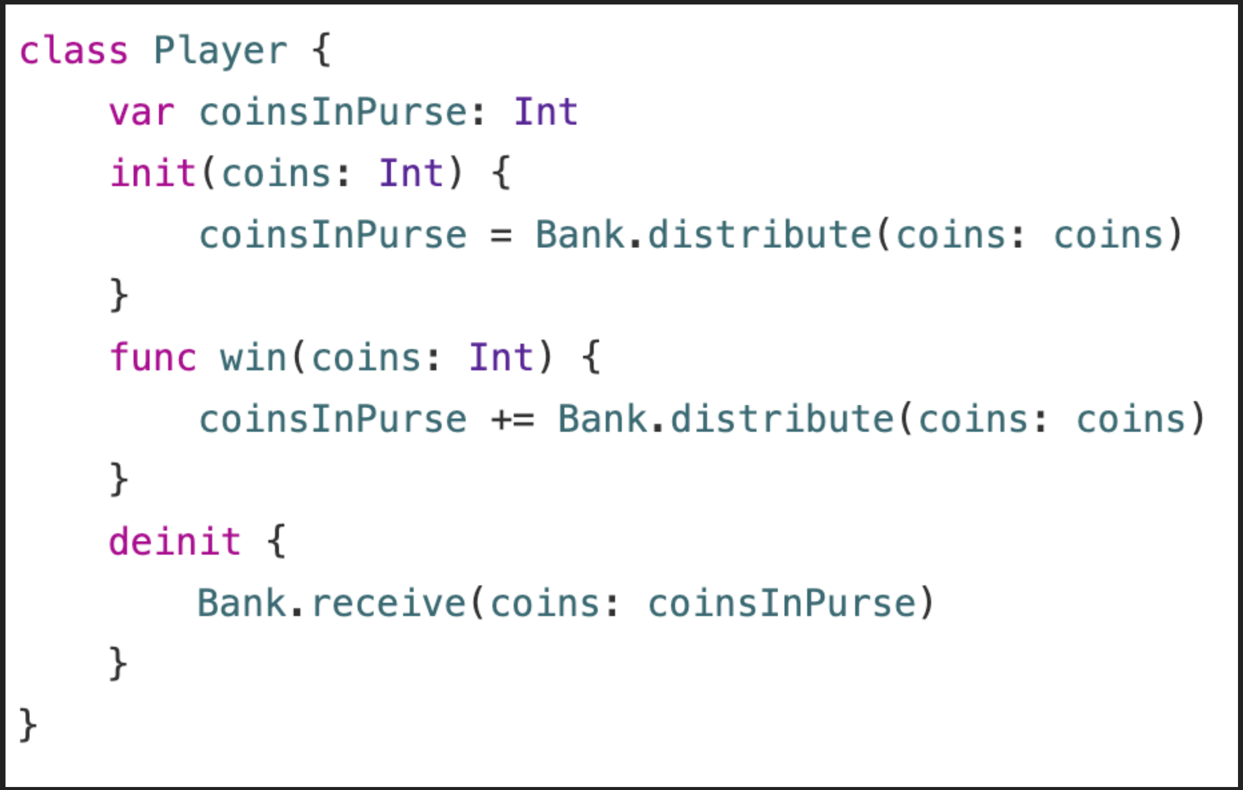
- Deinit allows us clear some resources, close files and other stuff before the class object is gone

DEINIT

- Deinit allows us clear some resources, close files and other stuff before the class object is gone
- Object is deinited when nobody has reference to it

DEINIT

- Deinit allows us clear some resources, close files and other stuff before the class object is gone
- Object is deinited when nobody has reference to it



```swift
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
}
```

ERROR HANDLING

- Function that may throw errors should be marked with ‘throws’

ERROR HANDLING

- Function that may throw errors should be marked with ‘throws’
- To handle errors we put throwing functions inside do-catch block

ERROR HANDLING

- Function that may throw errors should be marked with ‘throws’
- To handle errors we put throwing functions inside do-catch block
- You can ignore errors with try! or try? - they will return optional values

ERROR HANDLING

- Function that may throw errors should be marked with ‘throws’
- To handle errors we put throwing functions inside do-catch block
- You can ignore errors with try! or try? - they will return optional values
- You can catch some specific and casted errors if needed

ERROR HANDLING

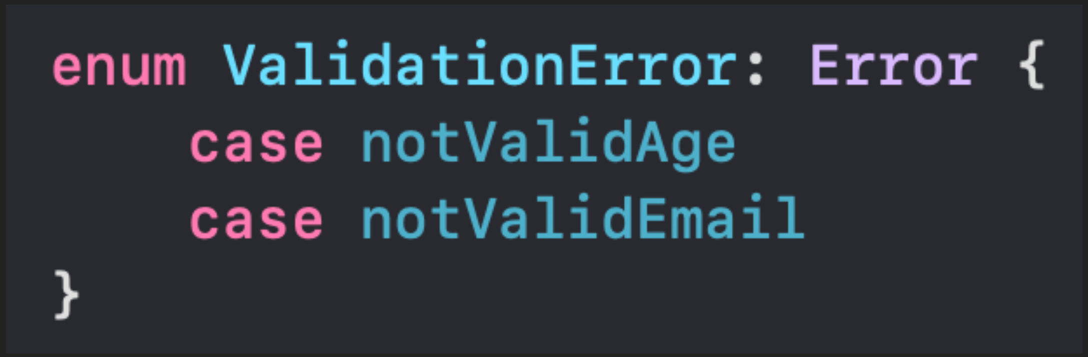
```
enum ValidationError: Error {
    case notValidAge
    case notValidEmail
}
```

ERROR HANDLING

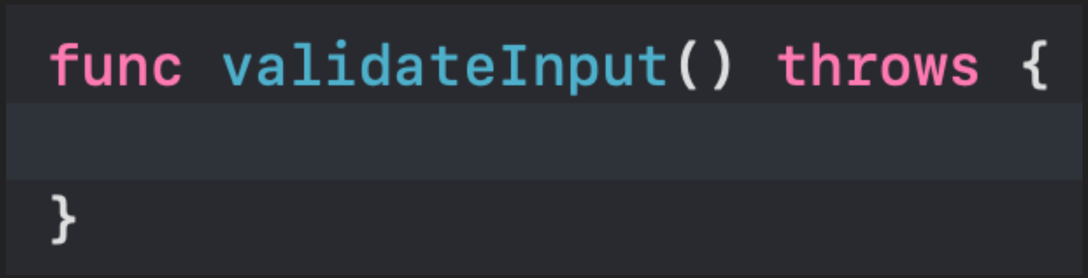
```
enum ValidationError: Error {
    case notValidAge
    case notValidEmail
}
```

```
func validateInput() throws {

}
```

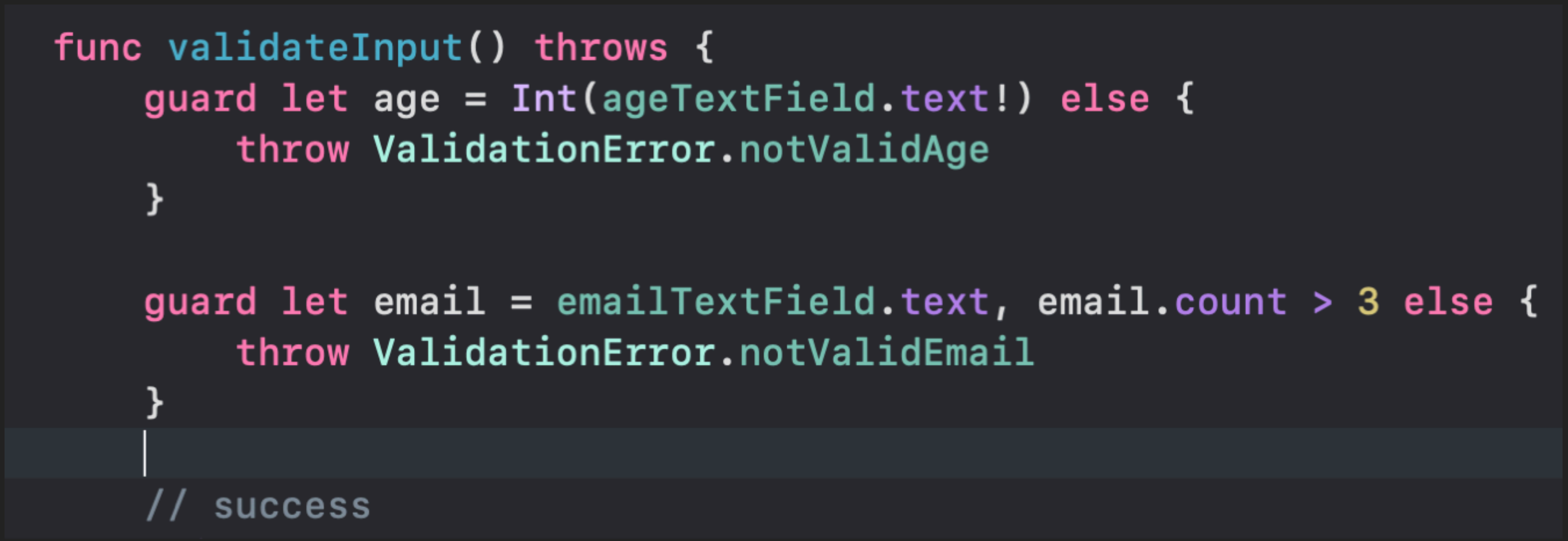
ERROR HANDLING



```swift
enum ValidationError: Error {
    case notValidAge
    case notValidEmail
}
```

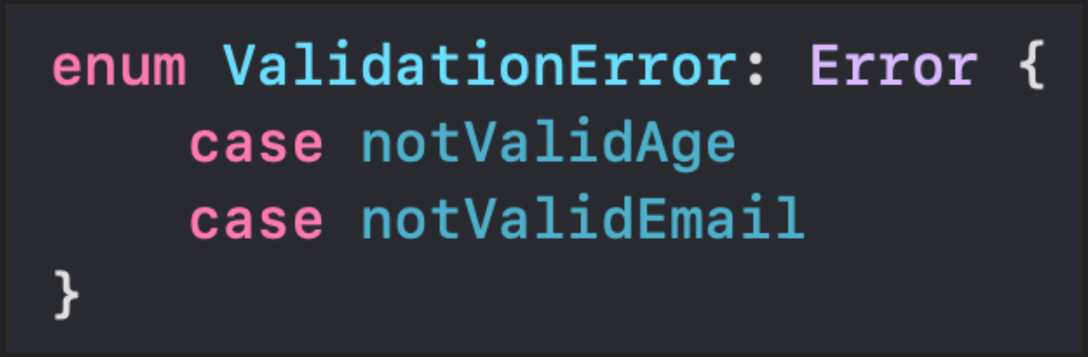


```swift
func validateInput() throws {

}
```

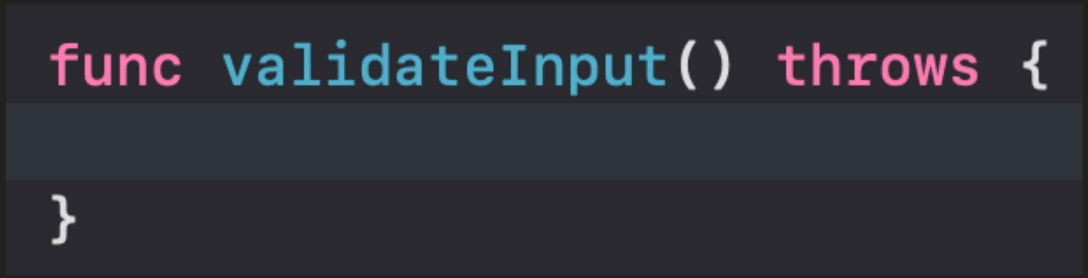


```swift
func validateInput() throws {
    guard let age = Int(ageTextField.text!) else {
        throw ValidationError.notValidAge
    }

    guard let email = emailTextField.text, email.count > 3 else {
        throw ValidationError.notValidEmail
    }

    // success
```

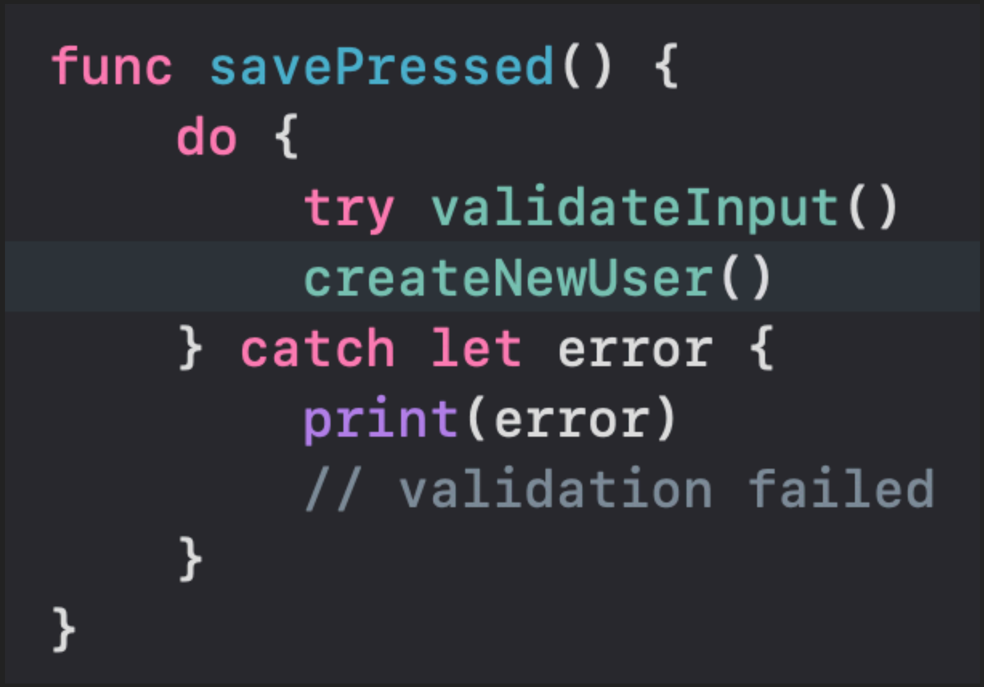
ERROR HANDLING



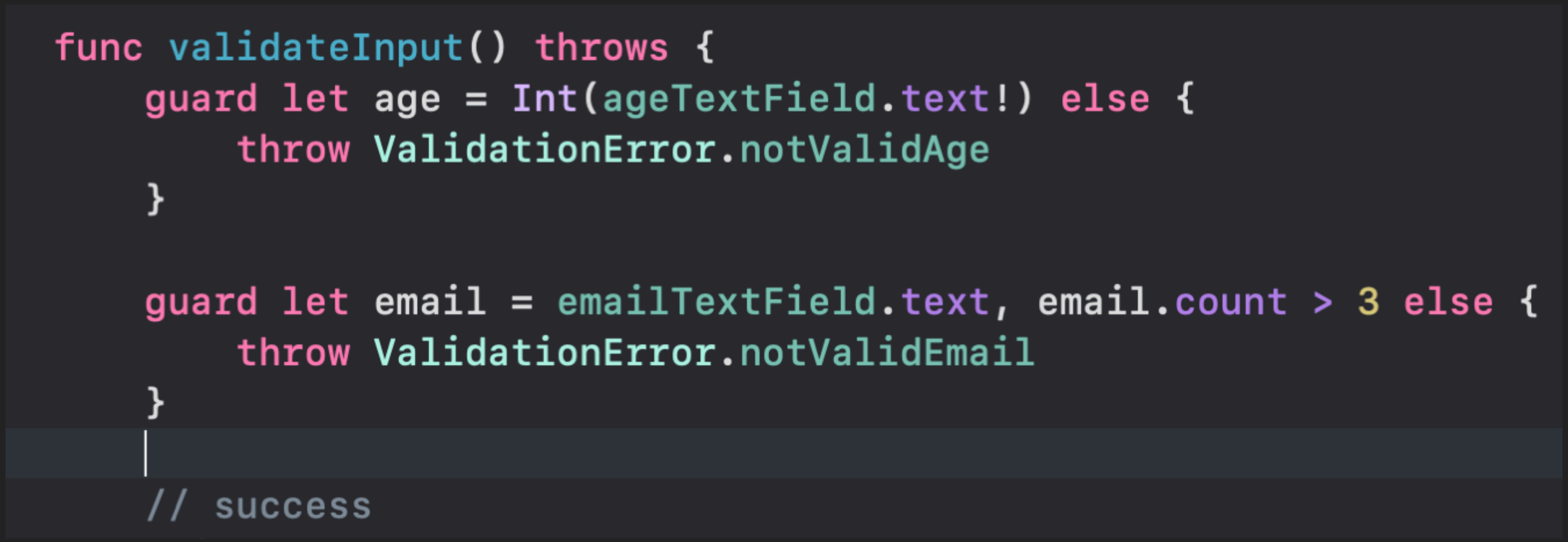
```swift
enum ValidationError: Error {
    case notValidAge
    case notValidEmail
}
```



```swift
func validateInput() throws {

}
```



```swift
func savePressed() {
    do {
        try validateInput()
        createNewUser()
    } catch let error {
        print(error)
        // validation failed
    }
}
```



```swift
func validateInput() throws {
    guard let age = Int(ageTextField.text!) else {
        throw ValidationError.notValidAge
    }

    guard let email = emailTextField.text, email.count > 3 else {
        throw ValidationError.notValidEmail
    }

    // success
```

HERE IT COMES
makeameme.org

Closures
HERE IT COMES
makeameme.org

CLOSURES

- Self-contained blocks of code that can be passed around

CLOSURES

- Self-contained blocks of code that can be passed around
- Have no name

CLOSURES

- Self-contained blocks of code that can be passed around
- Have no name
- They are reference type!!!

CLOSURES

- Self-contained blocks of code that can be passed around
- Have no name
- They are reference type!!!
- Closures capture and hold values inside its body

CLOSURES

- Self-contained blocks of code that can be passed around
- Have no name
- They are reference type!!!
- Closures capture and hold values inside its body
- Lightweight syntax

CLOSURES

- Self-contained blocks of code that can be passed around
- Have no name
- They are reference type!!!
- Closures capture and hold values inside its body
- Lightweight syntax
- Used everywhere in iOS programming

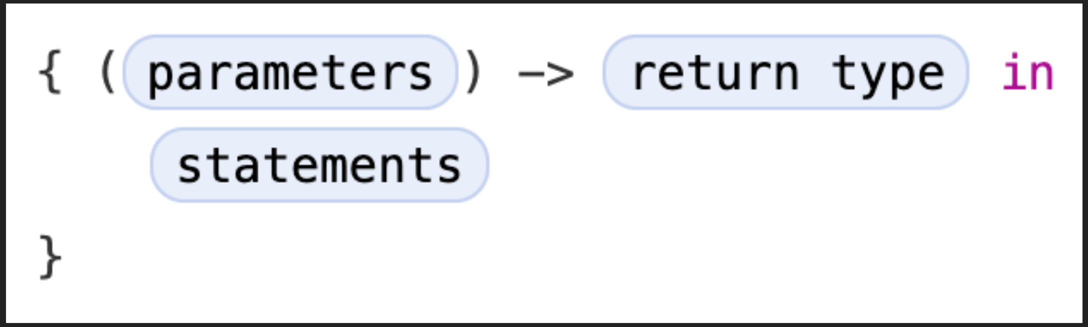
CLOSURES

- Self-contained blocks of code that can be passed around
- Have no name
- They are reference type!!!
- Closures capture and hold values inside its body
- Lightweight syntax
- Used everywhere in iOS programming
- Functions are special types of closures

CLOSURES

GLOBAL FUNCTION
NESTED FUNCTION
CLOSURE

CLOSURES SYNTAX



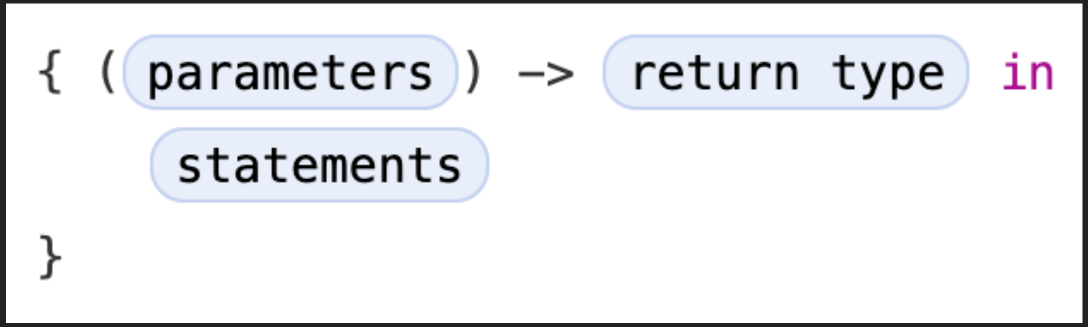
```
{ (parameters) -> return type in
    statements
}
```

CLOSURES SYNTAX

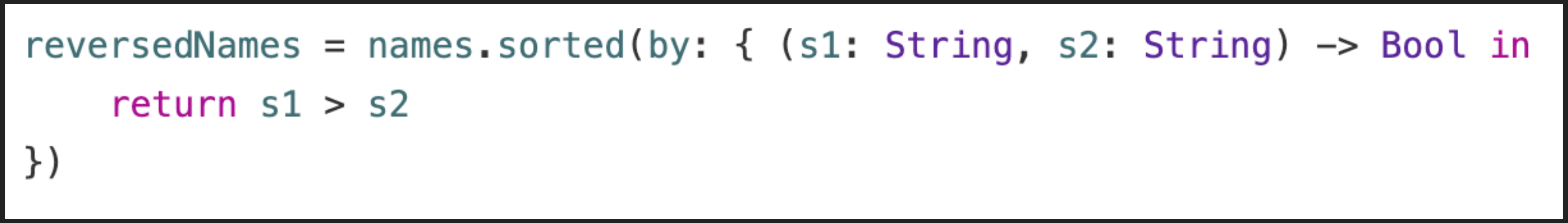
```
{ (parameters) -> return type in
    statements
}
```

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

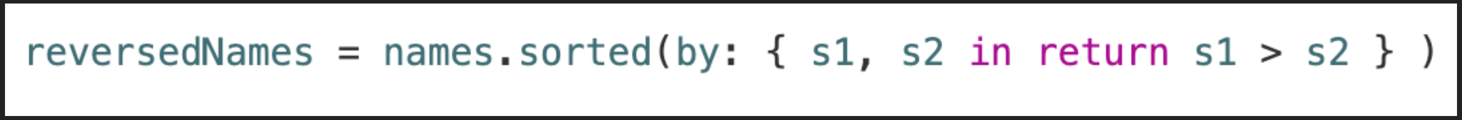
CLOSURES SYNTAX



```
{ (parameters) -> return type in
    statements
}
```

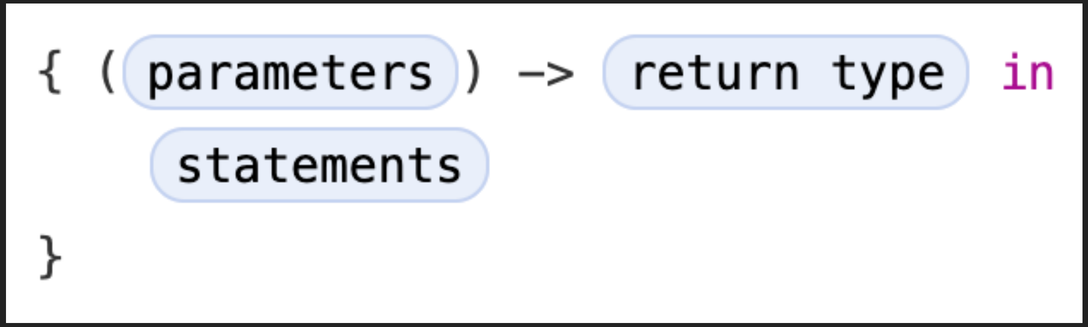


```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

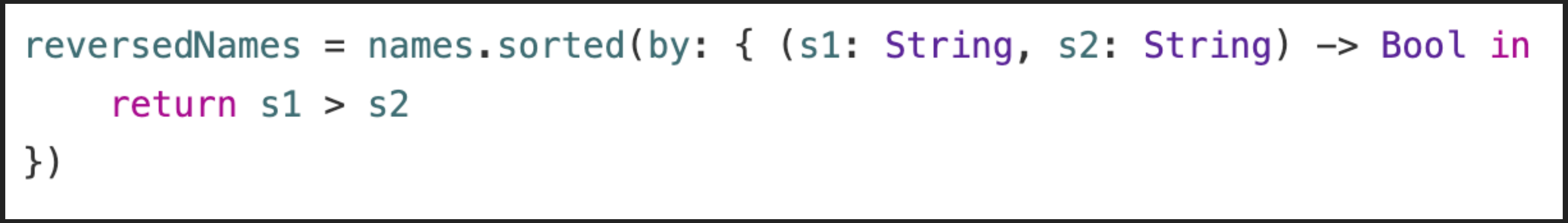


```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

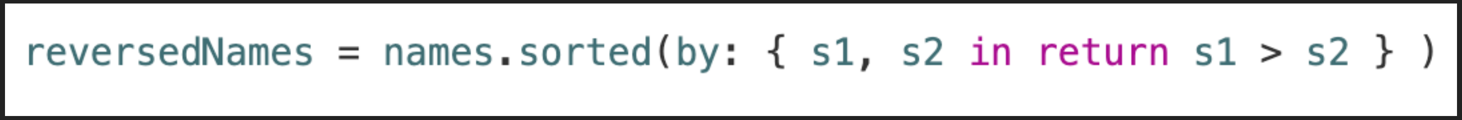
CLOSURES SYNTAX



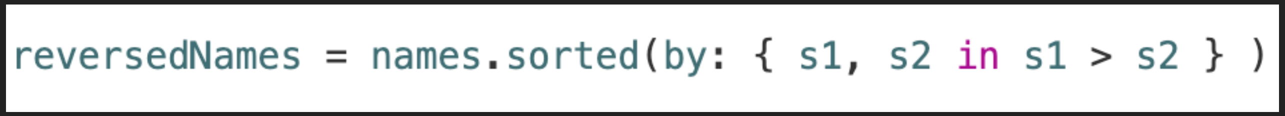
```
{ (parameters) -> return type in
    statements
}
```



```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

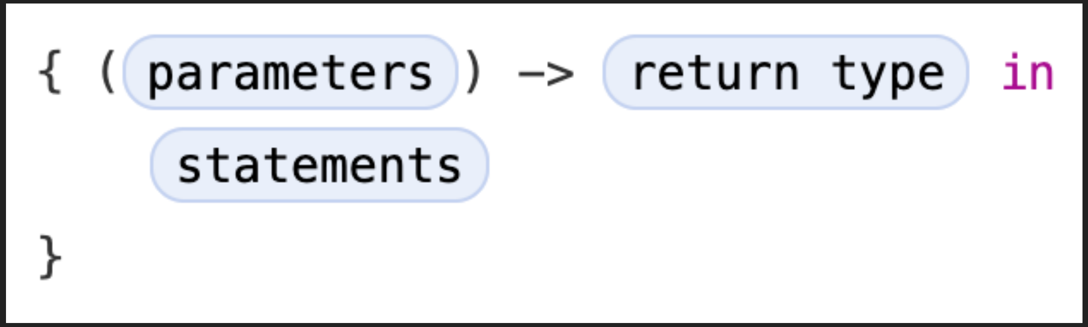


```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

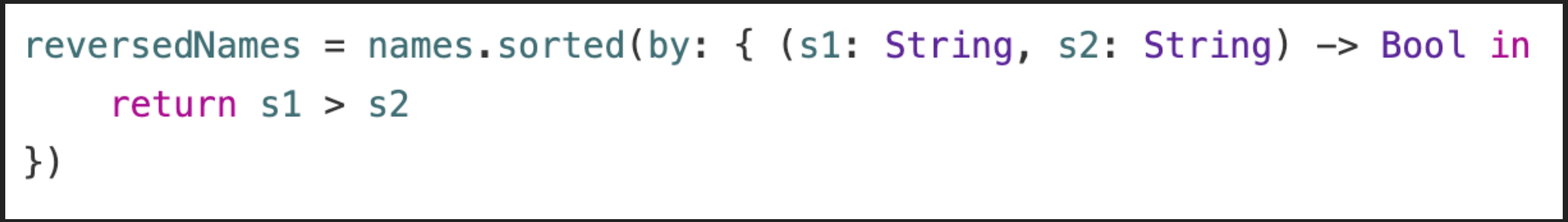


```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

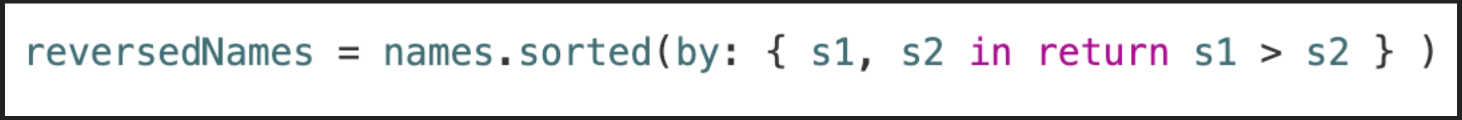
CLOSURES SYNTAX



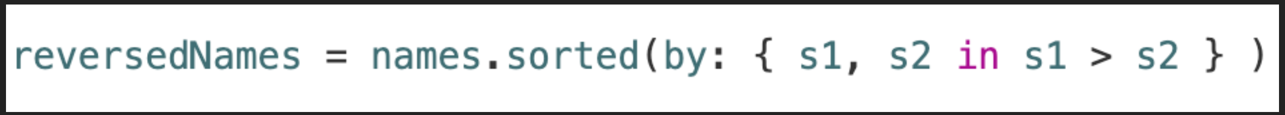
```
{ (parameters) -> return type in
    statements
}
```



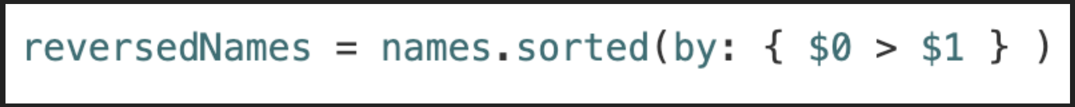
```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```



```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```



```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```



```
reversedNames = names.sorted(by: { $0 > $1 } )
```

CLOSURES SYNTAX

```
{ (parameters) -> return type in
    statements
}
```

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

```
reversedNames = names.sorted(by: >)
```

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

CLOSURES SYNTAX

```
{ (parameters) -> return type in
    statements
}
```

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

```
reversedNames = names.sorted(by: >)
```

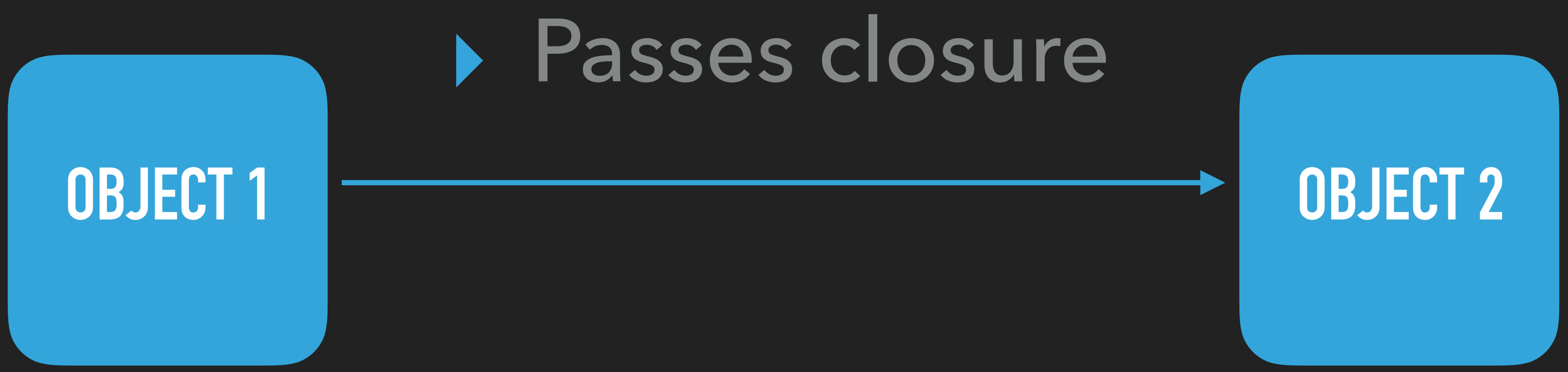
Wow!

CLOSURES USAGE

- Pass it to another class / object

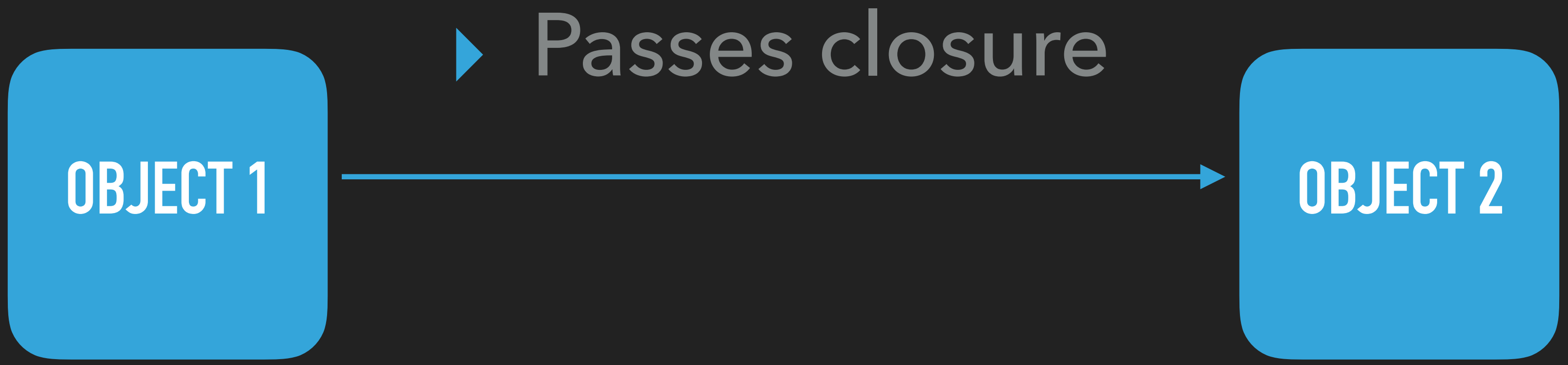
CLOSURES USAGE

- Pass it to another class / object



CLOSURES USAGE

- Pass it to another class / object



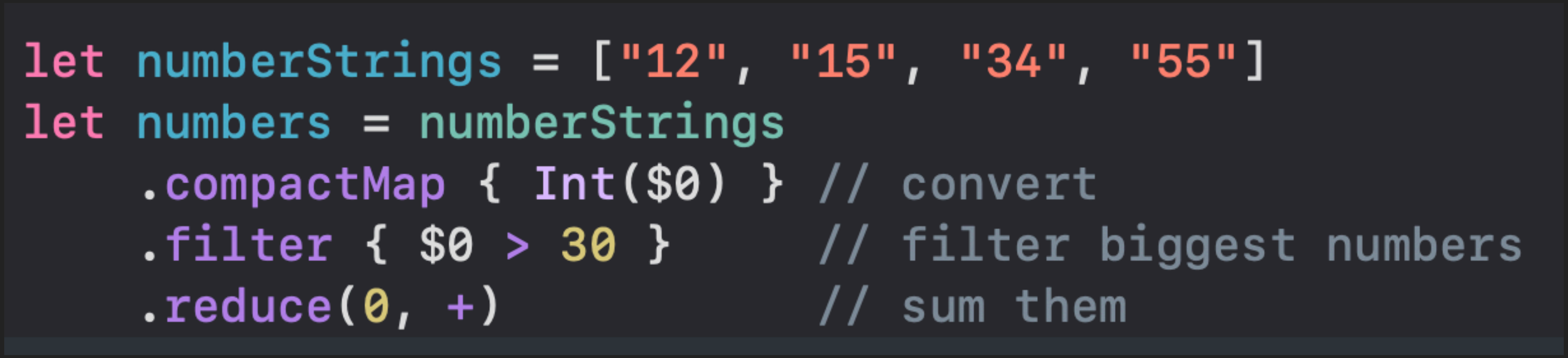
- Something happens in object 2 and instead of calling object 1 func, it just executes closure, without knowing context

CLOSURES USAGE

- High order functions like map, flatmap, filter, reduce

CLOSURES USAGE

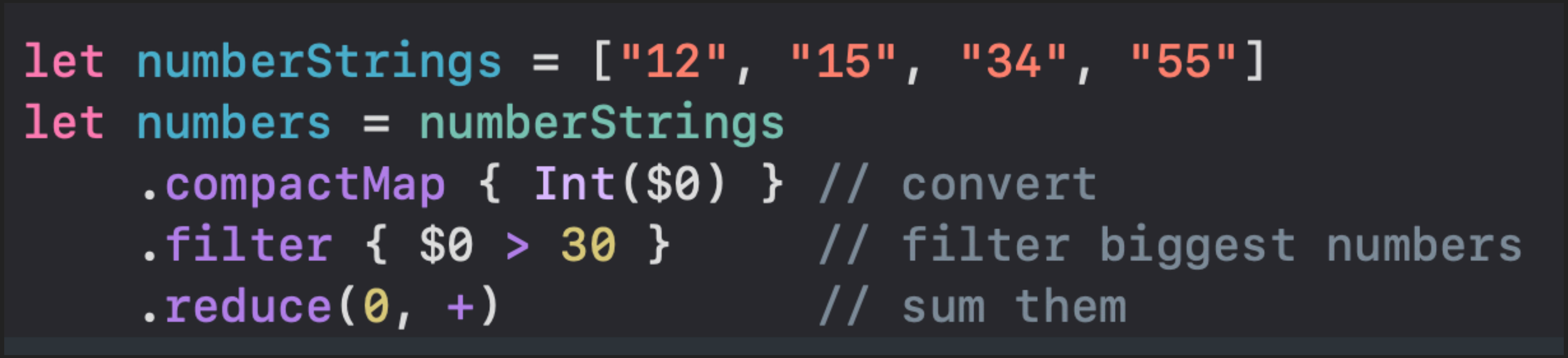
- High order functions like map, flatmap, filter, reduce



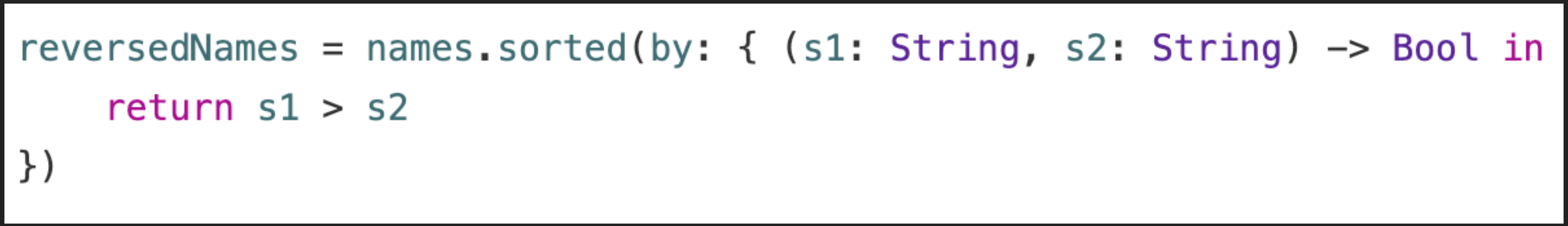
```swift
let numberStrings = ["12", "15", "34", "55"]
let numbers = numberStrings
    .compactMap { Int($0) } // convert
    .filter { $0 > 30 }     // filter biggest numbers
    .reduce(0, +)           // sum them
```

CLOSURES USAGE

- High order functions like map, flatmap, filter, reduce



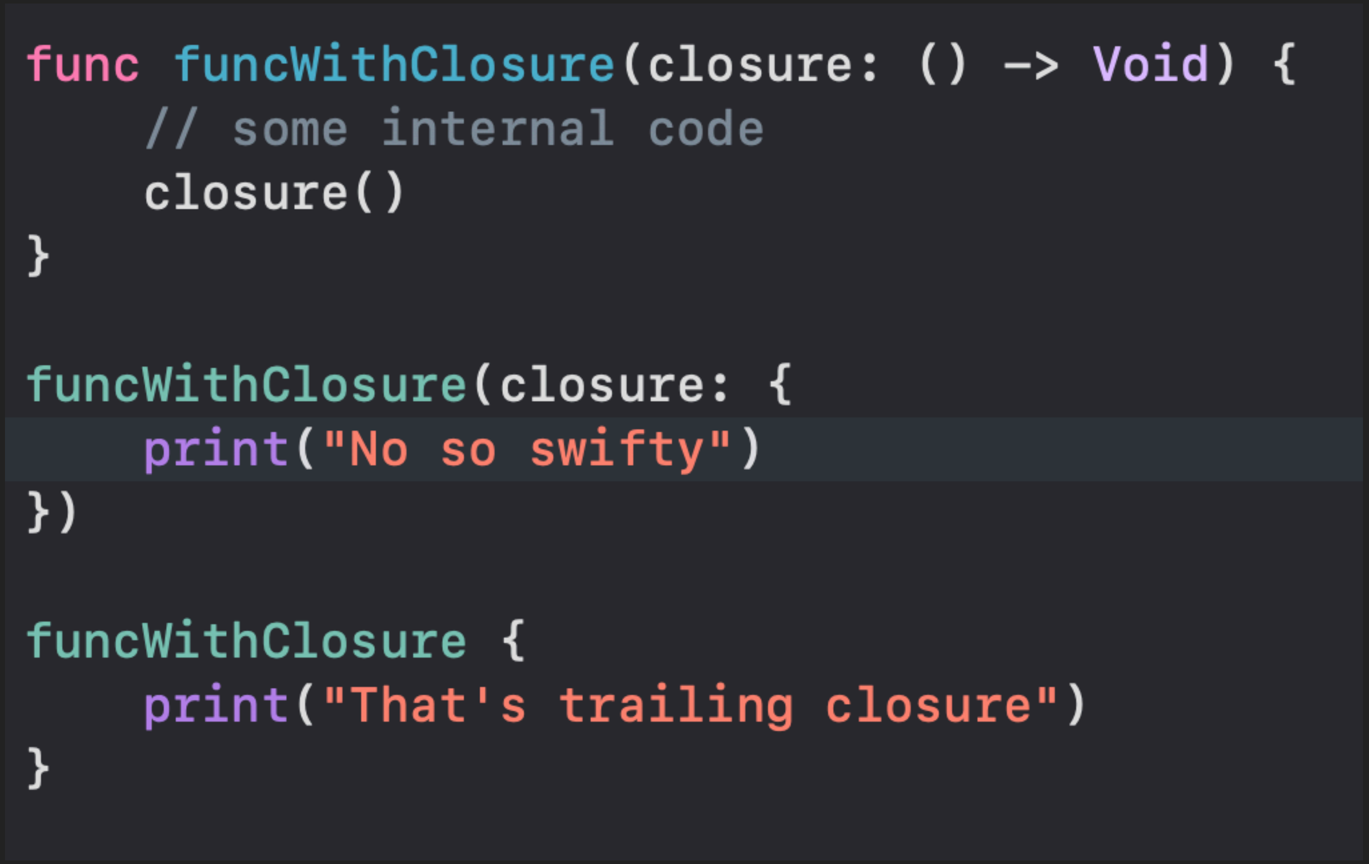
```swift
let numberStrings = ["12", "15", "34", "55"]
let numbers = numberStrings
    .compactMap { Int($0) } // convert
    .filter { $0 > 30 }     // filter biggest numbers
    .reduce(0, +)           // sum them
```



```swift
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

TRAILING CLOSURES

- If closure param is last in function we can not specify call label



```swift
func funcWithClosure(closure: () -> Void) {
    // some internal code
    closure()
}

funcWithClosure(closure: {
    print("No so swifty")
})

funcWithClosure {
    print("That's trailing closure")
}
```

Very nice used with async callbacks

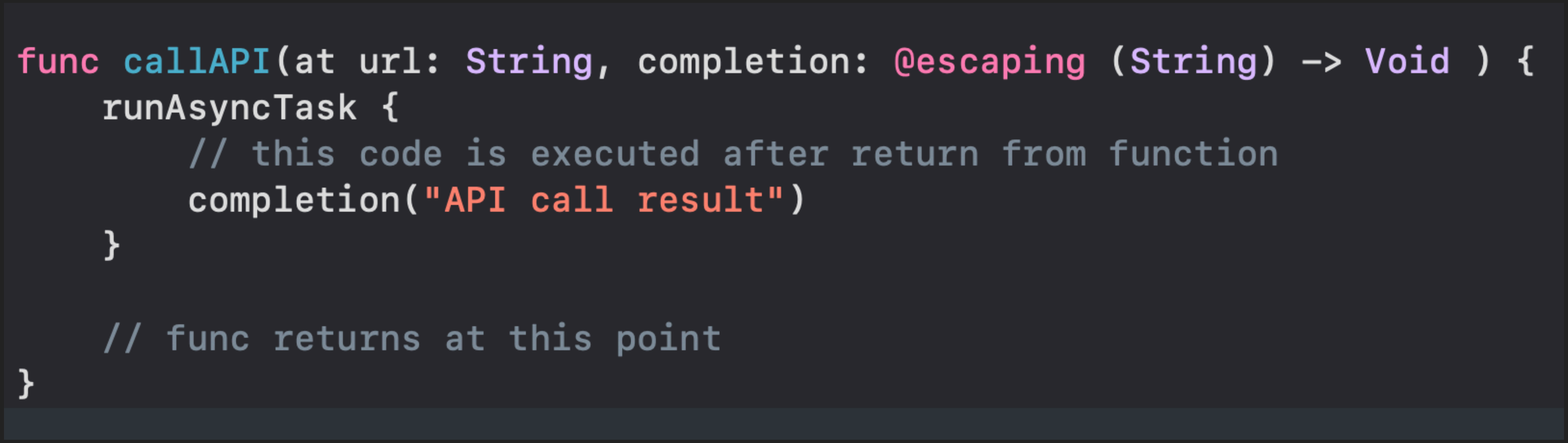
CLOSURES ARE REFERENCE TYPE

- That means they are living their own life
- Values they captured can only be changed inside their body
- They create strong reference to class that has created them (more on that later)

That stuff can create a lot of memory leaks

ESCAPING CLOSURES

- Used when closure is executed after function returns
- It can be call to server



```swift
func callAPI(at url: String, completion: @escaping (String) -> Void ) {
    runAsyncTask {
        // this code is executed after return from function
        completion("API call result")
    }

    // func returns at this point
}
```

There is also Autoclosures

NOT COVERED TOPICS

- Generics

NOT COVERED TOPICS

- Generics
- Memory management

NOT COVERED TOPICS

- Generics
- Memory management
- Opaque types

NOT COVERED TOPICS

- Generics
- Memory management
- Opaque types
- Property wrappers

NOT COVERED TOPICS

- Generics
- Memory management
- Opaque types
- Property wrappers
- Async/await