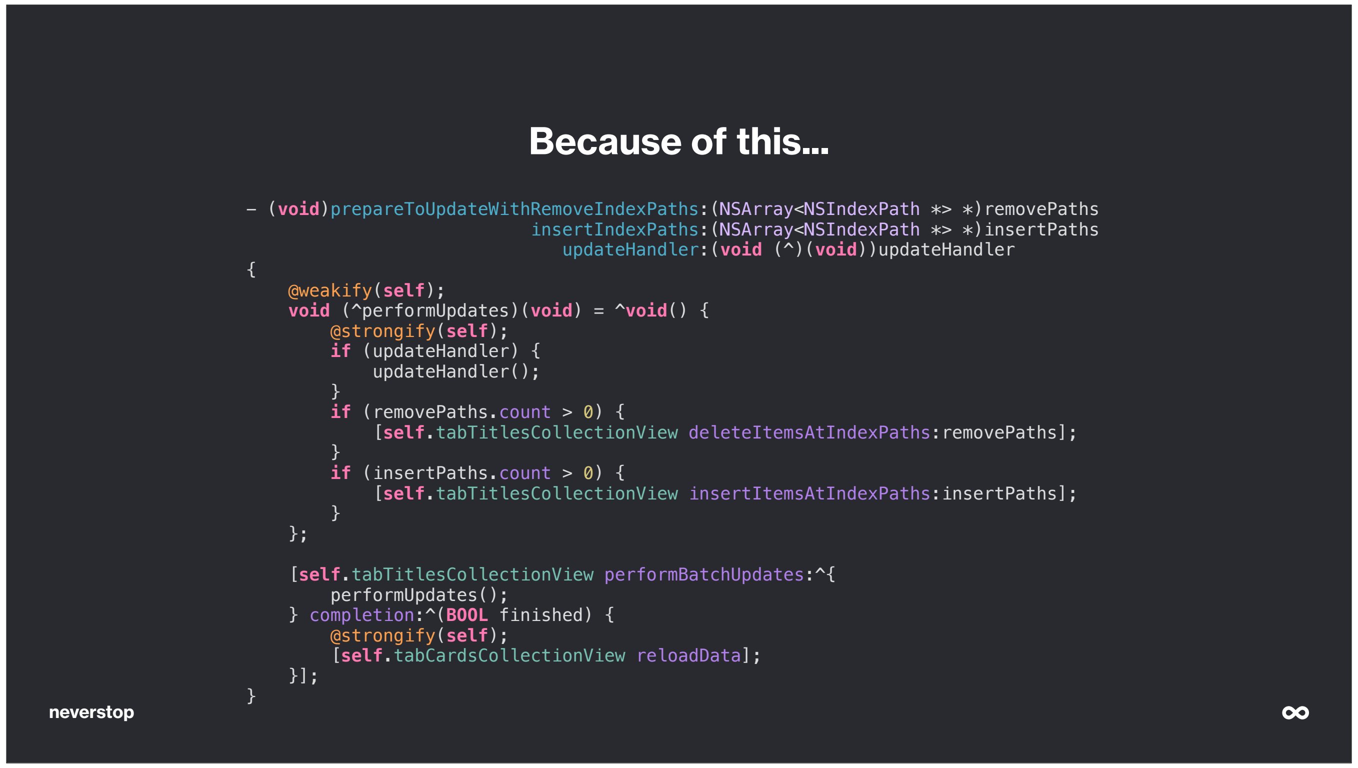
02. Swiftly about Swift

neverstop

INFINUM

01

Why Swift?



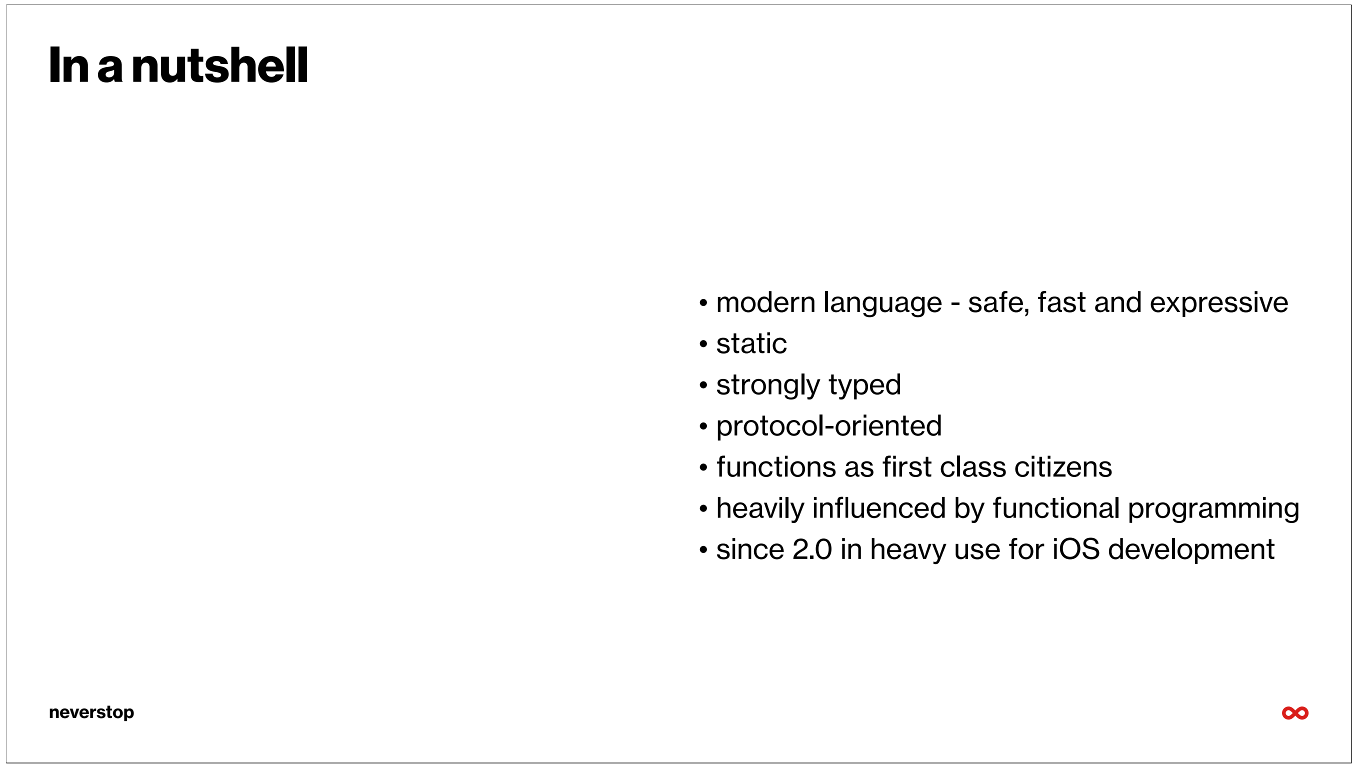
Because of this...

```
- (void)prepareToUpdateWithRemoveIndexPaths:(NSArray<NSIndexPath *> *)removePaths
                           insertIndexPaths:(NSArray<NSIndexPath *> *)insertPaths
                              updateHandler:(void (^)(void))updateHandler
{
    @weakify(self);
    void (^performUpdates)(void) = ^void() {
        @strongify(self);
        if (updateHandler) {
            updateHandler();
        }
        if (removePaths.count > 0) {
            [self.tabTitlesCollectionView deleteItemsAtIndexPaths:removePaths];
        }
        if (insertPaths.count > 0) {
            [self.tabTitlesCollectionView insertItemsAtIndexPaths:insertPaths];
        }
    };

    [self.tabTitlesCollectionView performBatchUpdates:^{
        performUpdates();
    } completion:^(BOOL finished) {
        @strongify(self);
        [self.tabCardsCollectionView reloadData];
    }];
}
```

neverstop

∞

- before Swift, Objective-C, but it's in a history
- most of iOS foundation is still written in ObjC
- previous academies, spent few weeks just learning ObjC syntax
- Swift - quick to start with, but packed with features

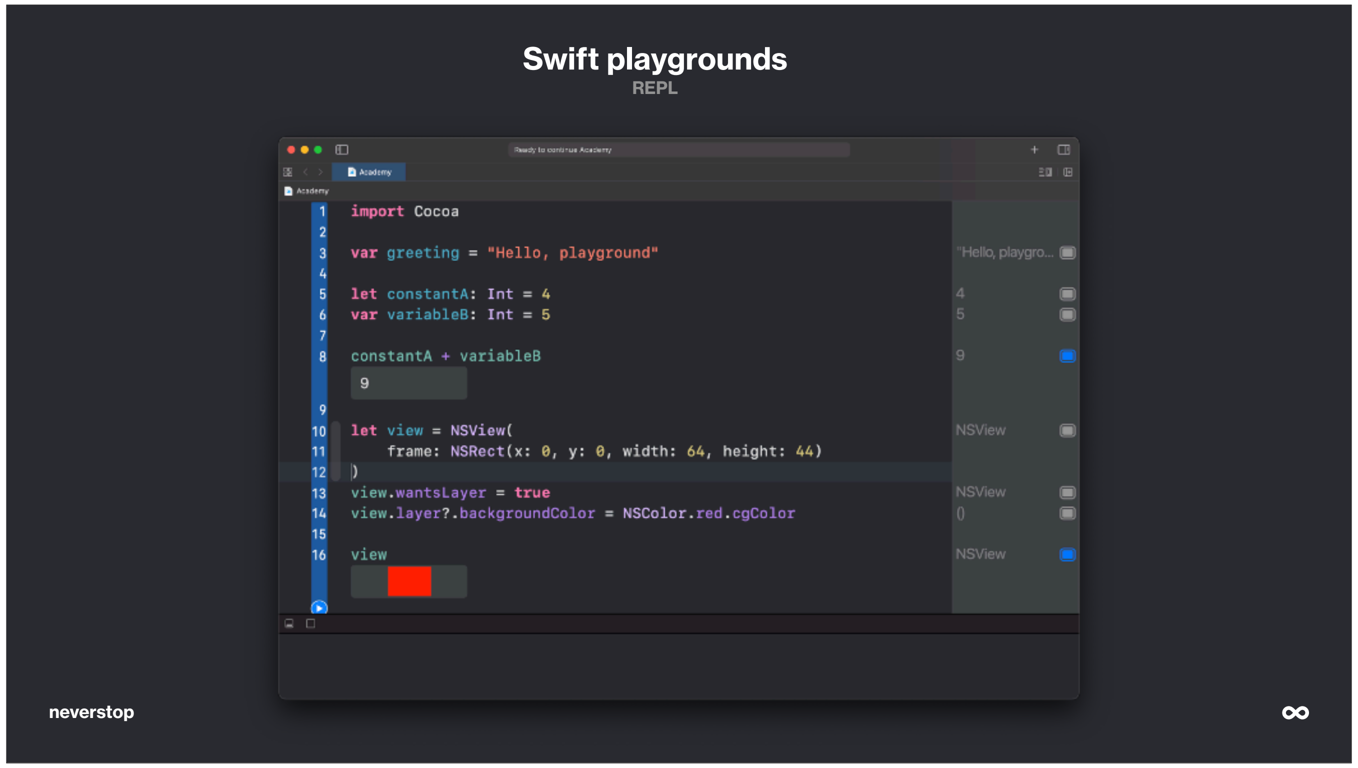


In a nutshell

- modern language - safe, fast and expressive
- static
- strongly typed
- protocol-oriented
- functions as first class citizens
- heavily influenced by functional programming
- since 2.0 in heavy use for iOS development

neverstop

- most of stuff resolved in compile time - static and strongly typed
- Apple calls it protocol oriented
- similar to interface, but composition
- functions are objects
    - reference type
    - passed, returned...
- in the beginning, writing ObjC like code in Swift - NO



Swift playgrounds

REPL

```swift
import Cocoa

var greeting = "Hello, playground"

let constantA: Int = 4
var variableB: Int = 5

constantA + variableB
9

let view = NSView(
    frame: NSRect(x: 0, y: 0, width: 64, height: 44)
)
view.wantsLayer = true
view.layer?.backgroundColor = NSColor.red.cgColor

view
```

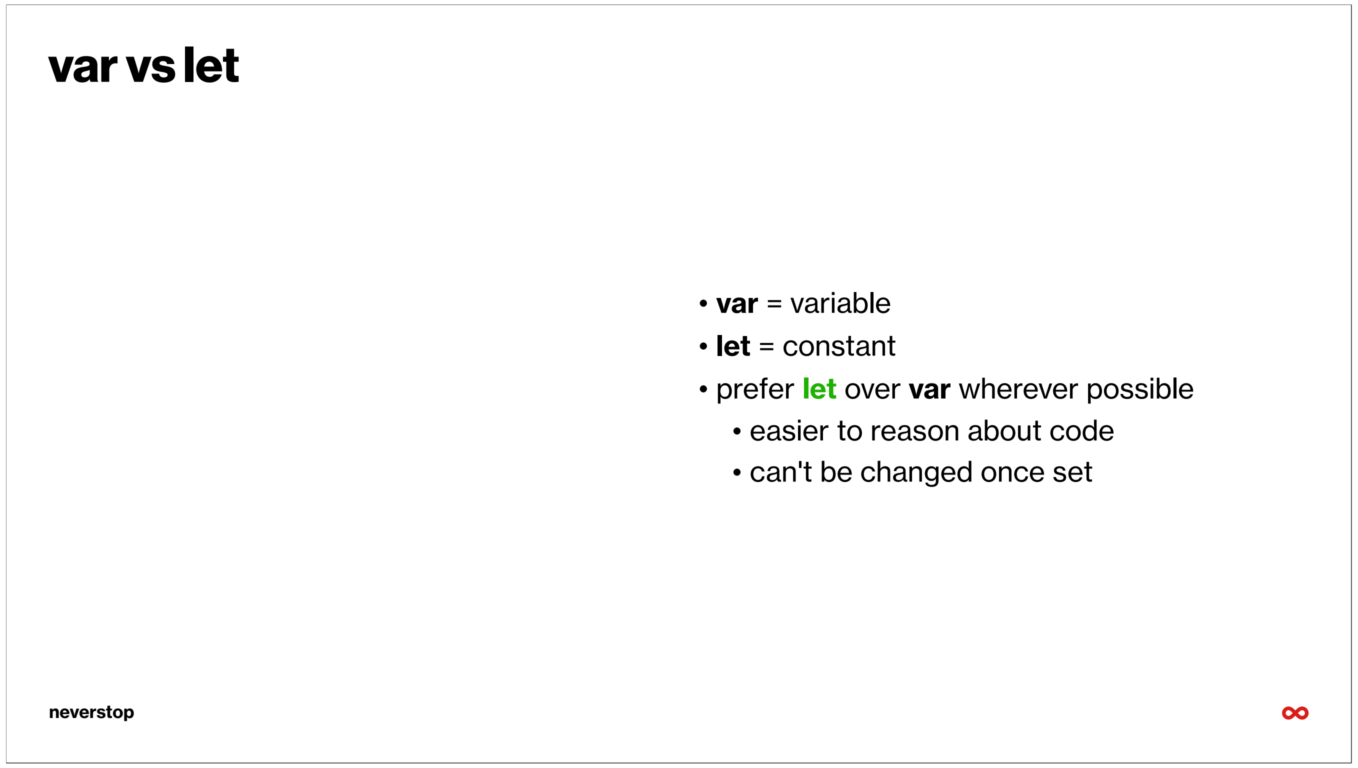
```
"Hello, playgro...
4
5
9
NSView
NSView
()
NSView
```

neverstop

- Read–eval–print loop
-

02

Basic concepts



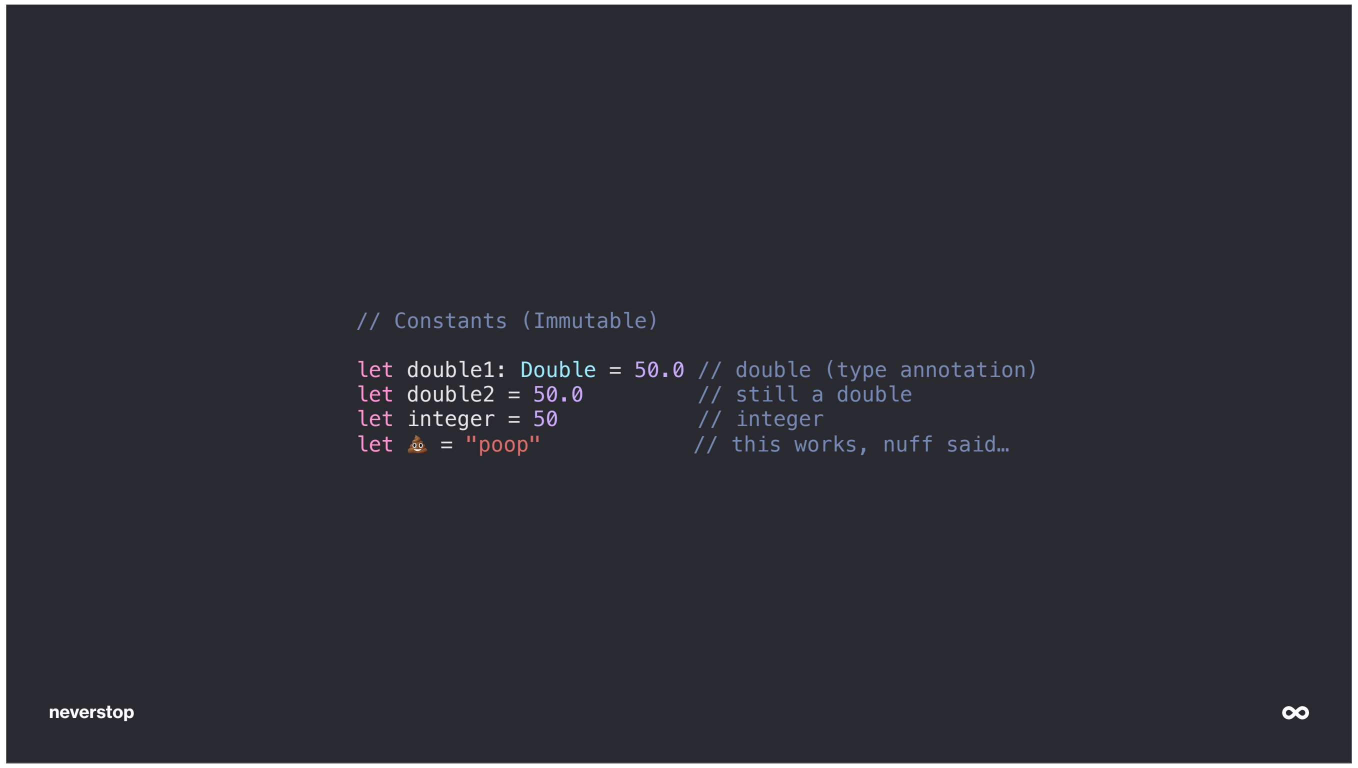
var vs let

- **var** = variable
- **let** = constant
- prefer **let** over **var** wherever possible
  - easier to reason about code
  - can't be changed once set

neverstop

- mutability modifiers
- let => a value, that WON'T change during the lifetime of the program
- var => a value, that MAY change during the lifetime of the program
- You should usually use let as much as possible
- Much more powerful than constants in C
- The value of a constant can't be changed once it's set, whereas a variable can be set to a different value in the future

- Why?
  - Well it is much easier to reason about code when you know that value can't be changed once set.
  - We will talk about this more in future…
- Example - car class
  - number of passengers
  - serial number



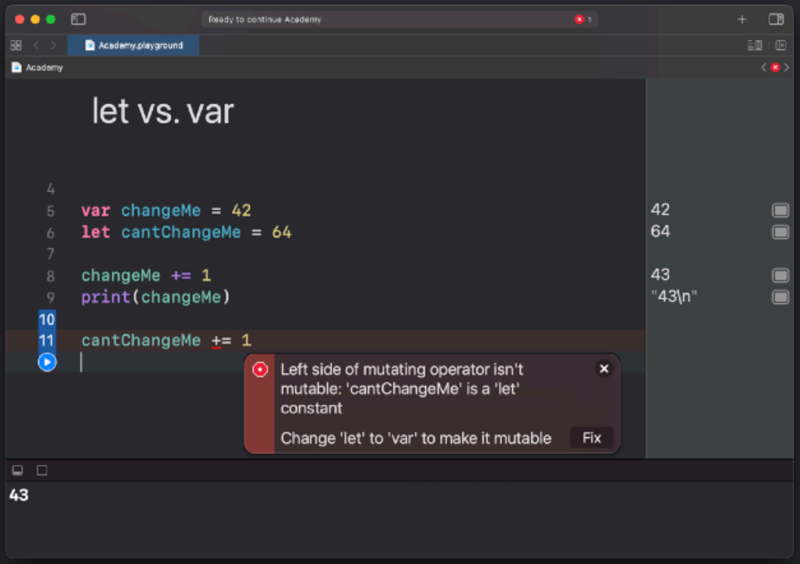
```
// Constants (Immutable)

let double1: Double = 50.0 // double (type annotation)
let double2 = 50.0         // still a double
let integer = 50           // integer
let 💩 = "poop"            // this works, nuff said…
```

neverstop

∞

- has type inference, so you basically don't need to write type annotations
- type after colon
- but sometimes you must
    - when using closures, will talk more about that later on
    - when compiler fails
- Swift supports Unicode – emoji support, but please avoid :)

let vs var



let vs. var

```
var changeMe = 42
let cantChangeMe = 64

changeMe += 1
print(changeMe)

cantChangeMe += 1
```

```
42
64
43
"43\n"
```

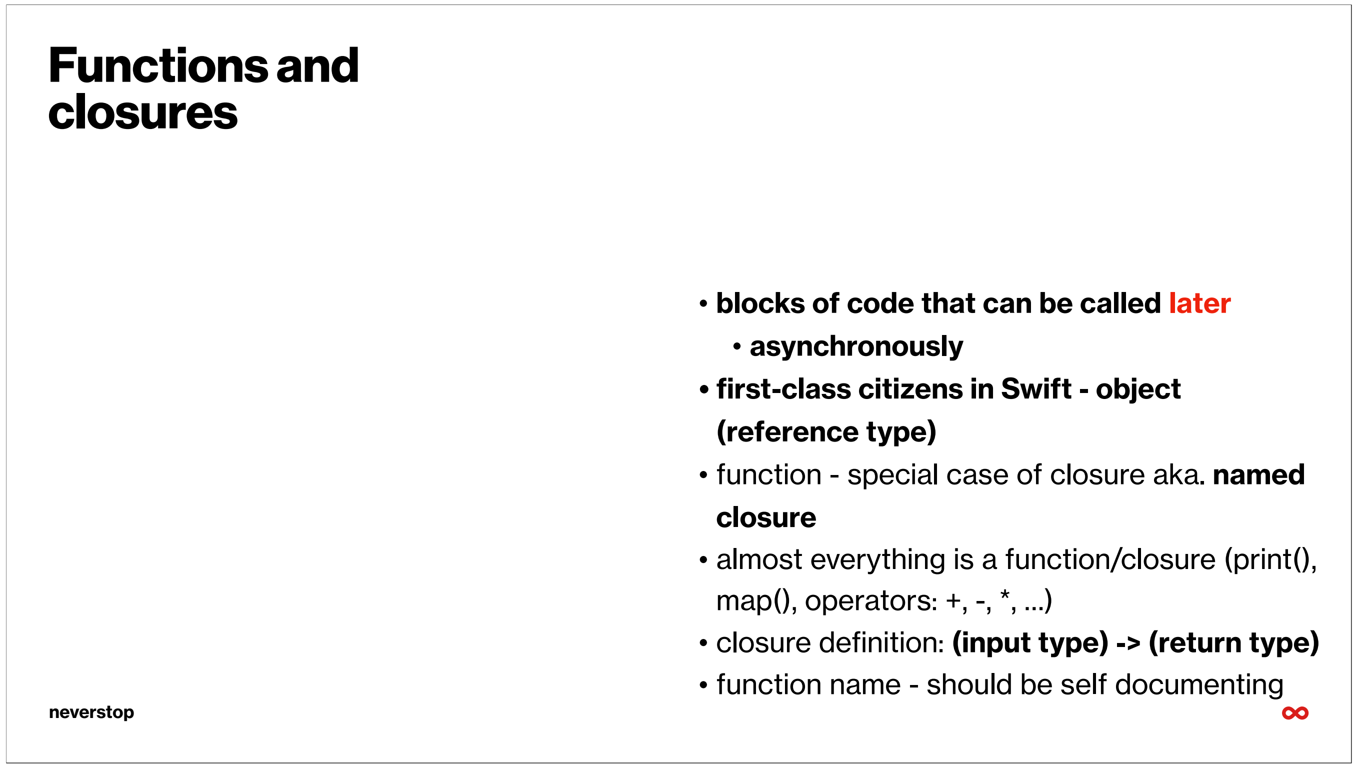
Left side of mutating operator isn't mutable: 'cantChangeMe' is a 'let' constant

Change 'let' to 'var' to make it mutable

```
43
```

neverstop

- playgrounds
- compiler will complain
- don't fret - you'll get better grasp when start coding



Functions and closures

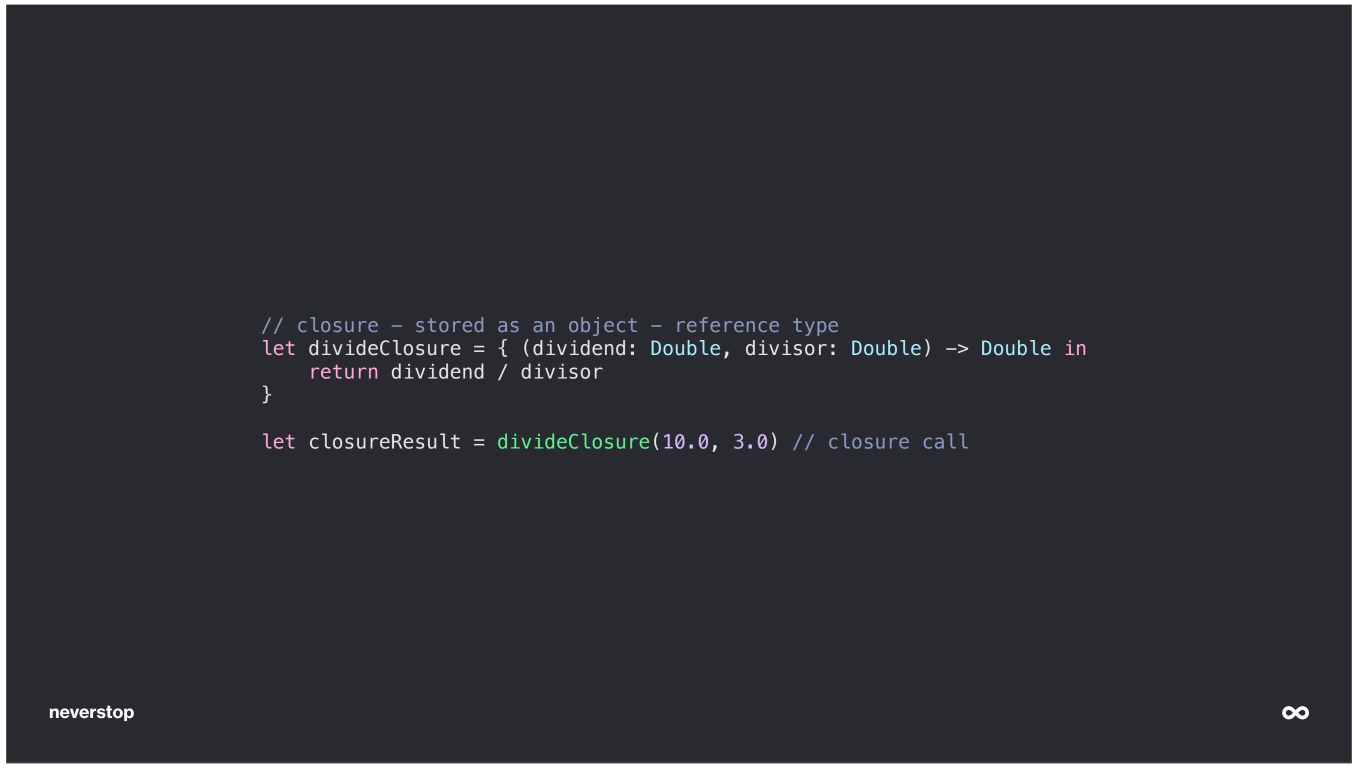
- **blocks of code that can be called later**
  - **asynchronously**
- **first-class citizens in Swift - object (reference type)**
- function - special case of closure aka. **named closure**
- almost everything is a function/closure (print(), map(), operators: +, -, *, ...)
- closure definition: **(input type) -> (return type)**
- function name - should be self documenting

neverstop

Closures are just un-named or anonymous functions – **WRONG**, but easier to explain for someone who first time sees closure

Closures are also functions. But when a function captures state upon its creation, we call it a closure – CORRECT, but it requires some knowledge on Swift internals

- similar to anonymous functions/lambdas in Java & JavaScript
- operators are functions as well
  - improved safety and resolved ambiguity
  - Double(2.3) + Int(2) = ?
- functions use preposition
  - removeItem(at index: Int)

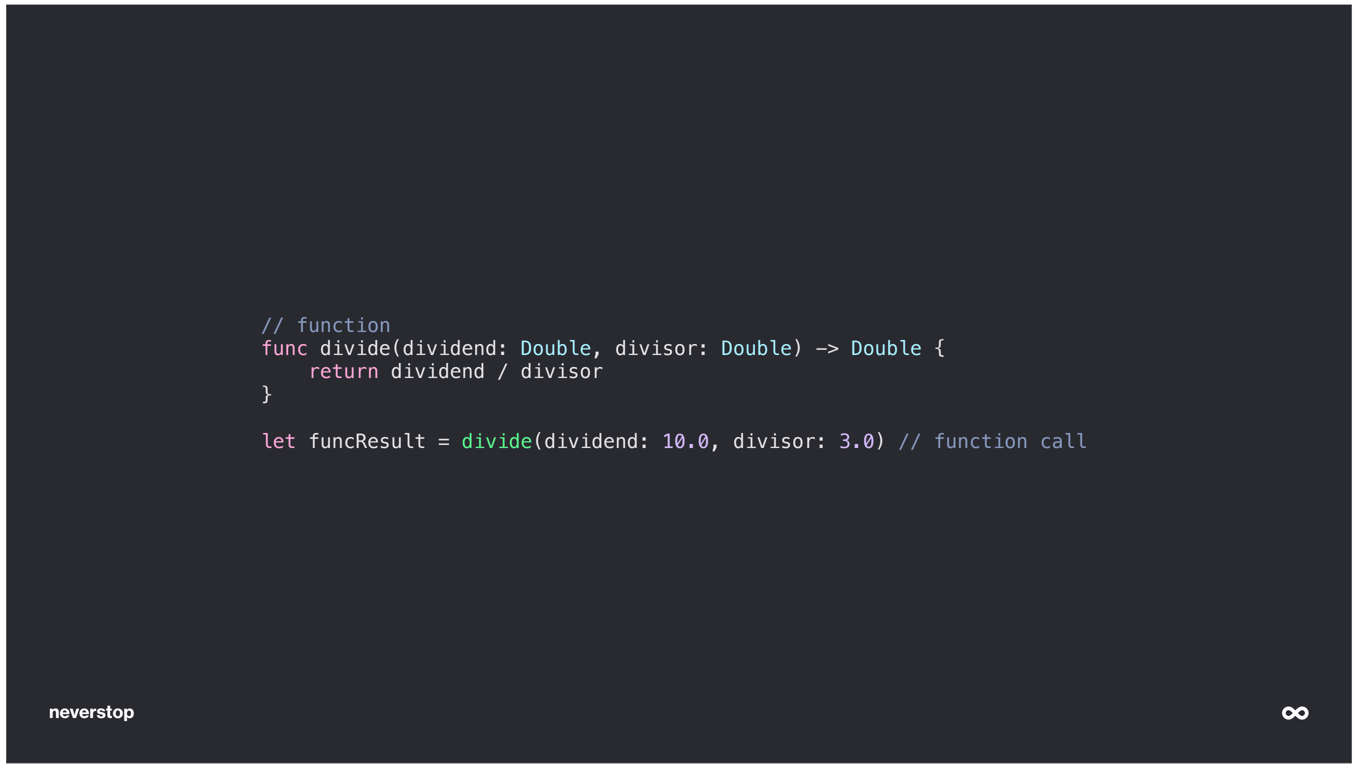


```
// closure - stored as an object - reference type
let divideClosure = { (dividend: Double, divisor: Double) -> Double in
    return dividend / divisor
}

let closureResult = divideClosure(10.0, 3.0) // closure call
```

neverstop

∞

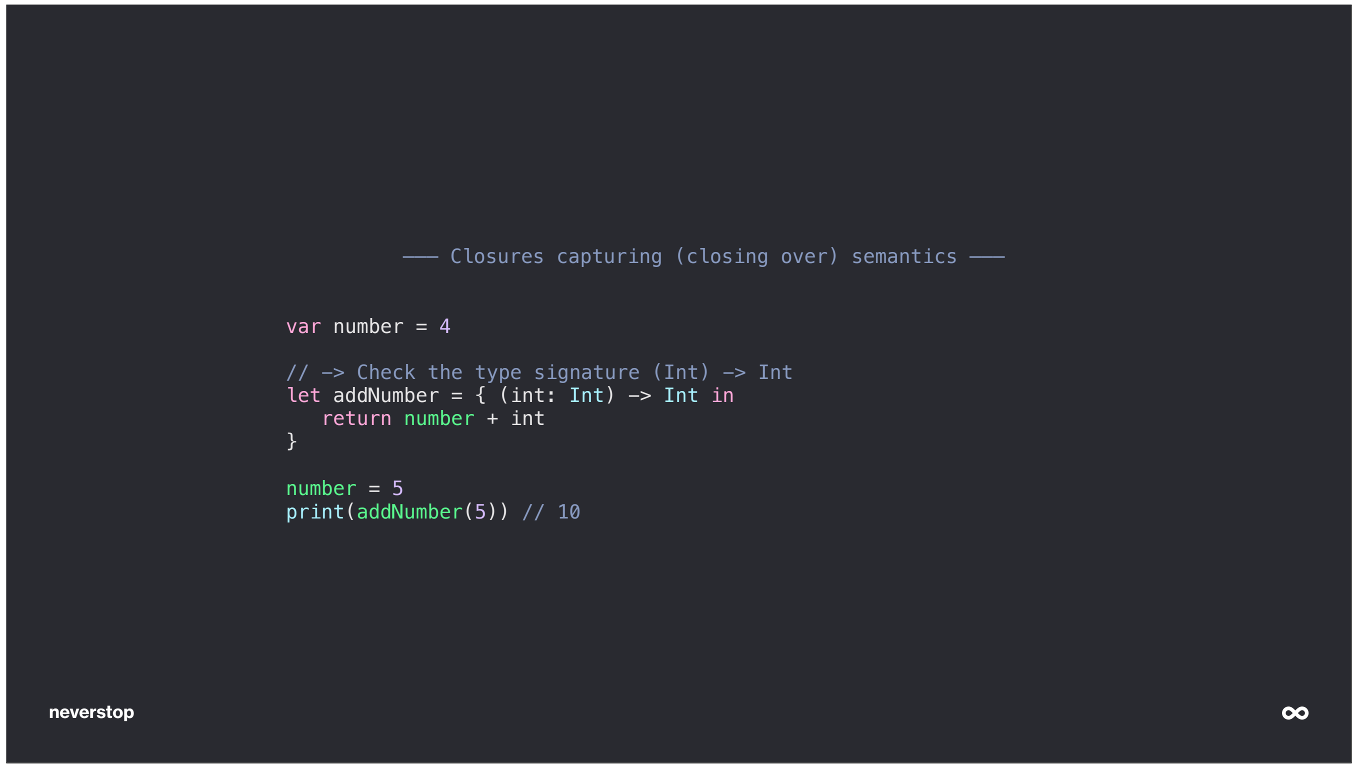
- self documenting methods, named parameters
- closure - same thing as object, stored, evaluated later - asynchronously



```
// function
func divide(dividend: Double, divisor: Double) -> Double {
    return dividend / divisor
}

let funcResult = divide(dividend: 10.0, divisor: 3.0) // function call
```

neverstop ∞

- self documenting methods, named parameters
- closure - same thing as object, stored, evaluated later - asynchronously
- argument names are part of the function name
- self-documenting



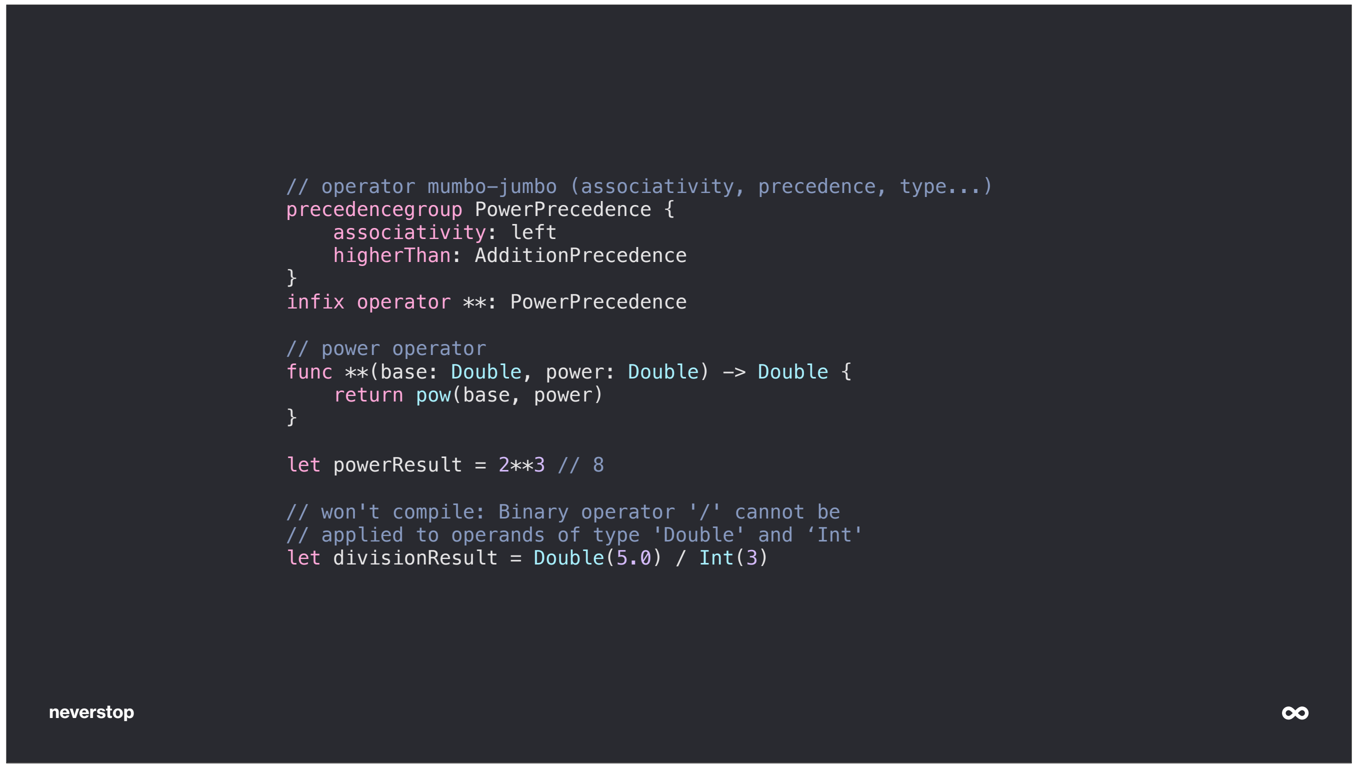
—— Closures capturing (closing over) semantics ——

```
var number = 4

// -> Check the type signature (Int) -> Int
let addNumber = { (int: Int) -> Int in
   return number + int
}

number = 5
print(addNumber(5)) // 10
```

neverstop

∞

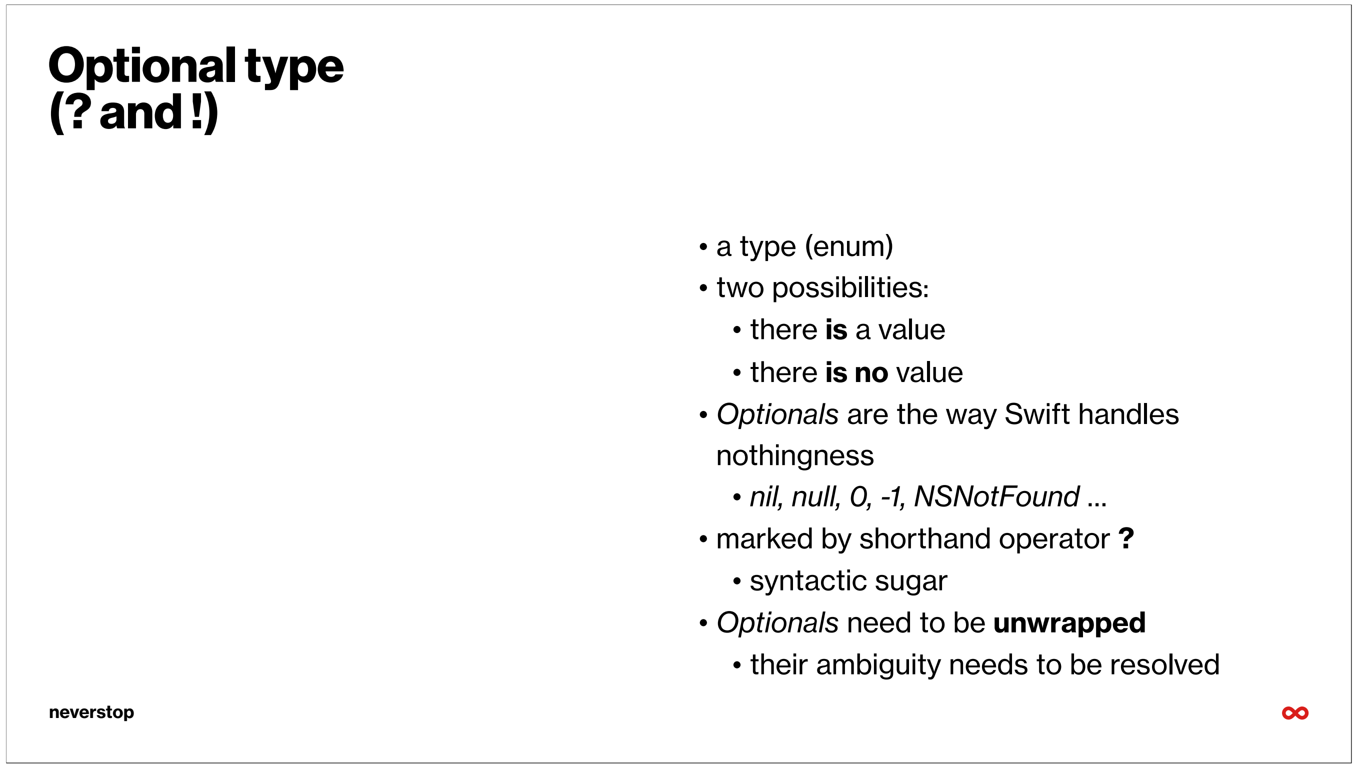
- difference between closure and function
- capture semantics - closure captures variable from definition scope - like anonymous class in Java!
- asynchronously executed
- example: Dealing with API call - refreshing data - store reference to view and refresh it when API call is done



```swift
// operator mumbo-jumbo (associativity, precedence, type...)
precedencegroup PowerPrecedence {
    associativity: left
    higherThan: AdditionPrecedence
}
infix operator **: PowerPrecedence

// power operator
func **(base: Double, power: Double) -> Double {
    return pow(base, power)
}

let powerResult = 2**3 // 8

// won't compile: Binary operator '/' cannot be
// applied to operands of type 'Double' and ‘Int'
let divisionResult = Double(5.0) / Int(3)
```

neverstop

- Almost everything is a function
- even operators are functions
- each operator is defined for specific combination of types
    - function which takes only predefined arguments Double/Double, Int/Int
    - leads to expressive language, less error prone
    - what is the result of division of an Int and Double?
    - you have to cast to wanted type



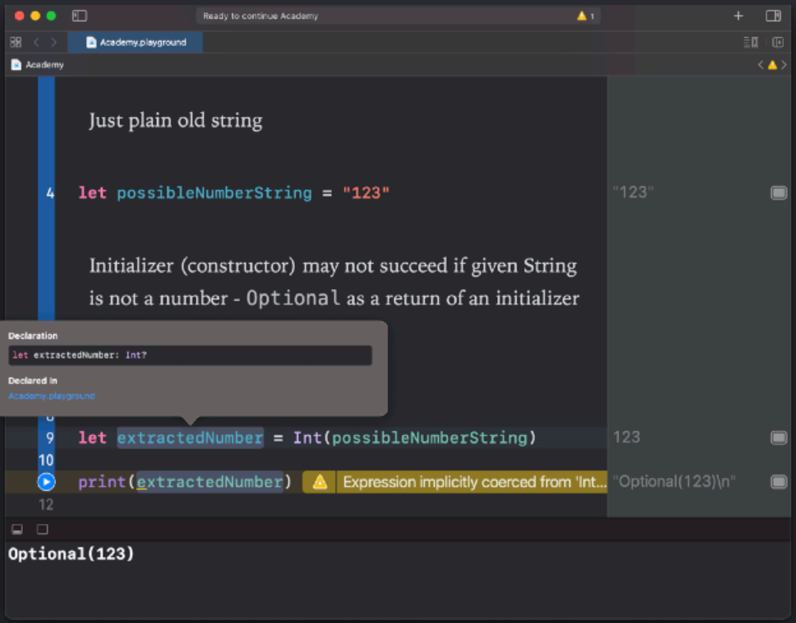
Optional type (? and !)

- a type (enum)
- two possibilities:
  - there **is** a value
  - there **is no** value
- *Optionals* are the way Swift handles nothingness
  - *nil, null, 0, -1, NSNotFound* ...
- marked by shorthand operator **?**
  - syntactic sugar
- *Optionals* need to be **unwrapped**
  - their ambiguity needs to be resolved

neverstop

- "Maybe" type in Haskell
- You want to reduce the usage of Optionals in your codebase as that will add a layer of complexity
- "No more NullPointerExceptions"
- another safety feature
- impossible to send Optional value to function which expect non-Optional

Optionals



Just plain old string

```
let possibleNumberString = "123"
```

"123"

Initializer (constructor) may not succeed if given String is not a number - `Optional` as a return of an initializer

Declaration

```
let extractedNumber: Int?
```

Declared In

Academy.playground

```
let extractedNumber = Int(possibleNumberString)

print(extractedNumber)
```

Expression implicitly coerced from 'Int...

123

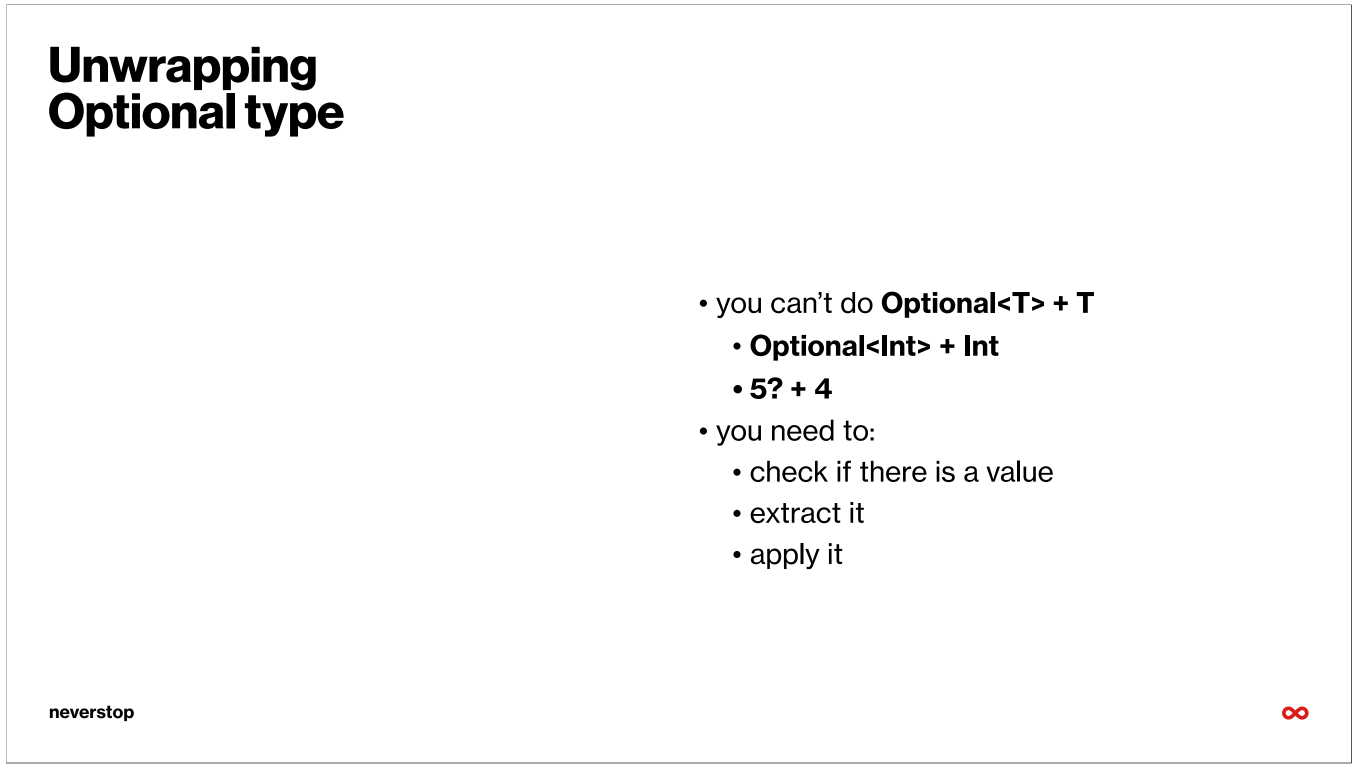
"Optional(123)\n"

```
Optional(123)
```

neverstop

∞

- Optional is a TYPE, so it is not value or absence of value.
- Optional is type (enum) which can contain value
- This is one of the reason why Swift can help you catch most of NULL pointer exception bugs during compile time
- Initializer – allocates memory and sets initial values

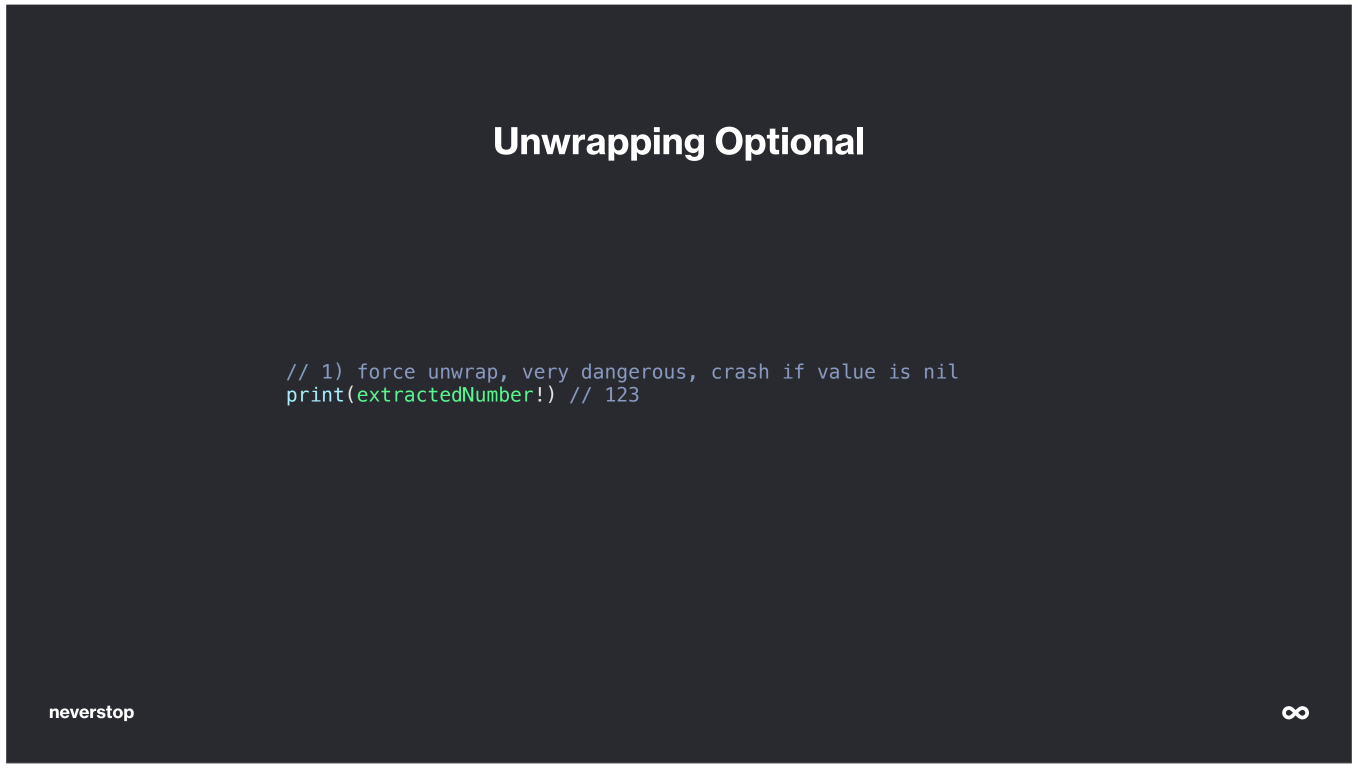


Unwrapping Optional type

- you can't do **Optional<T> + T**
  - **Optional<Int> + Int**
  - **5? + 4**
- you need to:
  - check if there is a value
  - extract it
  - apply it

neverstop

- You can't "apply" anything to Optional, you need to extract the value if any
- A bit cumbersome
- A day to day activity
- In the next couple of slides, we will show you how



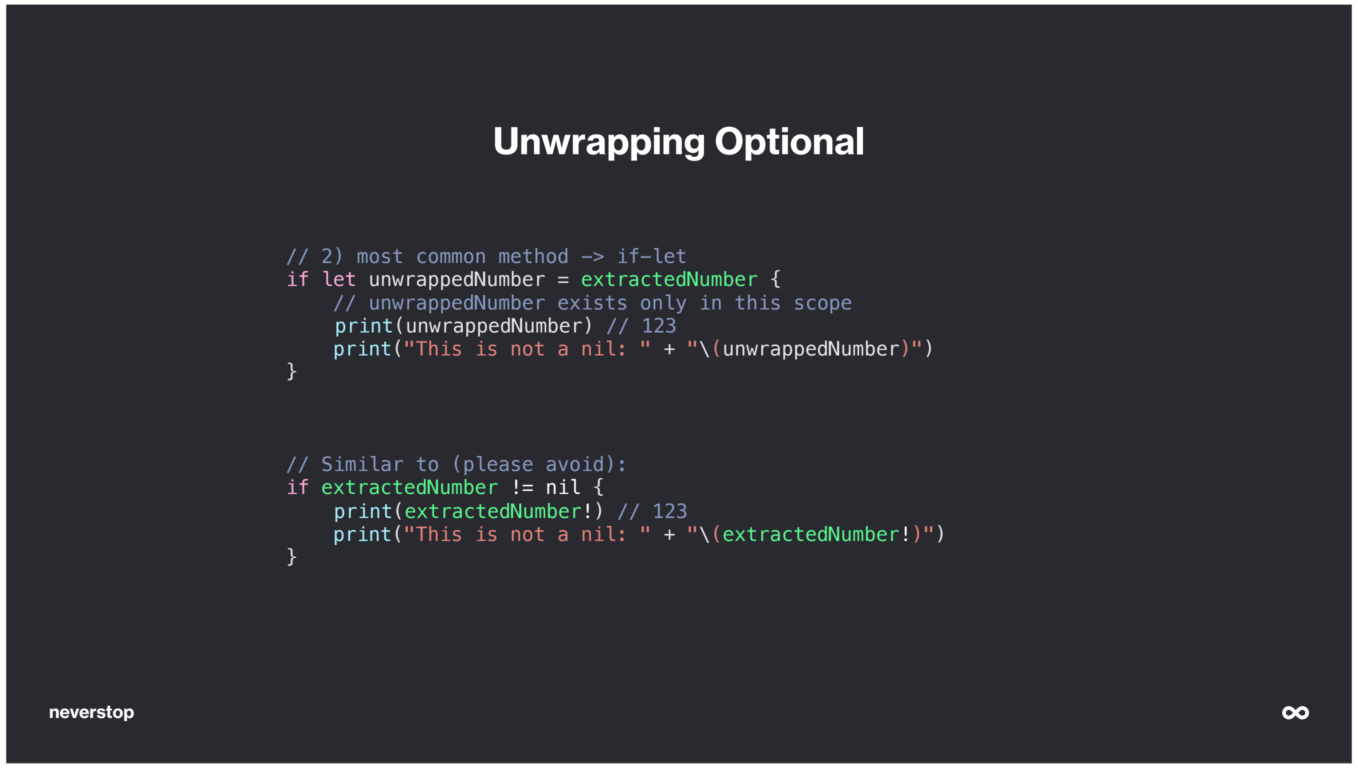
Unwrapping Optional

```
// 1) force unwrap, very dangerous, crash if value is nil
print(extractedNumber!) // 123
```

neverstop

∞

- force unwrapping
  - worst kind of unwrapping
  - most destructive one, you are effectively going around the compiler
  - it will crash if nil, use only when sure, I don’t suggest you use it during this course
  - used rarely

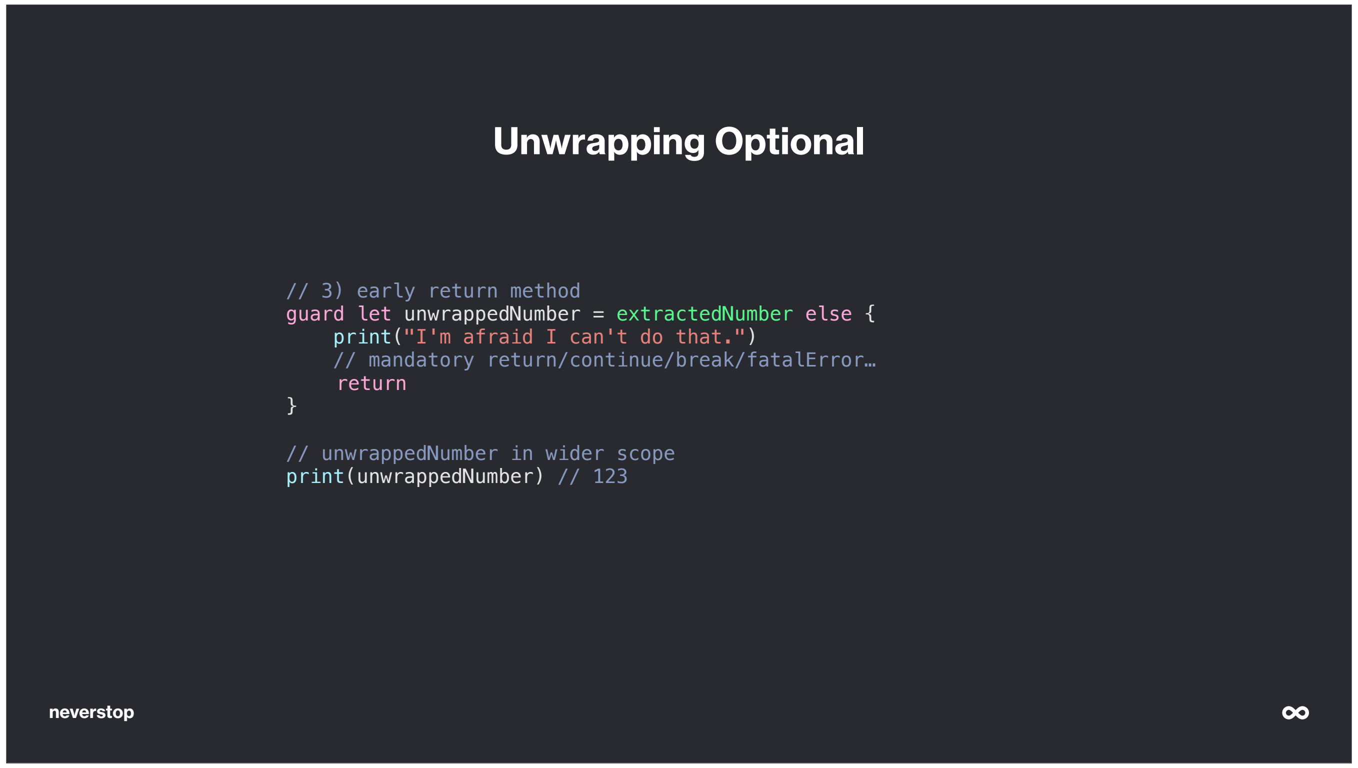


Unwrapping Optional

```
// 2) most common method -> if-let
if let unwrappedNumber = extractedNumber {
    // unwrappedNumber exists only in this scope
    print(unwrappedNumber) // 123
    print("This is not a nil: " + "\(unwrappedNumber)")
}


// Similar to (please avoid):
if extractedNumber != nil {
    print(extractedNumber!) // 123
    print("This is not a nil: " + "\(extractedNumber!)")
}
```

neverstop

- most common unwrapping method, similar to check
    - if convertedNumber != nil {
        - // do stuff
    - }
- compiler forces you to think about else case
- no NullPointerException
- will generate more code, but less error-prone
- second case - compiler won't warn you if someone deletes if check



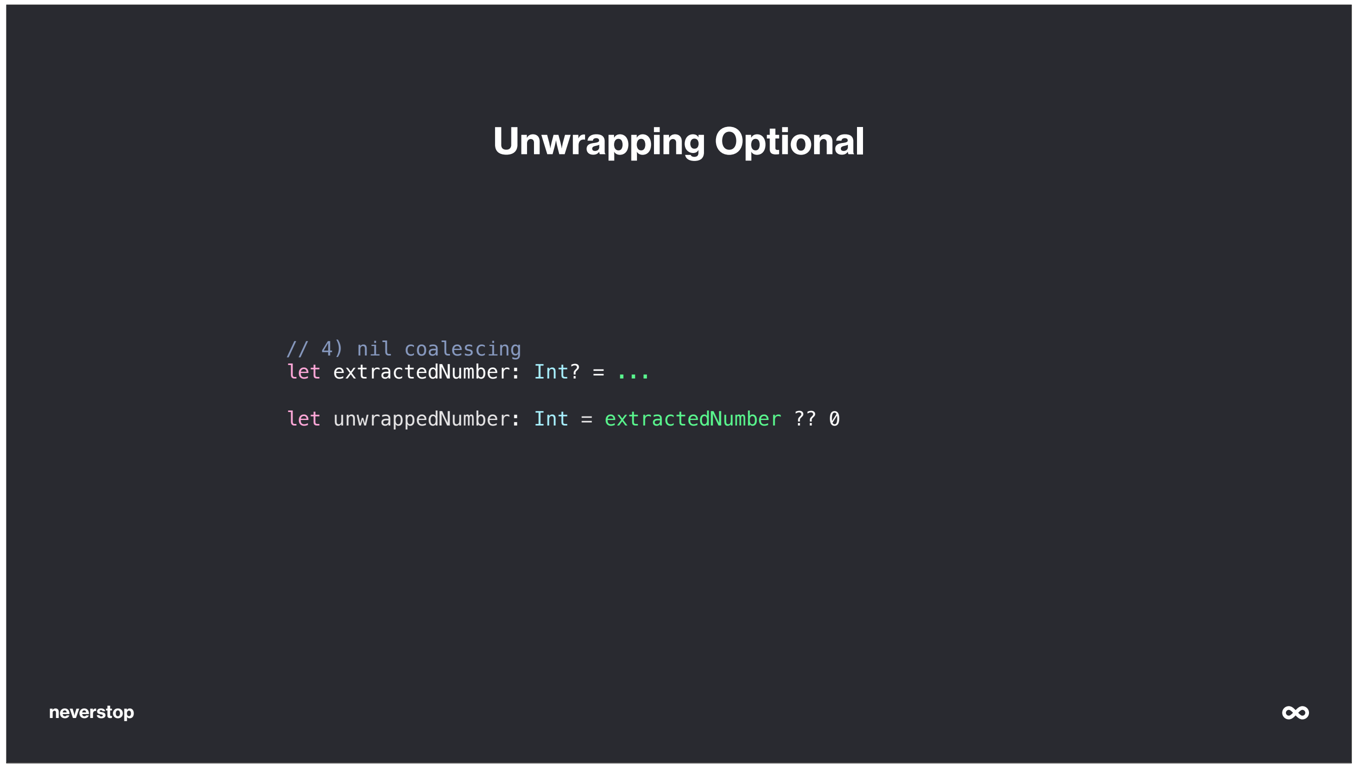
Unwrapping Optional

```swift
// 3) early return method
guard let unwrappedNumber = extractedNumber else {
    print("I'm afraid I can't do that.")
    // mandatory return/continue/break/fatalError…
     return
}

// unwrappedNumber in wider scope
print(unwrappedNumber) // 123
```

neverstop

∞

- second most common, use when you want early return
- it will allow you to write less indented code
- avoiding pyramid of doom
- you always need to break in else clause, so it won't enter scope after it since value will not exist - compile error
- won't indent whole function
- mostly used on top of the function
    - avoid in middle since it disrupts control flow, debugging and code understanding



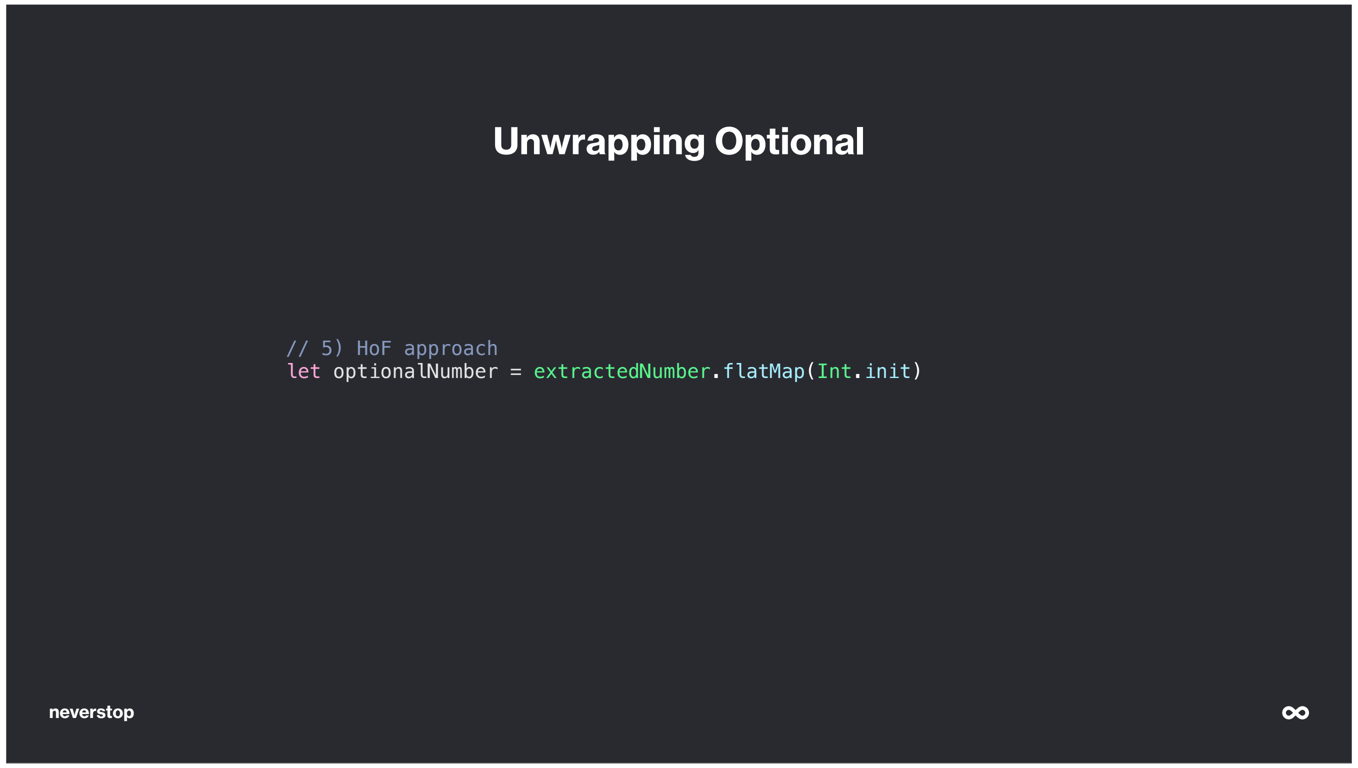
Unwrapping Optional

```
// 4) nil coalescing
let extractedNumber: Int? = ...

let unwrappedNumber: Int = extractedNumber ?? 0
```

neverstop

∞

- Similar to ternary operator

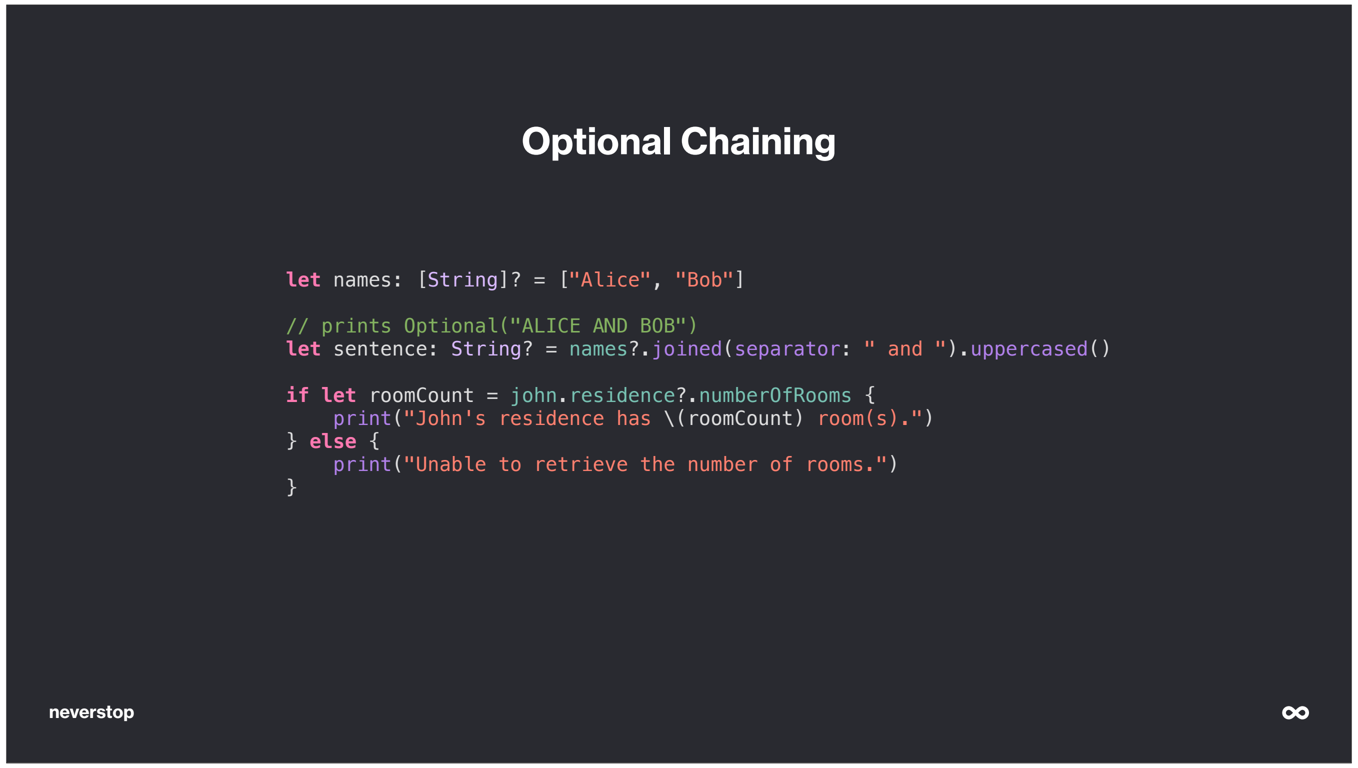


Unwrapping Optional

```
// 5) HoF approach
let optionalNumber = extractedNumber.flatMap(Int.init)
```

neverstop ∞

- we actually use this one a lot, it relies on higher order function called *map*
- you can use this approach if you just want to apply something to Optional type but still keep it’s context (e.g. Optional)



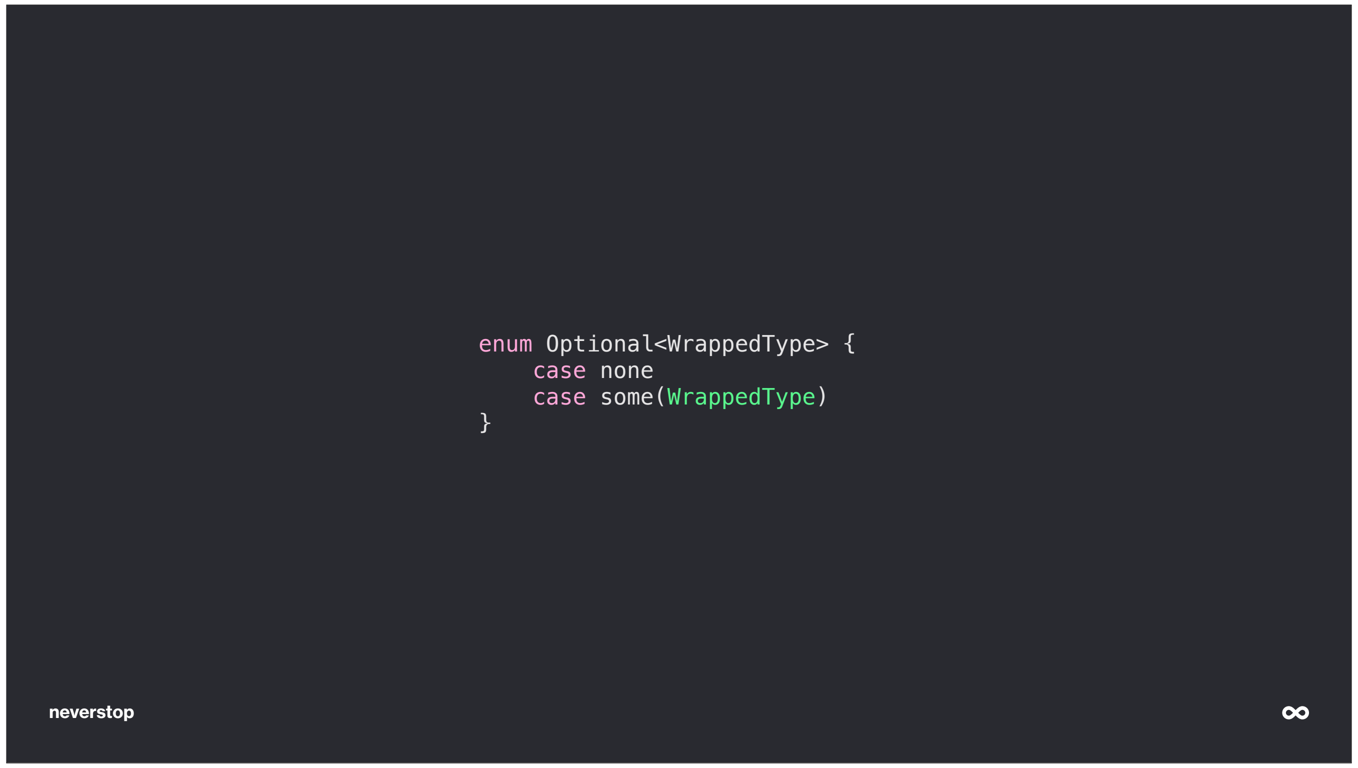
Optional Chaining

```swift
let names: [String]? = ["Alice", "Bob"]

// prints Optional("ALICE AND BOB")
let sentence: String? = names?.joined(separator: " and ").uppercased()

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
```

neverstop

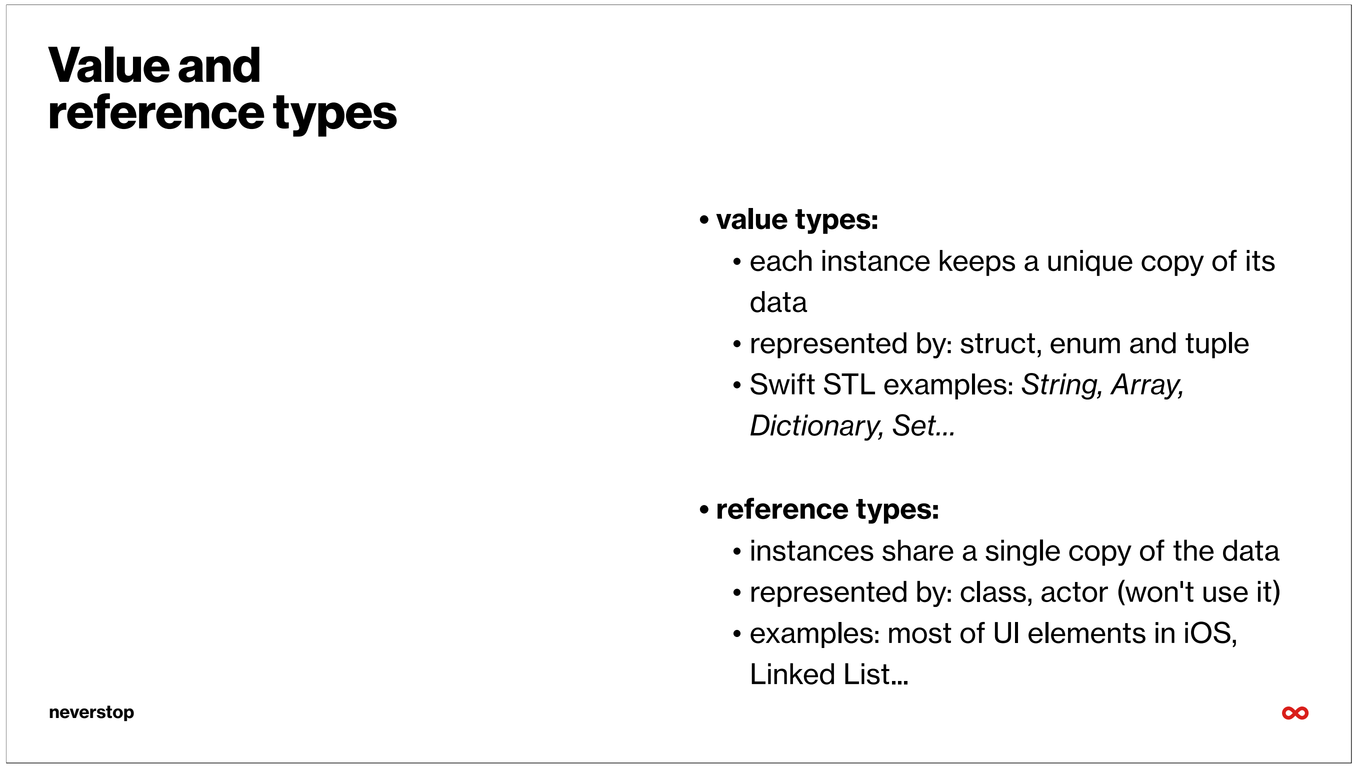
∞

- process for querying and calling properties, methods on an optional
- If the optional contains a value, returns a value
- if the optional is nil, the call returns nil
- multiple queries can be chained together
    - entire chain fails gracefully if any link in the chain is nil.



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

- just a peek how Optional is implemented in Swift

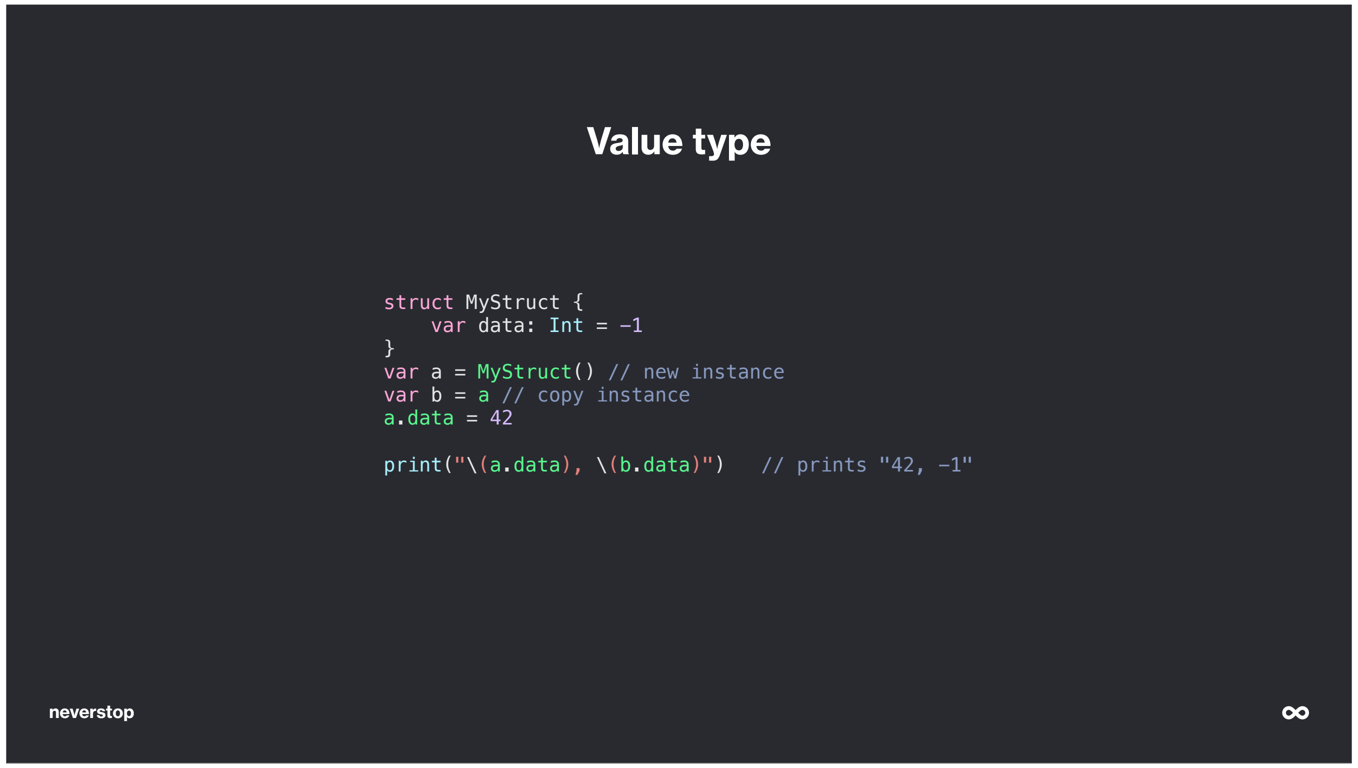


Value and reference types

- **value types:**
  - each instance keeps a unique copy of its data
  - represented by: struct, enum and tuple
  - Swift STL examples: *String, Array, Dictionary, Set…*
- **reference types:**
  - instances share a single copy of the data
  - represented by: class, actor (won't use it)
  - examples: most of UI elements in iOS, Linked List…

neverstop

- primitives are similar to value types
- but complex types can be value types - array, dictionary...
  - copy on write and backed storage
- each value type has its own copy of the data
  - value types get copied when changed, thus not changing original value



Value type

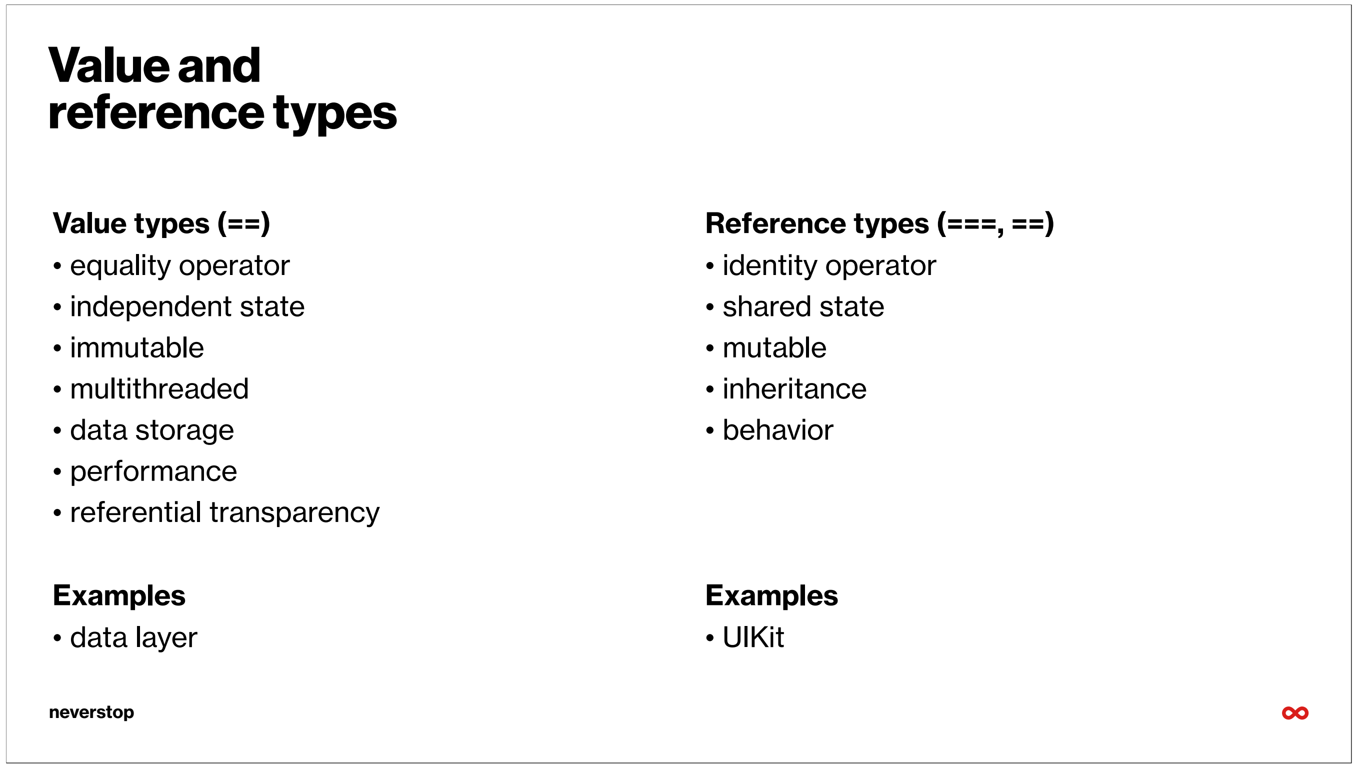
```
struct MyStruct {
    var data: Int = -1
}
var a = MyStruct() // new instance
var b = a // copy instance
a.data = 42

print("\(a.data), \(b.data)")   // prints "42, -1"
```

neverstop

- So from the slides above you can deduce that each time assignment is made on value types, copying will take place. That is not entirely correct
- To improve performance, there is a neat trick called CopyOnWrite, which will not do copy on assignment but rather when you start modifying it

Reference type

```
class MyClass {
    var data: Int = -1
}
var x = MyClass() // new instance
var y = x // copy reference to instance
x.data = 42

print("\(x.data), \(y.data)")   // prints "42, 42"
```



Value and reference types

Value types (==)
- equality operator
- independent state
- immutable
- multithreaded
- data storage
- performance
- referential transparency

Reference types (===, ==)
- identity operator
- shared state
- mutable
- inheritance
- behavior

Examples
- data layer

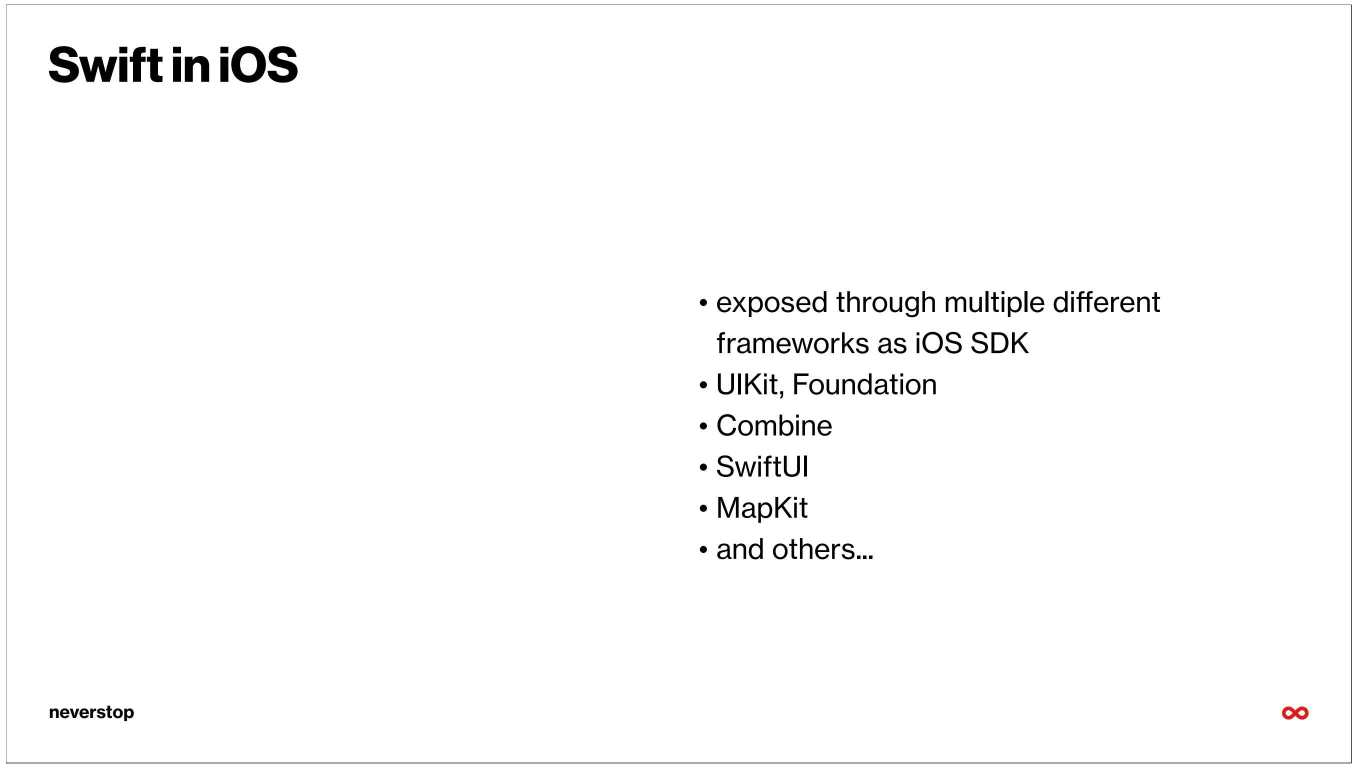
Examples
- UIKit

neverstop

- UIKit works mostly with objects aka reference types
- Object has a behaviour, while value type is inert
- Value types for data models, basic representation
- Value type is allocated on stack, reference type on heap, stack pointer movement away
- Value type much faster access read & write
- Referential Transparency – easier to reason about the logic
- Value types – thread safe

03

Swift in iOS

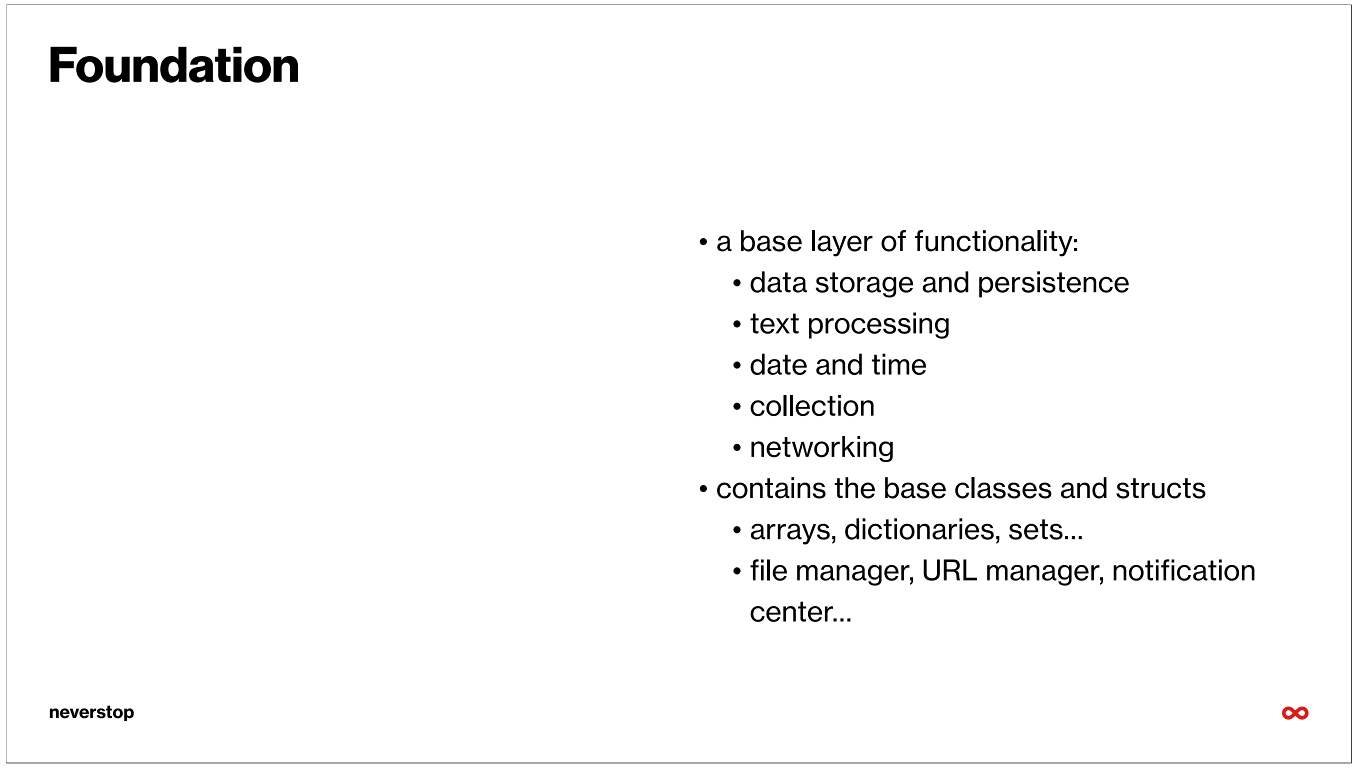


Swift in iOS

- exposed through multiple different frameworks as iOS SDK
- UIKit, Foundation
- Combine
- SwiftUI
- MapKit
- and others…

neverstop

- UIKit & Foundation the most common
- reactive programming
- declarative UI

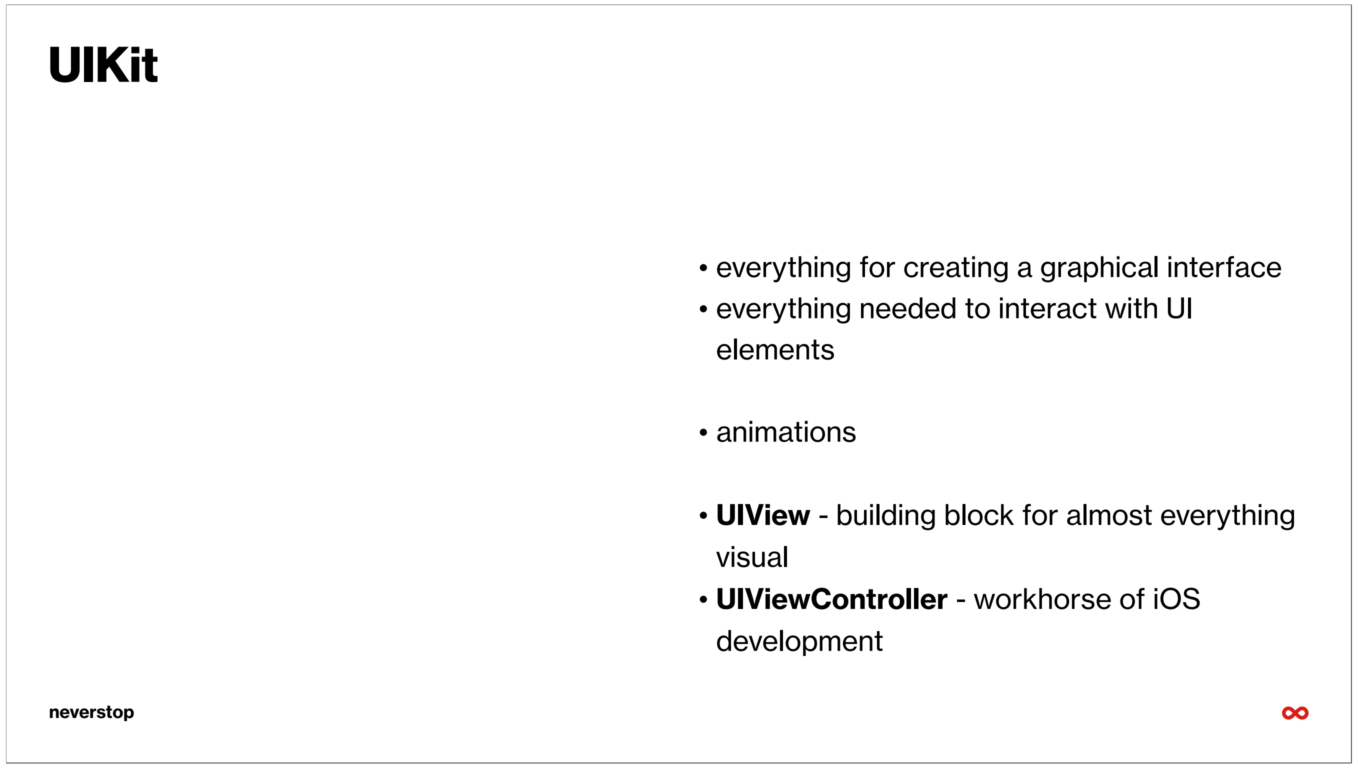


Foundation

- a base layer of functionality:
  - data storage and persistence
  - text processing
  - date and time
  - collection
  - networking
- contains the base classes and structs
  - arrays, dictionaries, sets...
  - file manager, URL manager, notification center...

neverstop

- some parts of Foundation are still written in ObjC
- but Apple made great interoperable support
  - you won't even notice that you are using ObjC



UIKit

- everything for creating a graphical interface
- everything needed to interact with UI elements

- animations

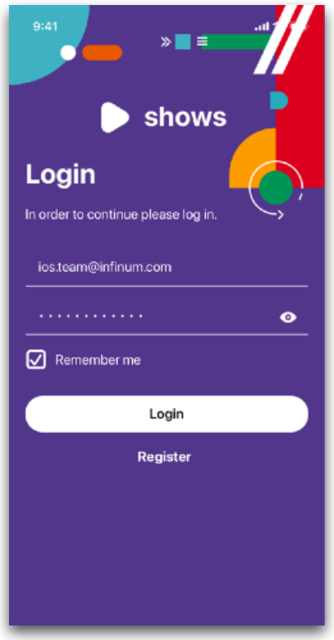
- **UIView** - building block for almost everything visual
- **UIViewController** - workhorse of iOS development

neverstop

∞

- UI development in iOS
- UIView <-> Base Object
- everything UI inherits or contains UIView

UIKit



- **UIView**
- **UIImageView**
- **UIButton**
- **UITextField**
- **UITableView**, **UICollectionView**, **UIScrollView** and many others

neverstop

- UIViewController
-

04

Home assignment

Home assignment

Read until next lecture

1. https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html
2. https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html
3. https://docs.swift.org/swift-book/LanguageGuide/Closures.html

05

Appendix

Links

- **Swift**
  - https://swift.org/
- **Documentation**
  - https://swift.org/documentation/
- **Books (app on your MacBook)**
  - App Development with Swift
  - Create Apple ID (free) and log in to Books app
- **Guides**
  - https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html
  - https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html
  - https://docs.swift.org/swift-book/LanguageGuide/Closures.html

Lecture recordings

- https://us02web.zoom.us/rec/share/ZAs5LEOg6IYmuki9FjAqZu0yEewpqIjdqvpl5ExyitQzm6h2GW9al-A4WkoJfJ11.m7ywE6FsEACuCoiY

neverstop

Thanks!

neverstop

Linkedin Instagram Twitter Facebook

∞ INFINUM