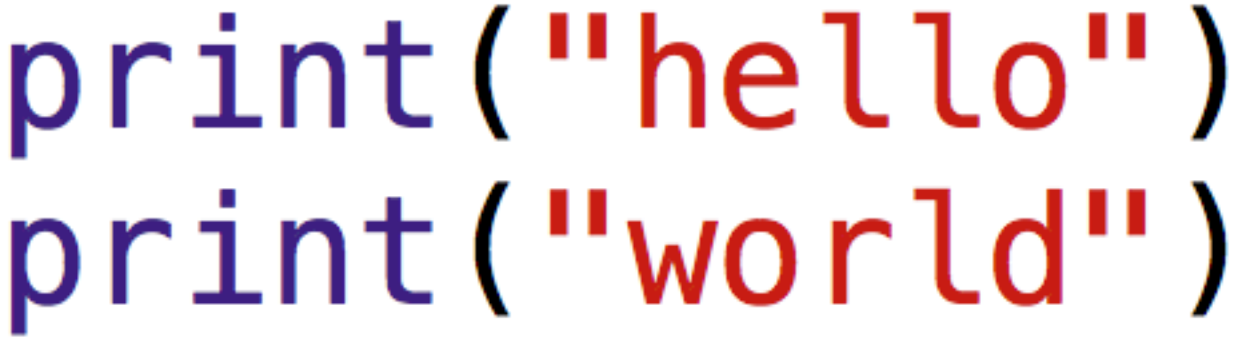
The Swift Language

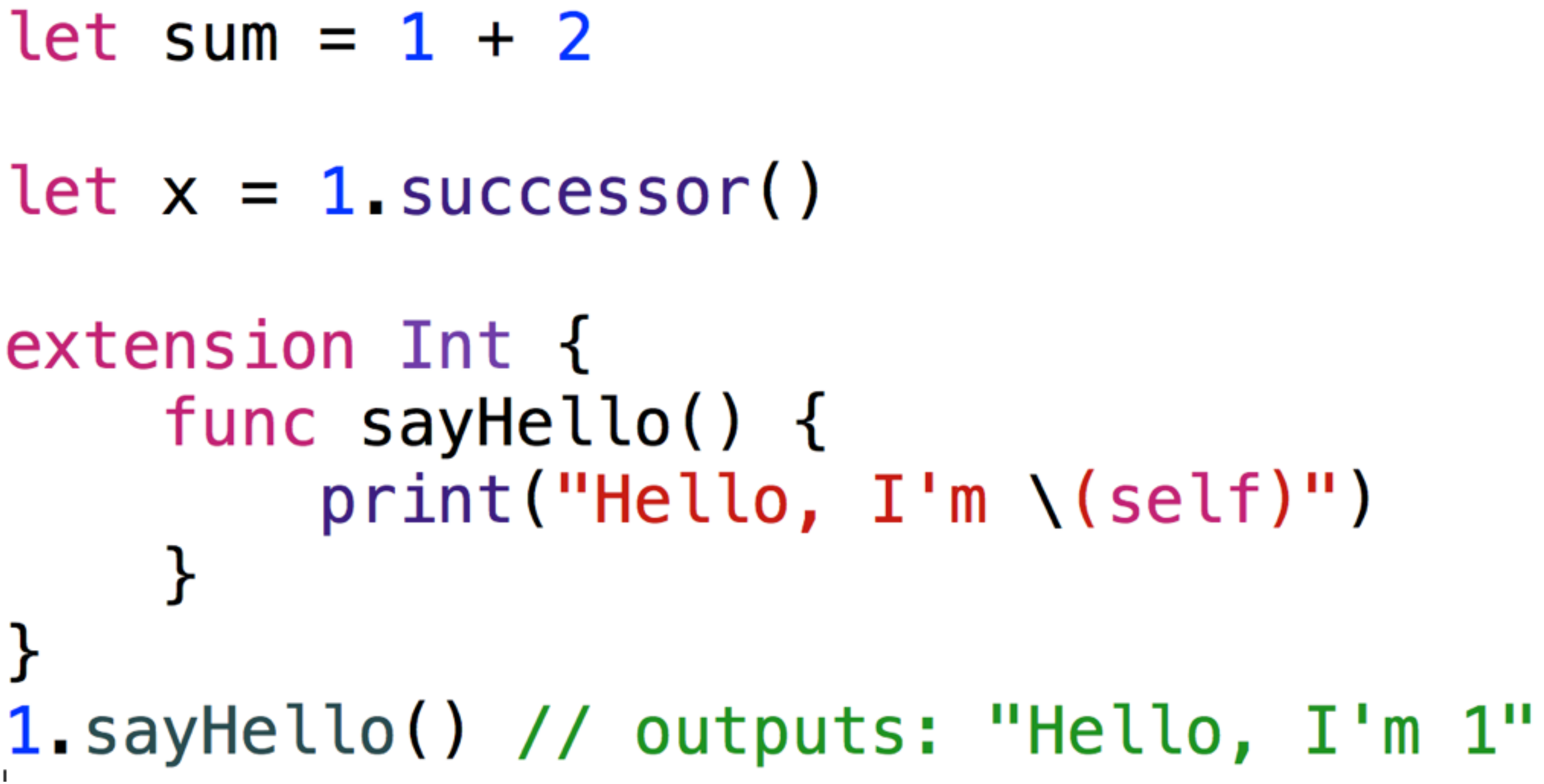
Ground of Being

- A complete Swift command is a statement.
- A Swift text file consists of multiple lines of text. Line breaks are meaningful.
- The typical layout of a program is one statement, one line:



```
print("hello")
print("world")
```

Everything Is an Object

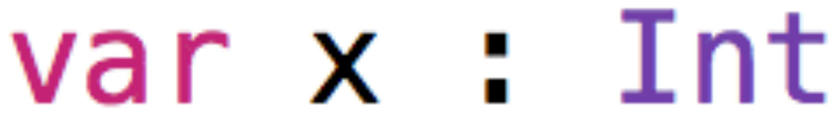


```
let sum = 1 + 2

let x = 1.successor()

extension Int {
    func sayHello() {
        print("Hello, I'm \(self)")
    }
}
1.sayHello() // outputs: "Hello, I'm 1"
```

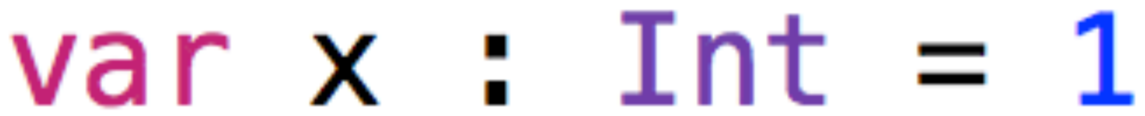
Three Flavors of Object Type

- Class
- Struct
- Enum

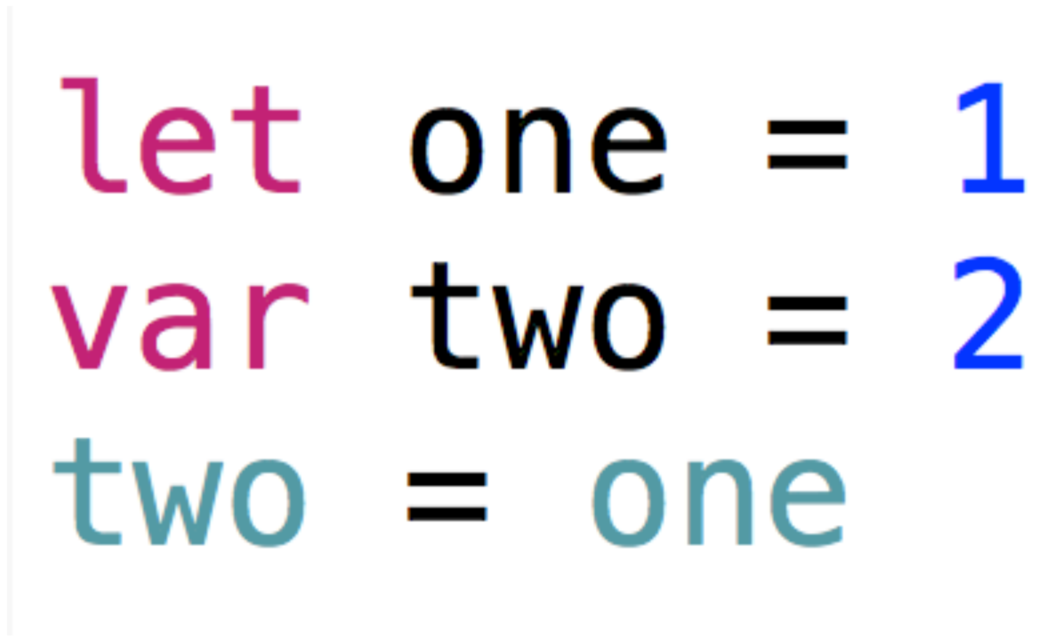
Variables



```
var x : Int
```

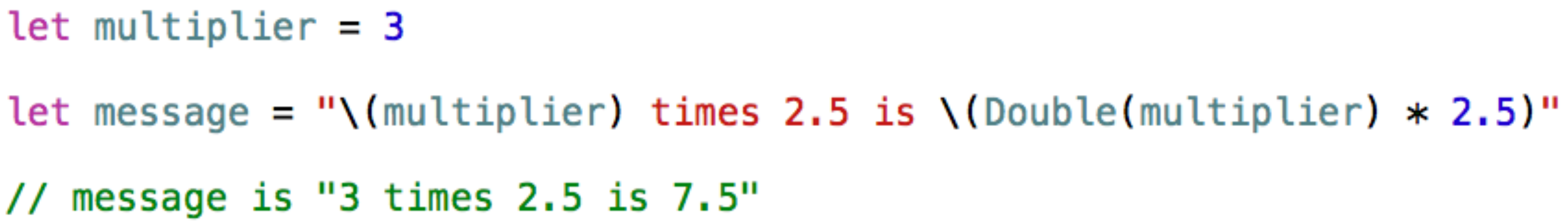


```
var x : Int = 1
```



```
let one = 1
var two = 2
two = one
```

String Interpolation



```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
// message is "3 times 2.5 is 7.5"
```

Range

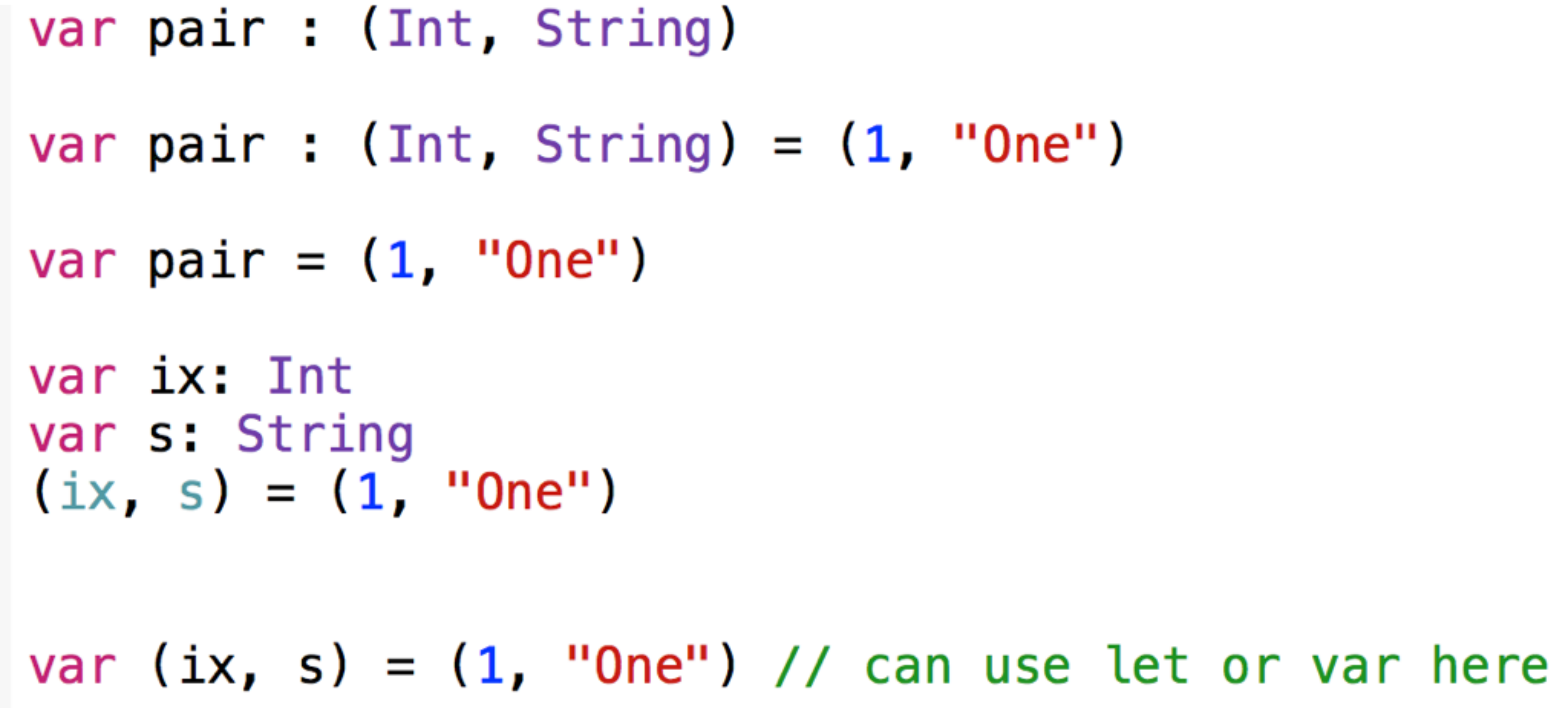
...

Closed interval operator. The notation a...b means “everything from a up to b, including b.”

..<

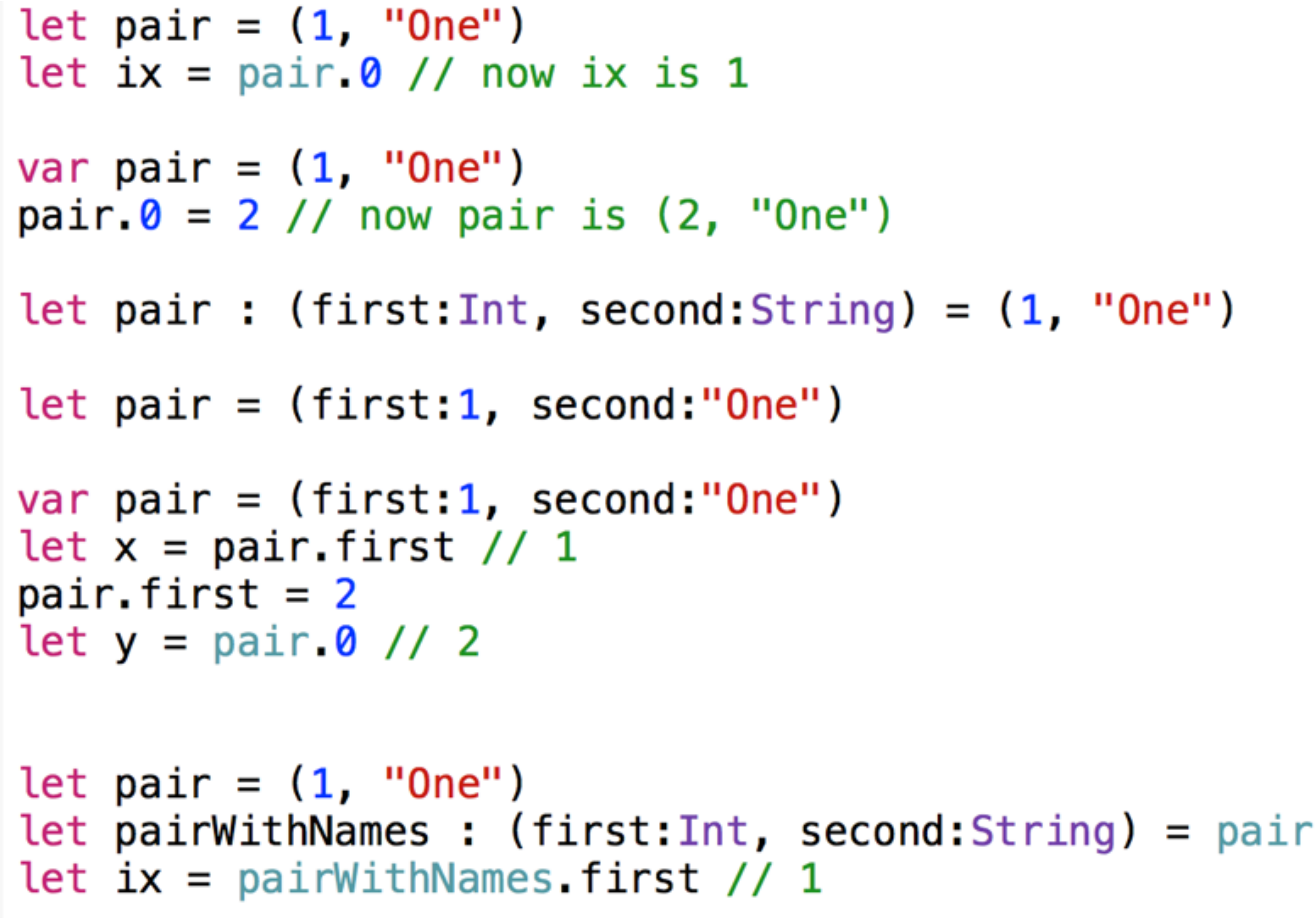
Half-open interval operator. The notation a..<b means “everything from a up to but not including b.”

Tuple



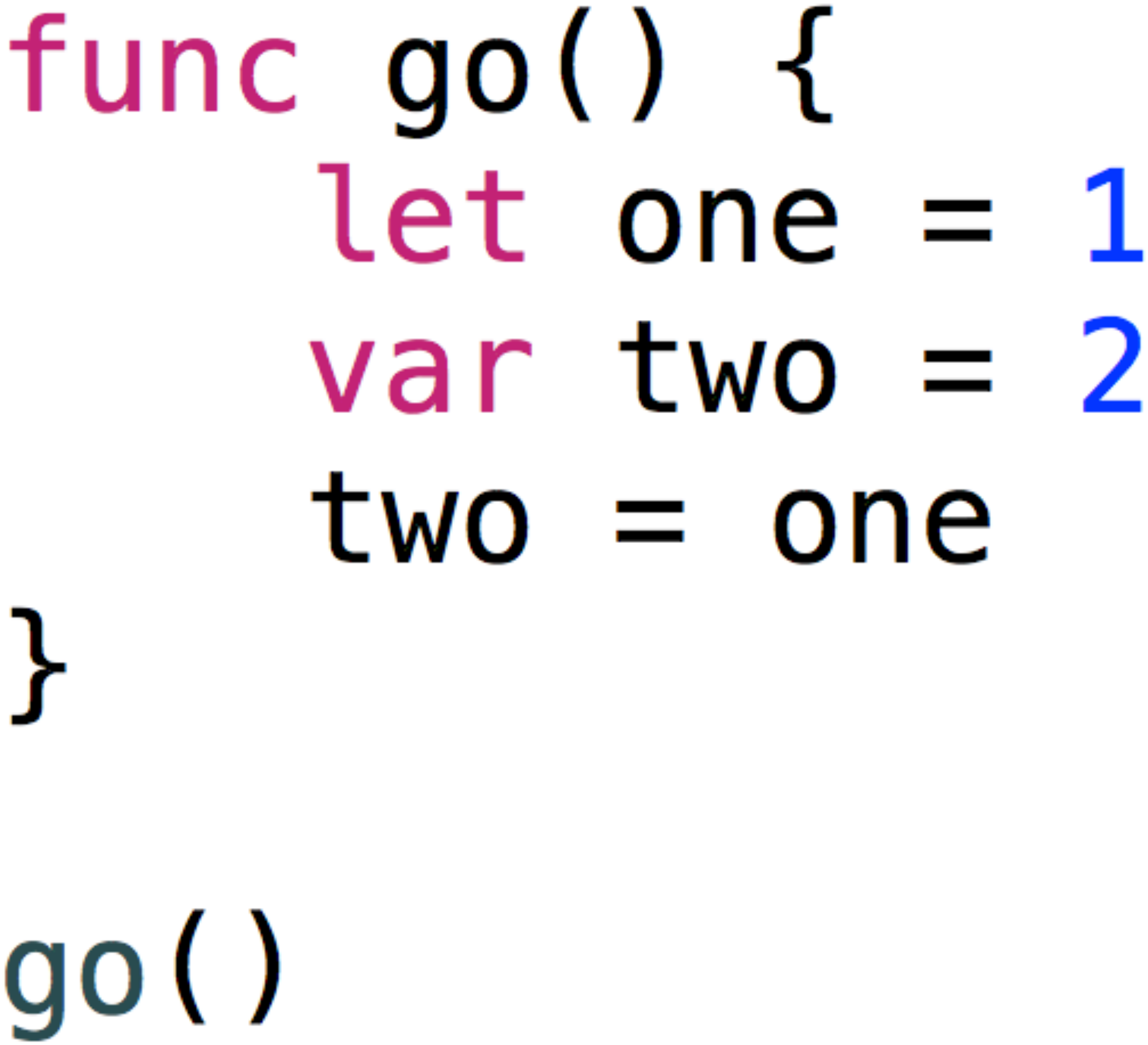
```
var pair : (Int, String)

var pair : (Int, String) = (1, "One")

var pair = (1, "One")

var ix: Int
var s: String
(ix, s) = (1, "One")


var (ix, s) = (1, "One") // can use let or var here
```

Tuple



```
let pair = (1, "One")
let ix = pair.0 // now ix is 1

var pair = (1, "One")
pair.0 = 2 // now pair is (2, "One")

let pair : (first:Int, second:String) = (1, "One")

let pair = (first:1, second:"One")

var pair = (first:1, second:"One")
let x = pair.first // 1
pair.first = 2
let y = pair.0 // 2


let pair = (1, "One")
let pairWithNames : (first:Int, second:String) = pair
let ix = pairWithNames.first // 1
```

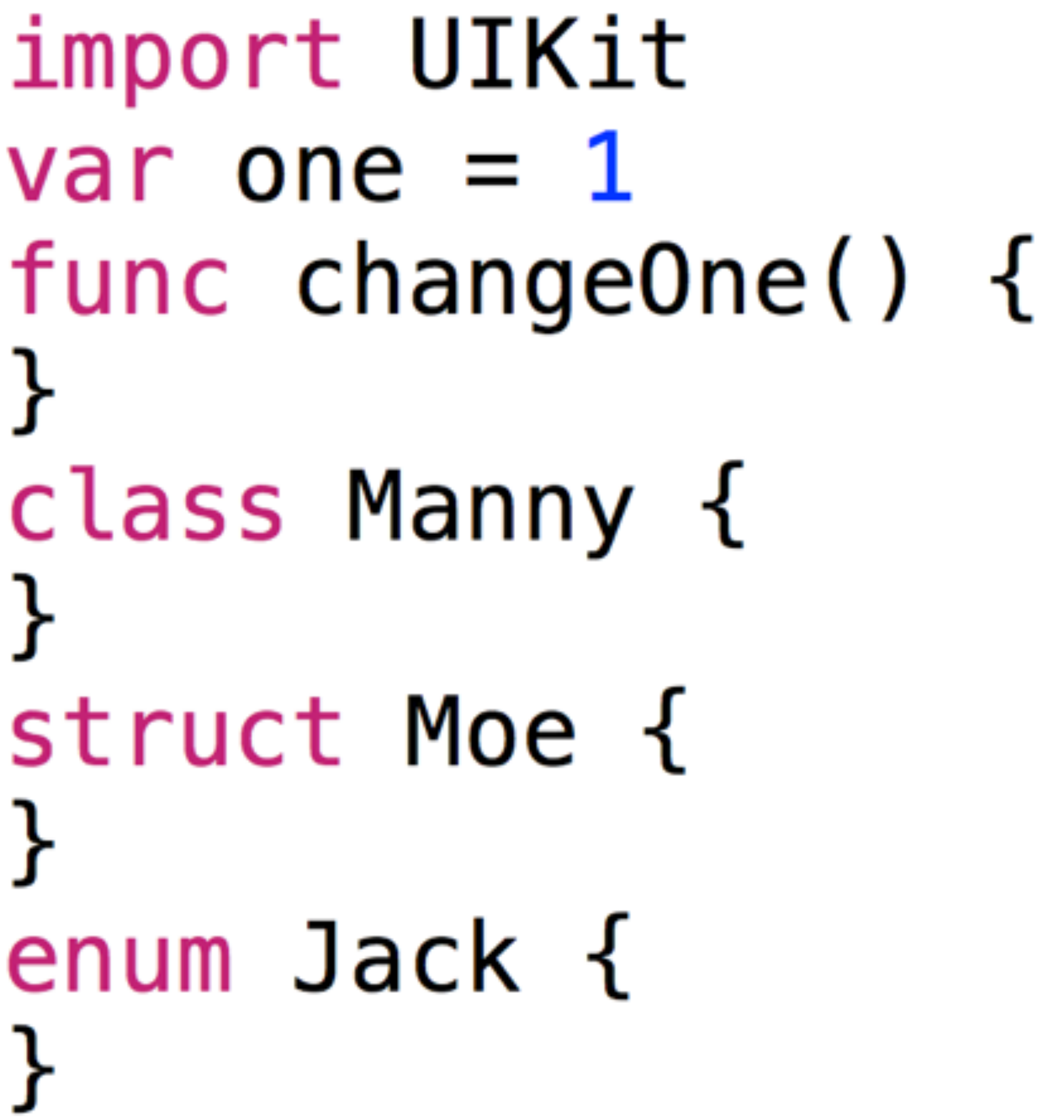
Functions



```
func go() {
    let one = 1
    var two = 2
    two = one
}

go()
```

The Structure of a Swift File

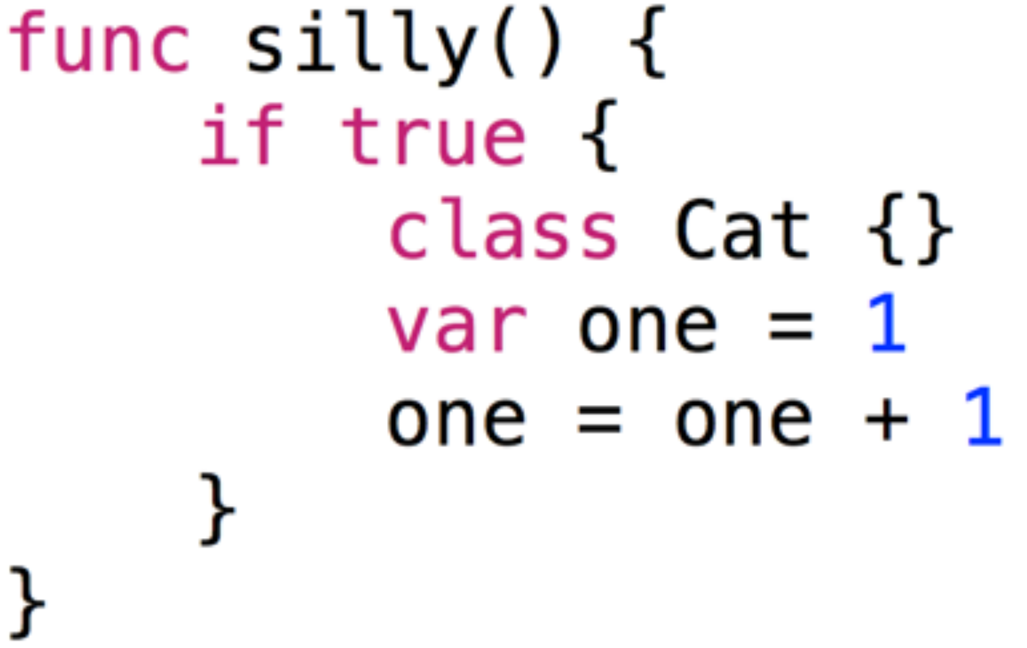
- Module *import* statements
- Variable declarations
- Function declarations
- Object type declarations



```
import UIKit
var one = 1
func changeOne() {
}
class Manny {
}
struct Moe {
}
enum Jack {
}
```

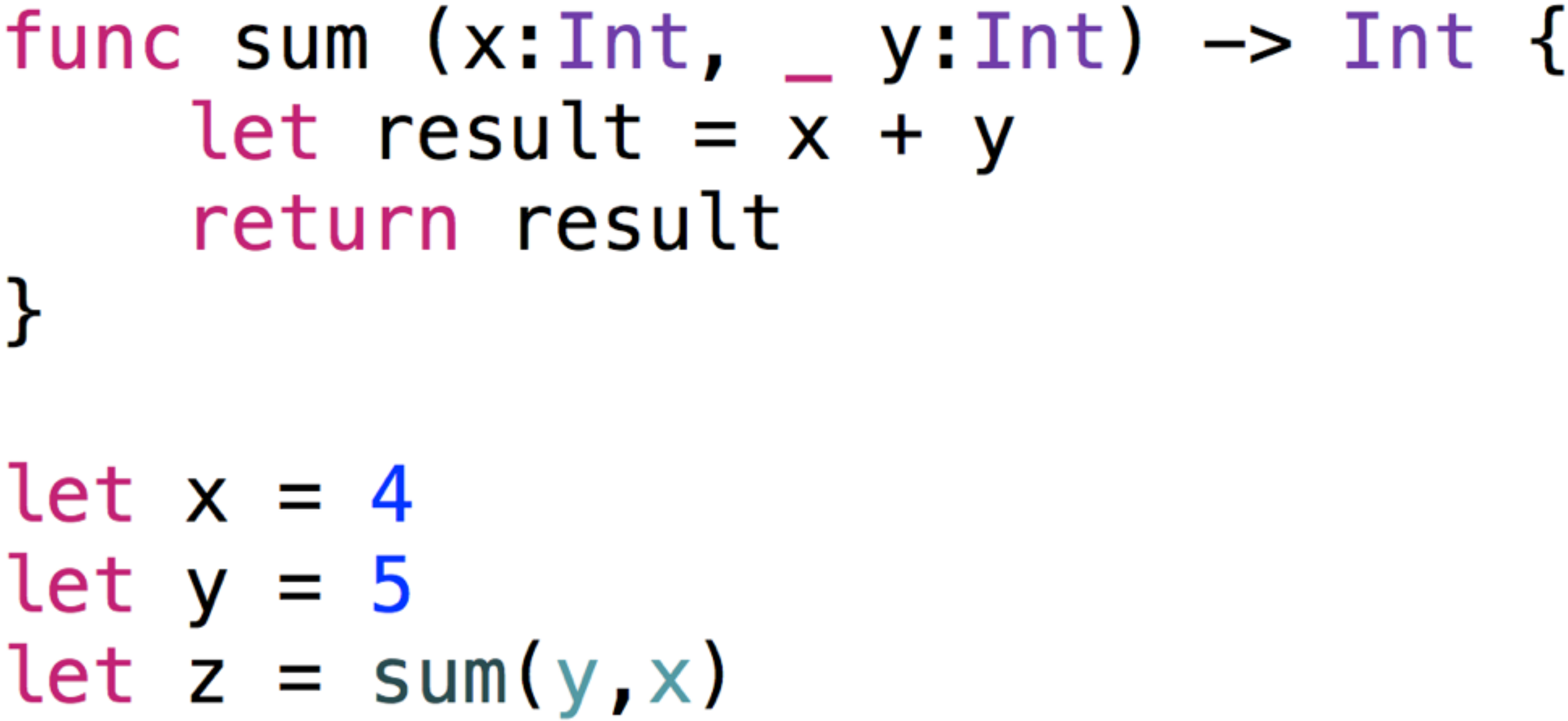
Scope and Lifetime

- A module is a scope.
- A file is a scope.
- An object declaration is a scope.
- Curly braces are a scope.



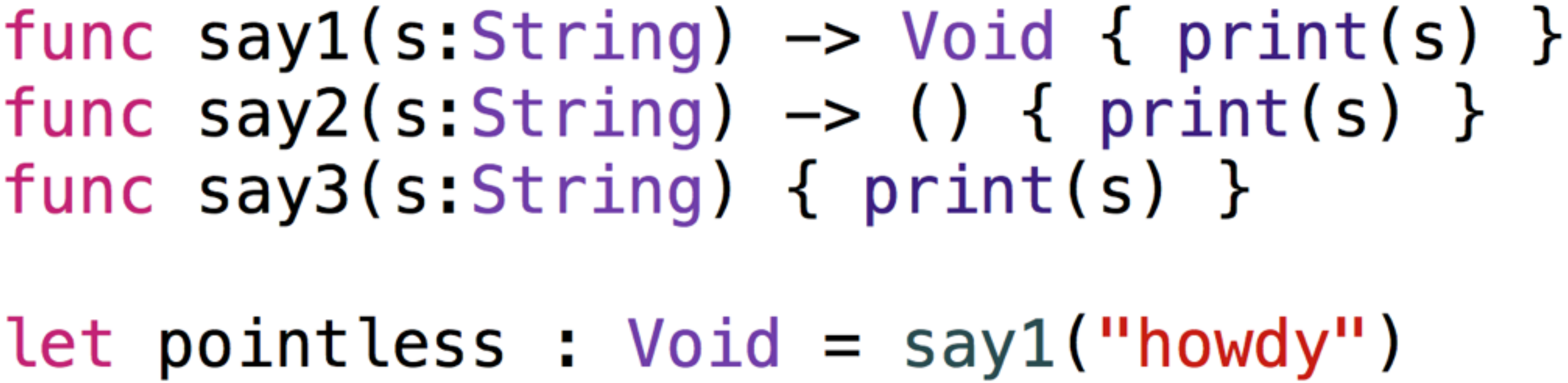
```
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

Function Parameters and Return Value



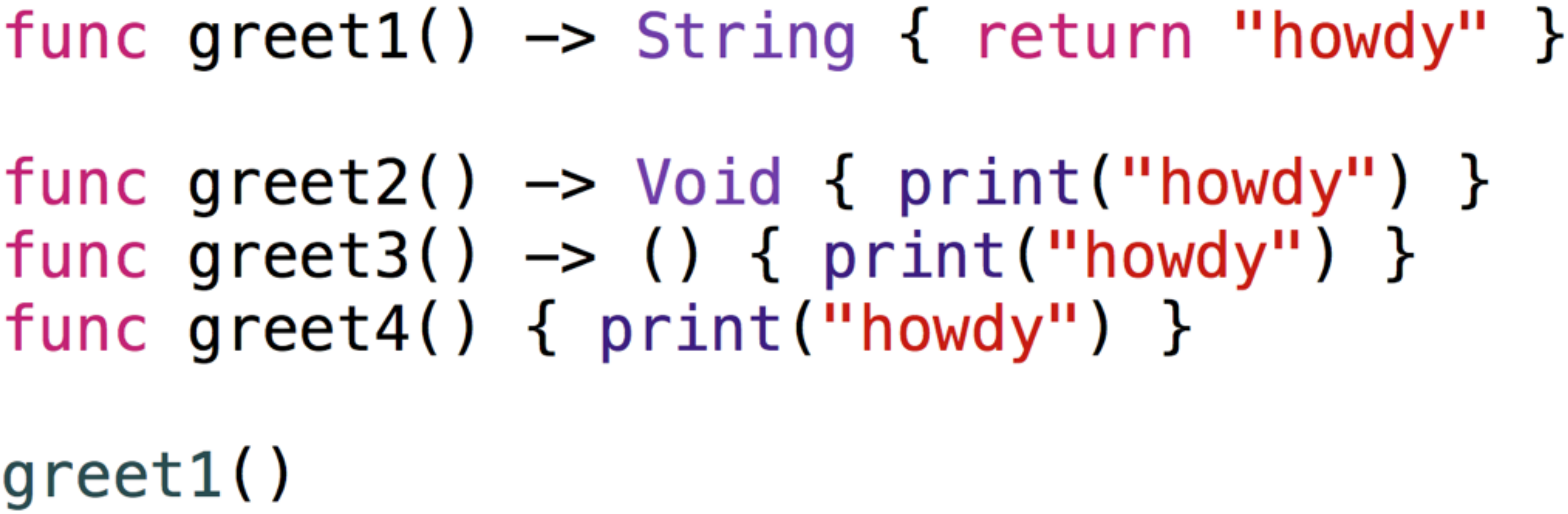
```
func sum (x:Int, _ y:Int) -> Int {
    let result = x + y
    return result
}

let x = 4
let y = 5
let z = sum(y,x)
```

A function without a return type



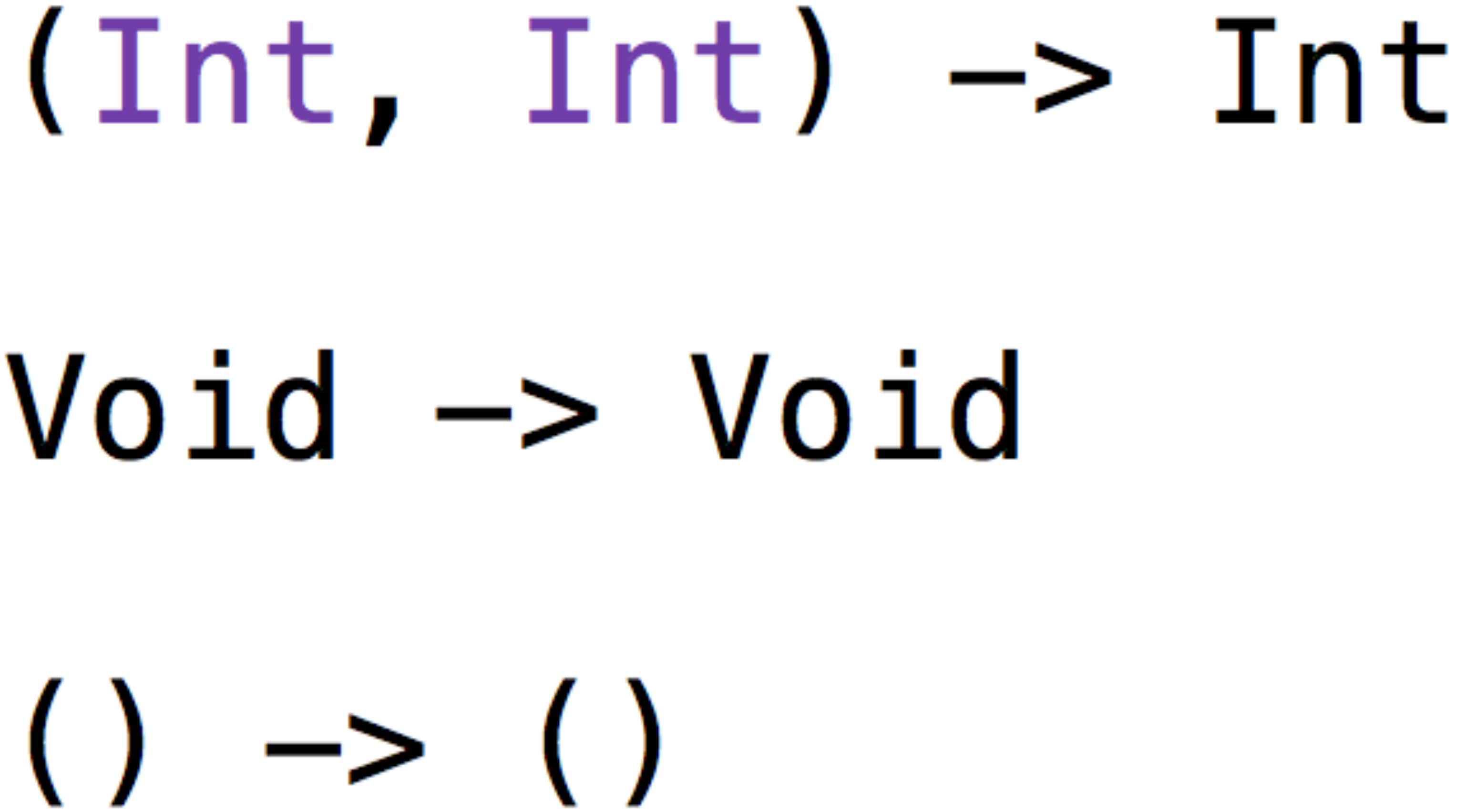
```
func say1(s:String) -> Void { print(s) }
func say2(s:String) -> () { print(s) }
func say3(s:String) { print(s) }

let pointless : Void = say1("howdy")
```

A function without any parameters



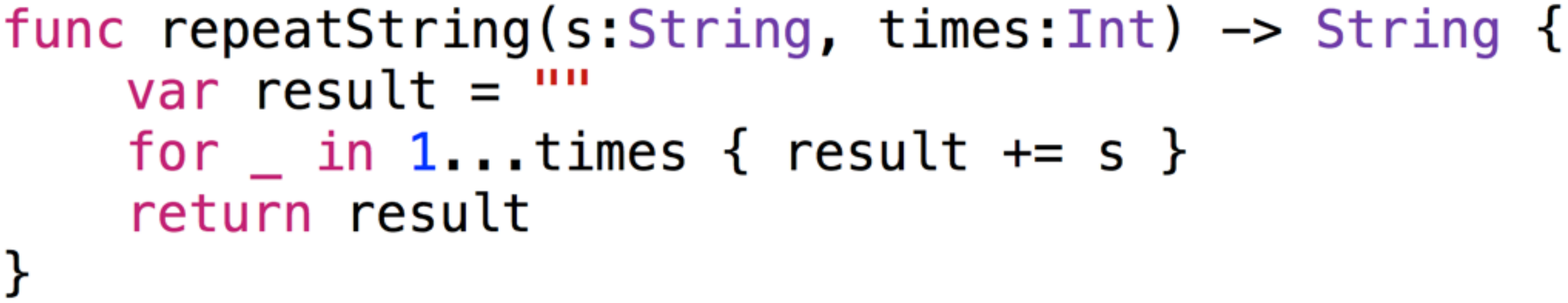
```
func greet1() -> String { return "howdy" }

func greet2() -> Void { print("howdy") }
func greet3() -> () { print("howdy") }
func greet4() { print("howdy") }

greet1()
```

Function Signature

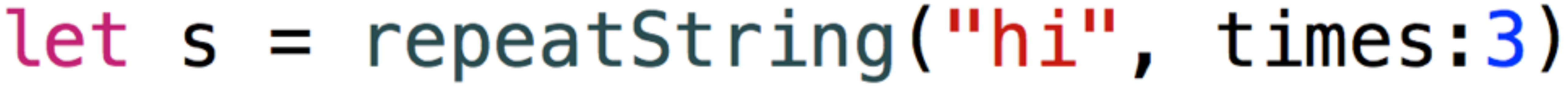


```
(Int, Int) -> Int

Void -> Void

() -> ()
```

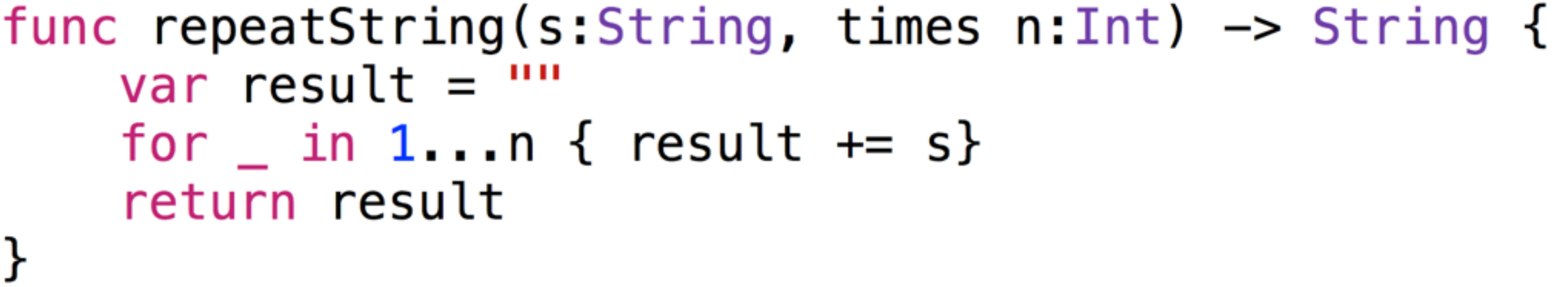
External Parameter Names



```
func repeatString(s:String, times:Int) -> String {
    var result = ""
    for _ in 1...times { result += s }
    return result
}
```

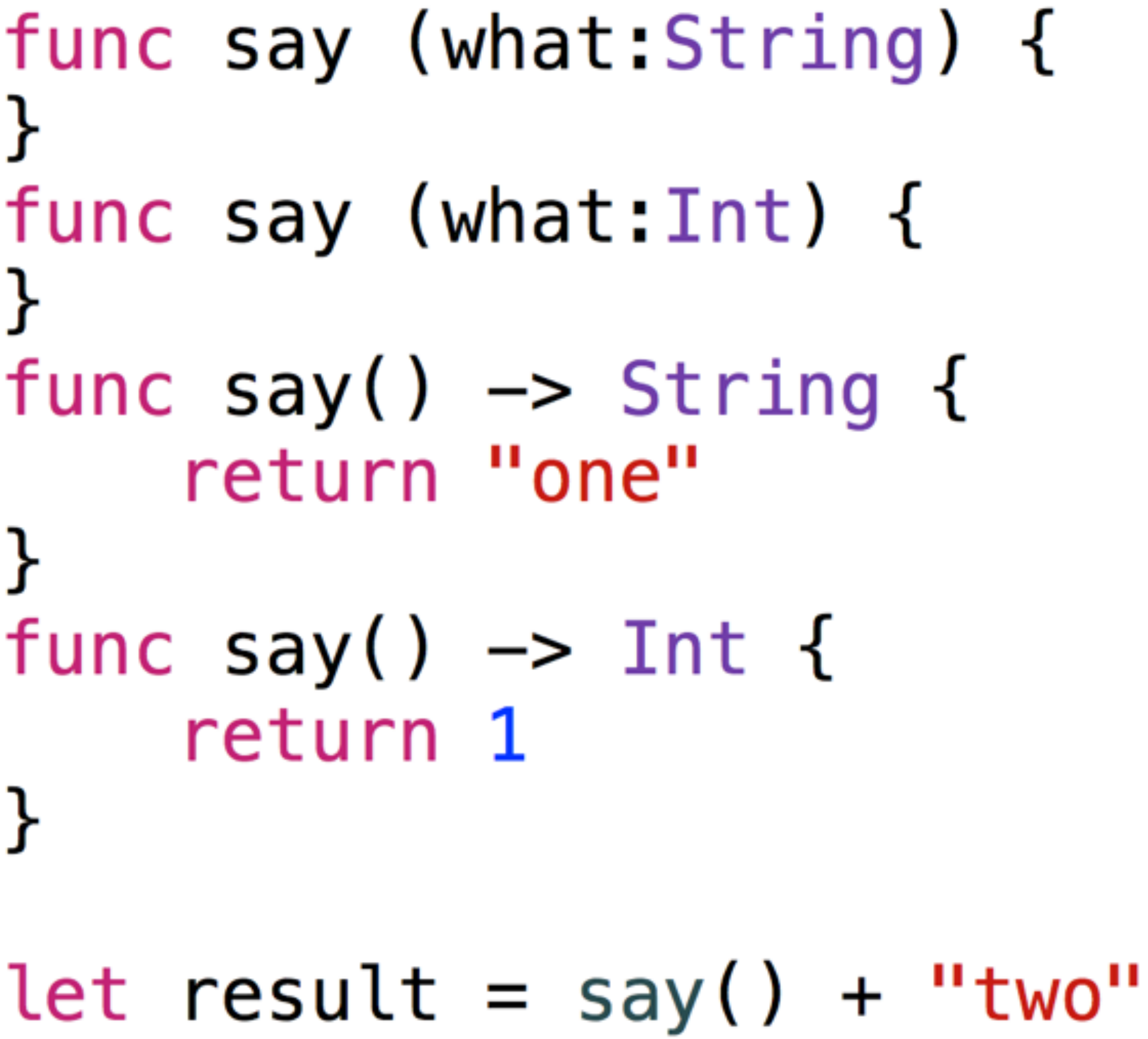


```
let s = repeatString("hi", times:3)
```



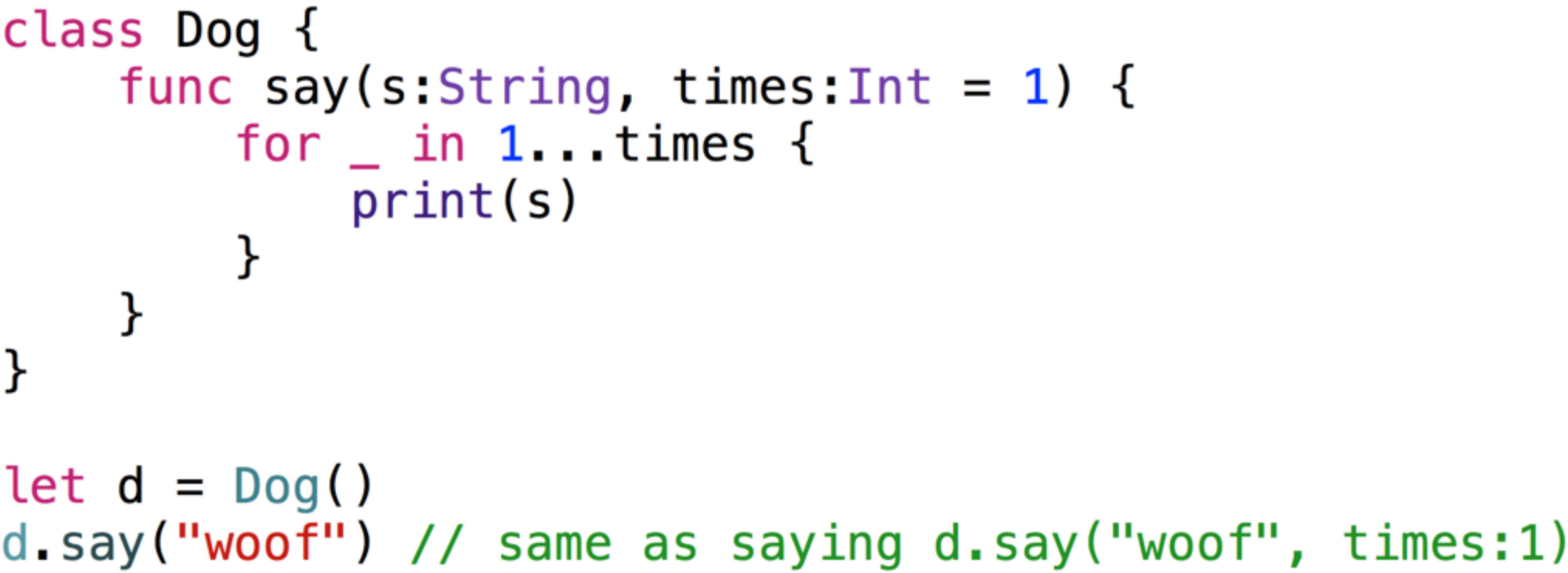
```
func repeatString(s:String, times n:Int) -> String {
    var result = ""
    for _ in 1...n { result += s}
    return result
}
```

Overloading



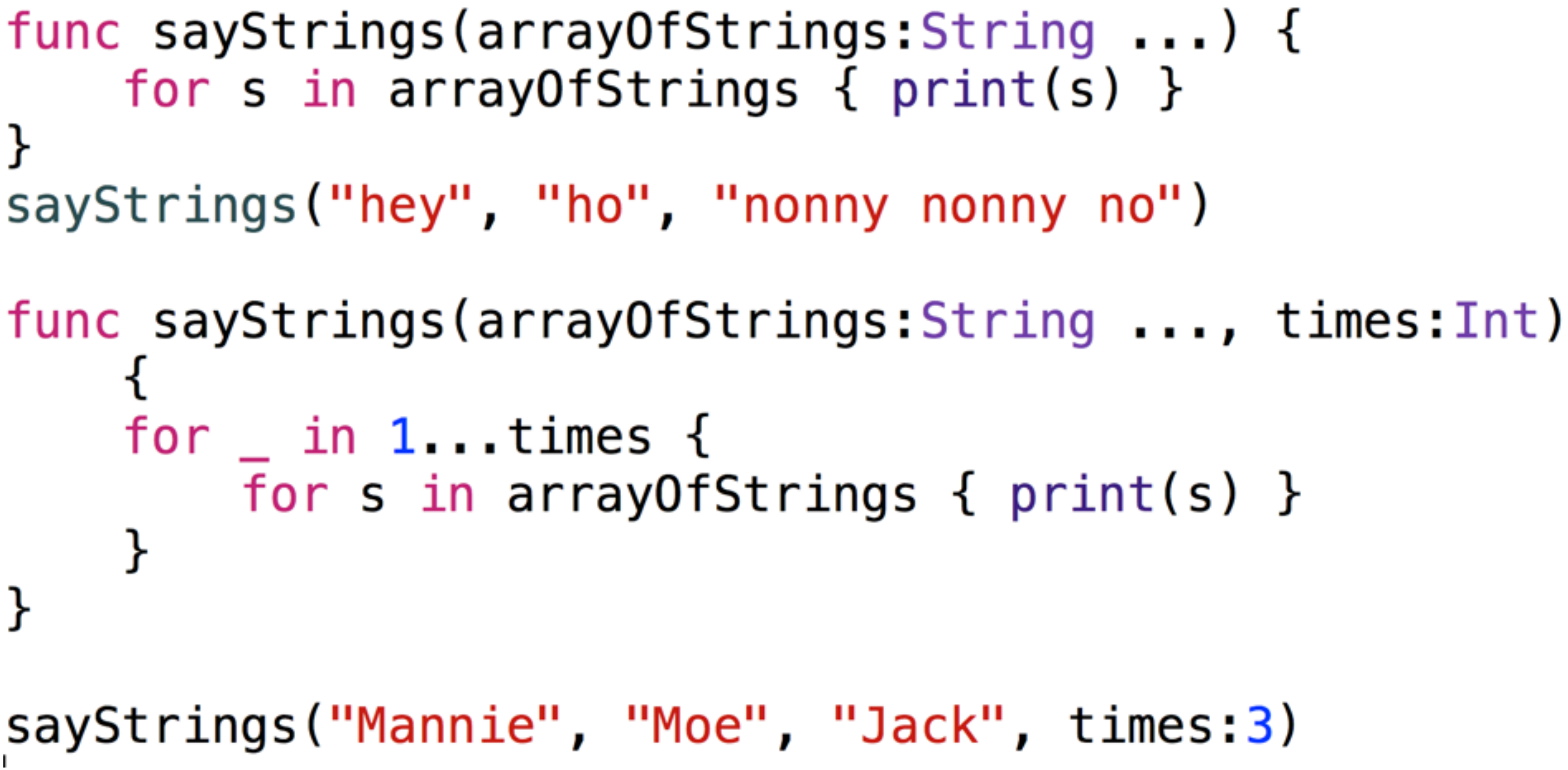
```
func say (what:String) {
}
func say (what:Int) {
}
func say() -> String {
    return "one"
}
func say() -> Int {
    return 1
}

let result = say() + "two"
```

Default Parameter Values



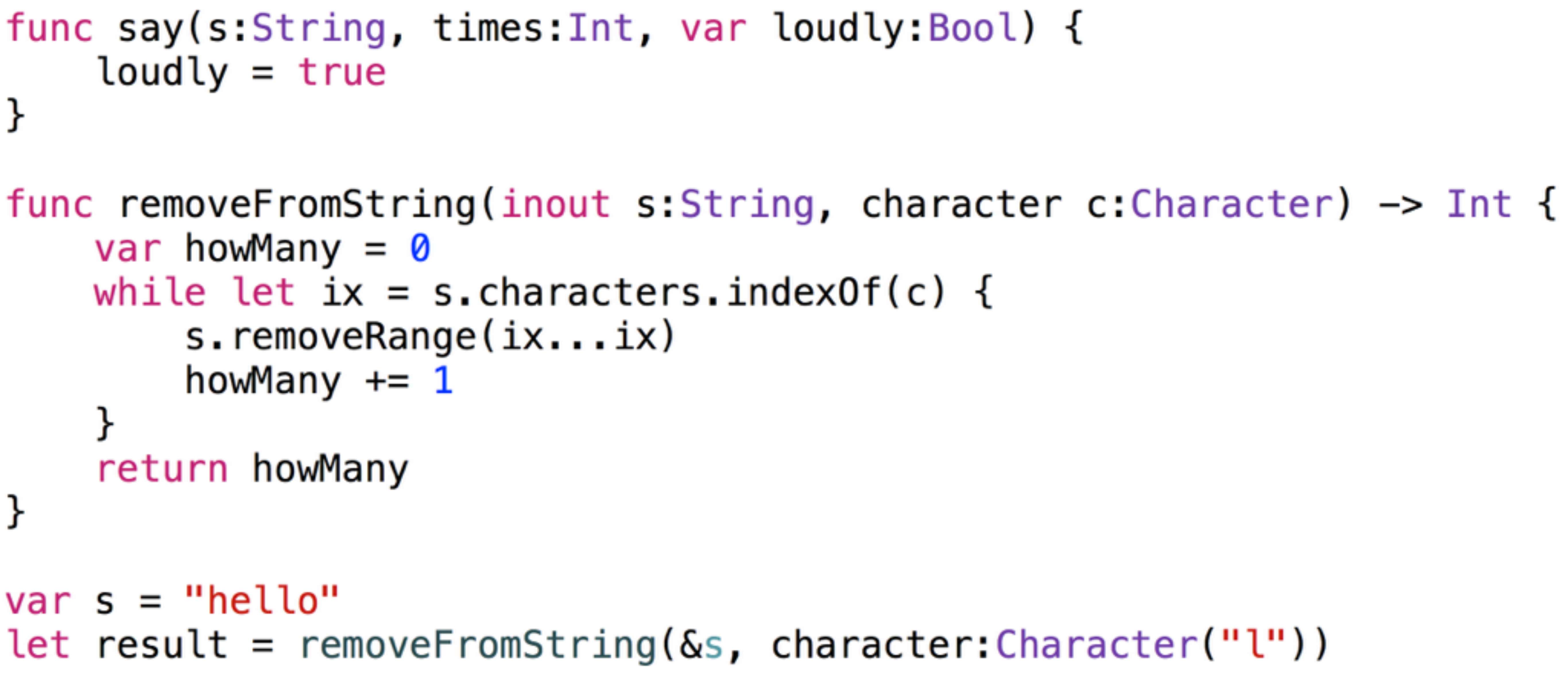
```
class Dog {
    func say(s:String, times:Int = 1) {
        for _ in 1...times {
            print(s)
        }
    }
}

let d = Dog()
d.say("woof") // same as saying d.say("woof", times:1)
```

Variadic Parameters



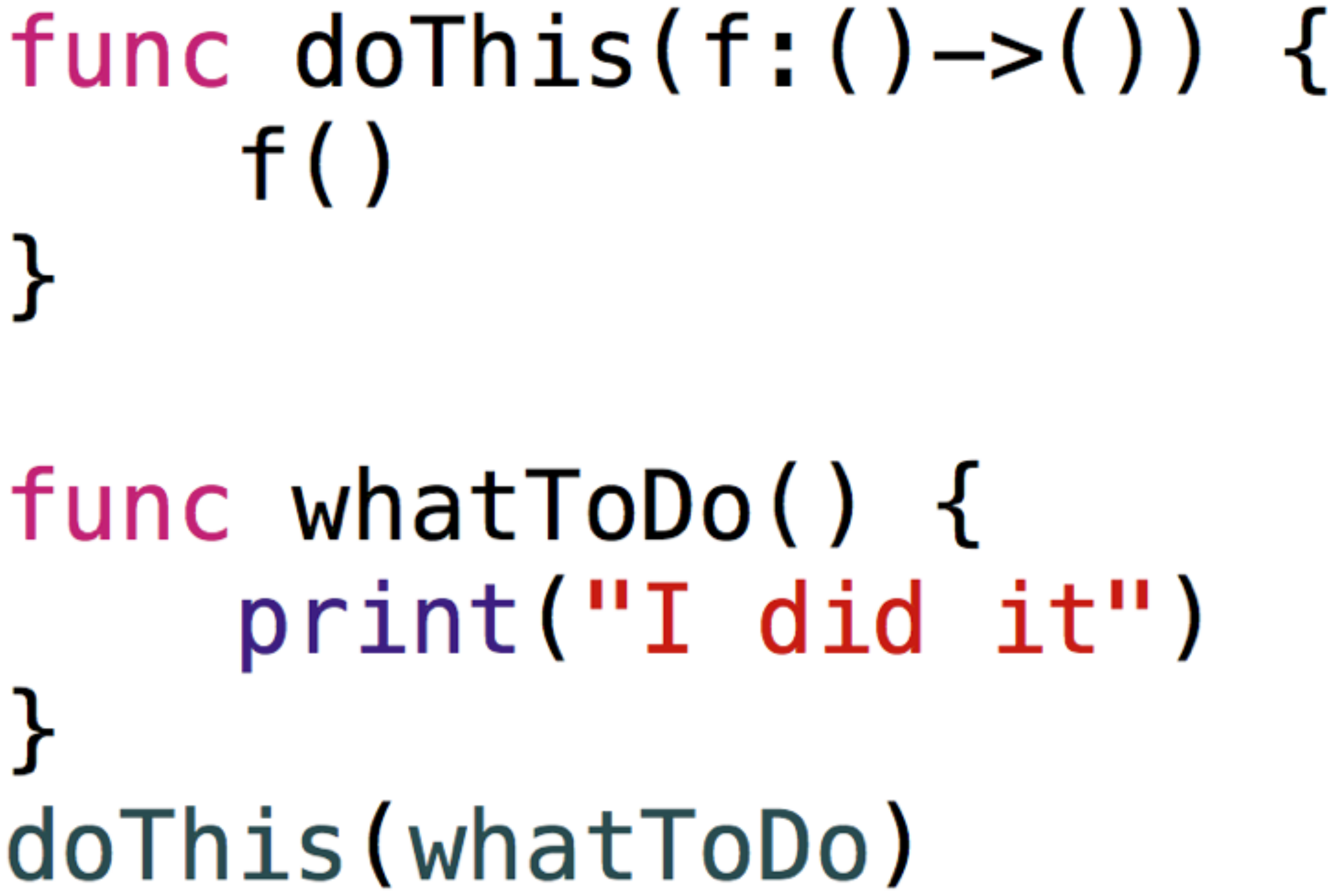
```
func sayStrings(arrayOfStrings:String ...) {
    for s in arrayOfStrings { print(s) }
}
sayStrings("hey", "ho", "nonny nonny no")

func sayStrings(arrayOfStrings:String ..., times:Int)
    {
    for _ in 1...times {
        for s in arrayOfStrings { print(s) }
    }
}

sayStrings("Mannie", "Moe", "Jack", times:3)
```

Modifiable Parameters



```swift
func say(s:String, times:Int, var loudly:Bool) {
    loudly = true
}

func removeFromString(inout s:String, character c:Character) -> Int {
    var howMany = 0
    while let ix = s.characters.indexOf(c) {
        s.removeRange(ix...ix)
        howMany += 1
    }
    return howMany
}

var s = "hello"
let result = removeFromString(&s, character:Character("l"))
```

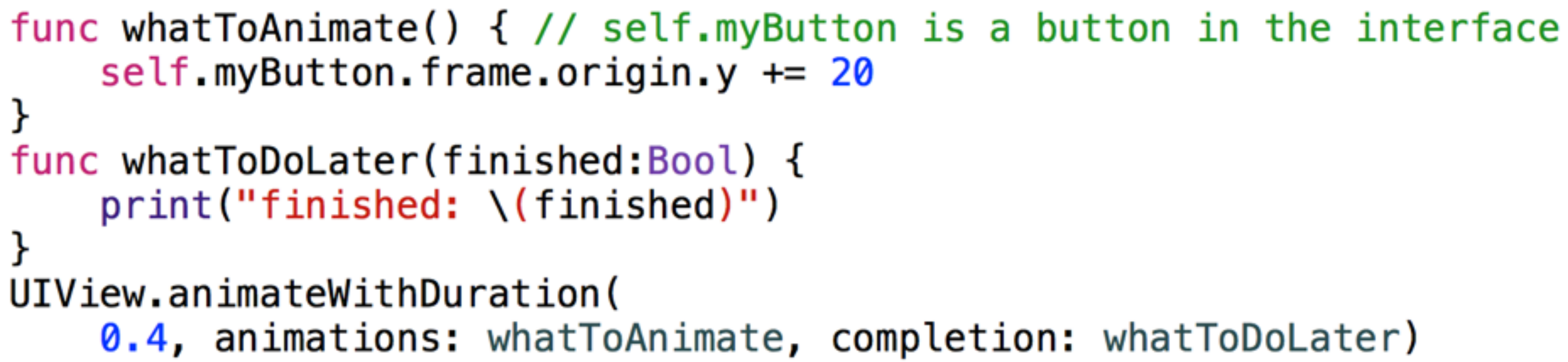
Function As Value



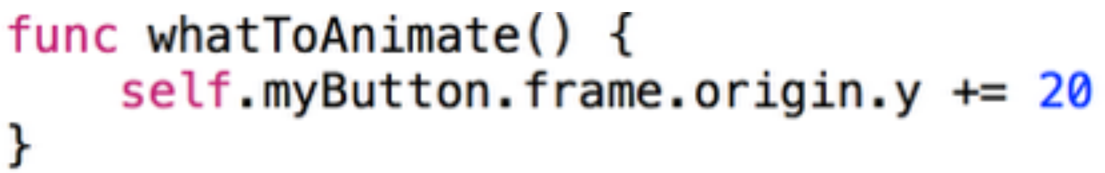
```
func doThis(f:()->()) {
    f()
}

func whatToDo() {
    print("I did it")
}
doThis(whatToDo)
```

Anonymous Functions

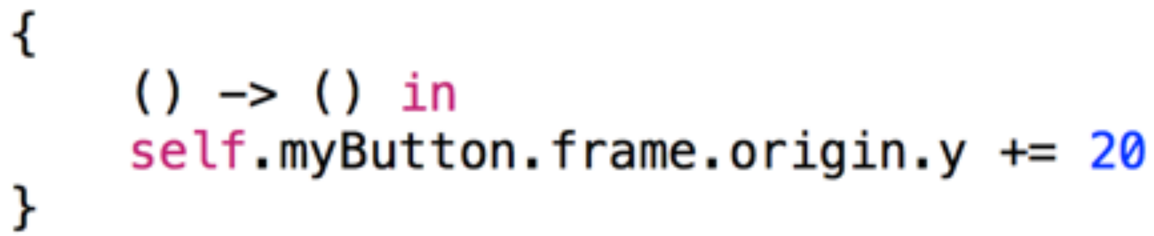
- To form an anonymous function, you do two things:
  - Create the function body itself, including the surrounding curly braces, but with no function declaration.
  - If necessary, express the function’s parameter list and return type as the first line inside the curly braces, followed by the keyword ***in***.



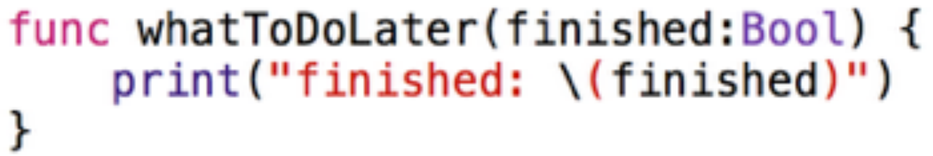
```swift
func whatToAnimate() { // self.myButton is a button in the interface
    self.myButton.frame.origin.y += 20
}
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
UIView.animateWithDuration(
    0.4, animations: whatToAnimate, completion: whatToDoLater)
```



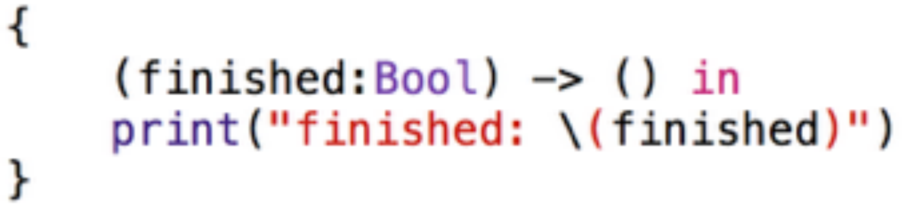
```swift
func whatToAnimate() {
    self.myButton.frame.origin.y += 20
}
```



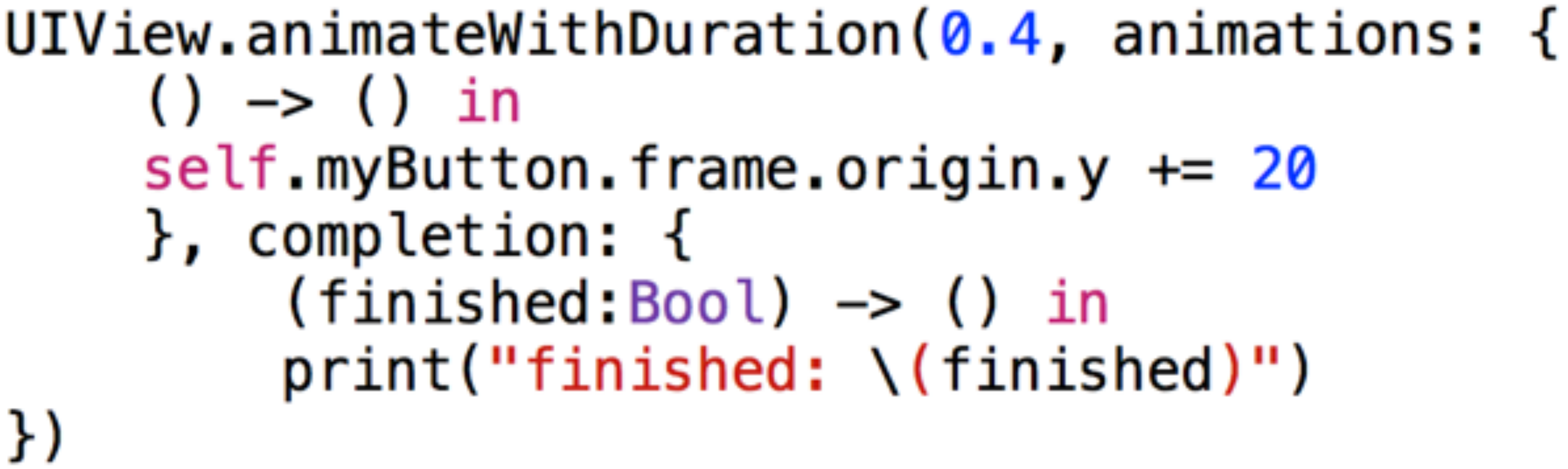
```swift
{
    () -> () in
    self.myButton.frame.origin.y += 20
}
```



```swift
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
```

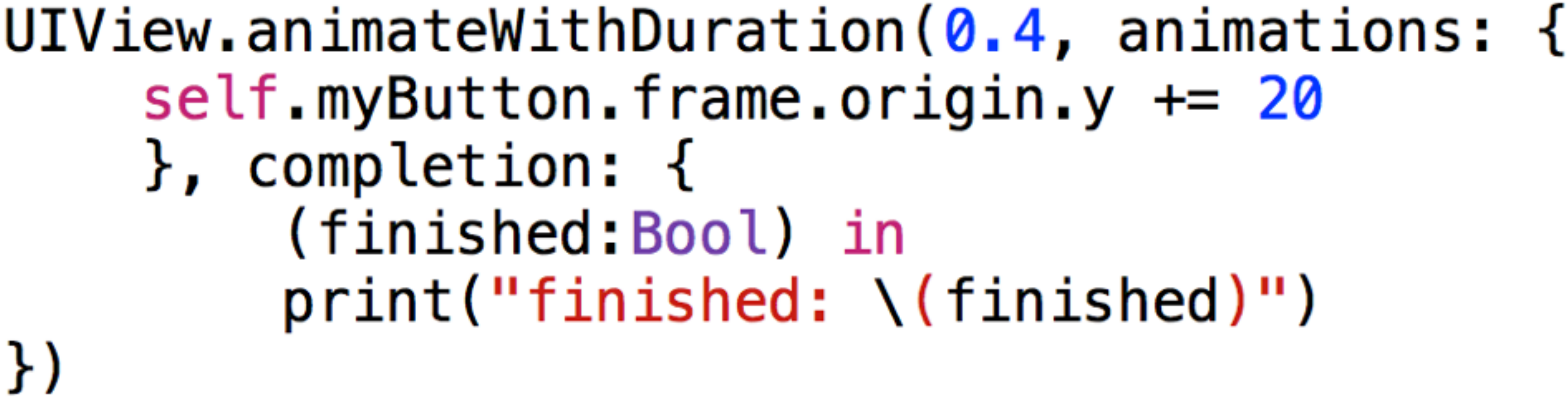


```swift
{
    (finished:Bool) -> () in
    print("finished: \(finished)")
}
```



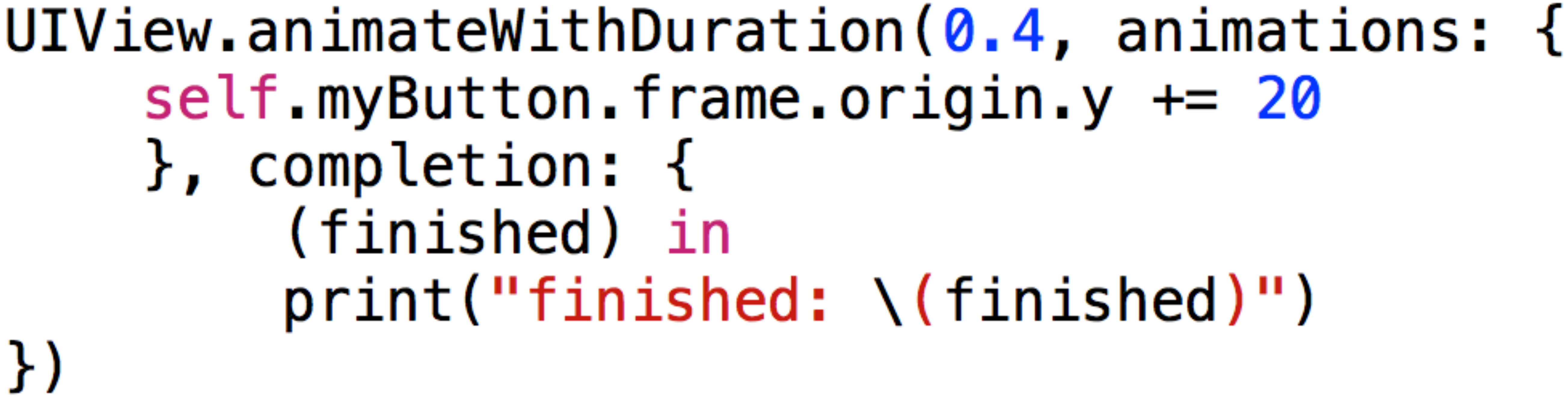
```swift
UIView.animateWithDuration(0.4, animations: {
    () -> () in
    self.myButton.frame.origin.y += 20
    }, completion: {
        (finished:Bool) -> () in
        print("finished: \(finished)")
})
```

- If the anonymous function takes no parameters, and if the return type can be omitted, the in line itself can be omitted entirely



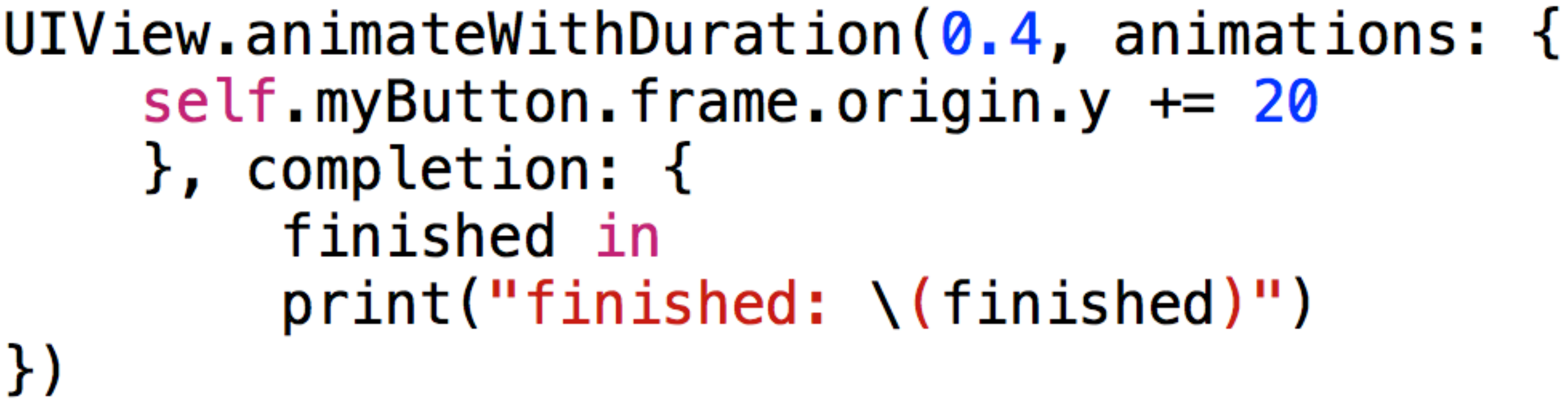
```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        (finished:Bool) in
        print("finished: \(finished)")
})
```

- If the anonymous function takes parameters and their types are known to the compiler, the types can be omitted



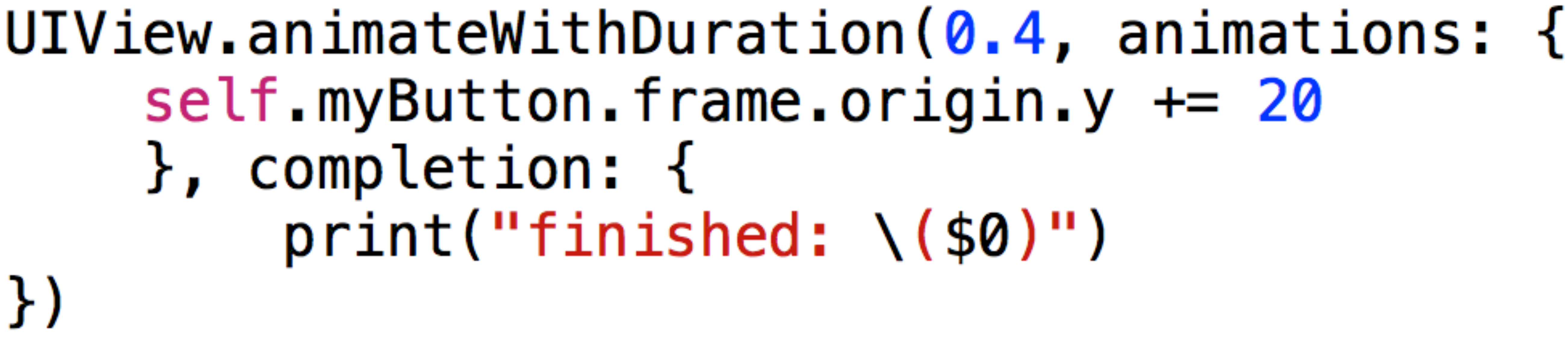
```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        (finished) in
        print("finished: \(finished)")
})
```

- If the parameter types are omitted, the parentheses around the parameter list can be omitted



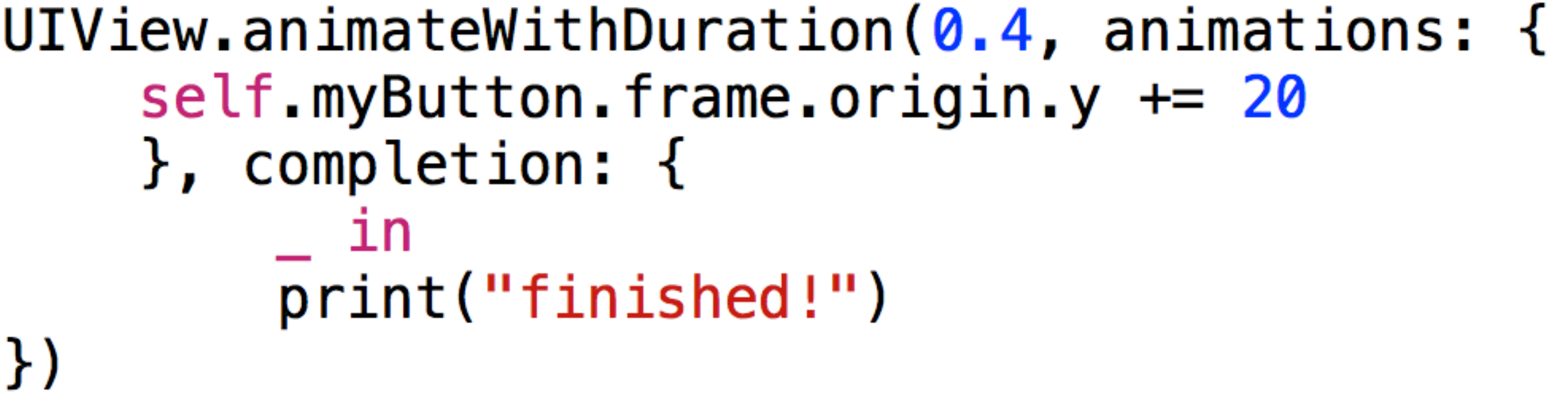
```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        finished in
        print("finished: \(finished)")
})
```

- If the return type can be omitted, and if the parameter types are known to the compiler, you can omit the in line and refer to the parameters directly within the body of the anonymous function by using the magic names $0, $1, and so on, in order



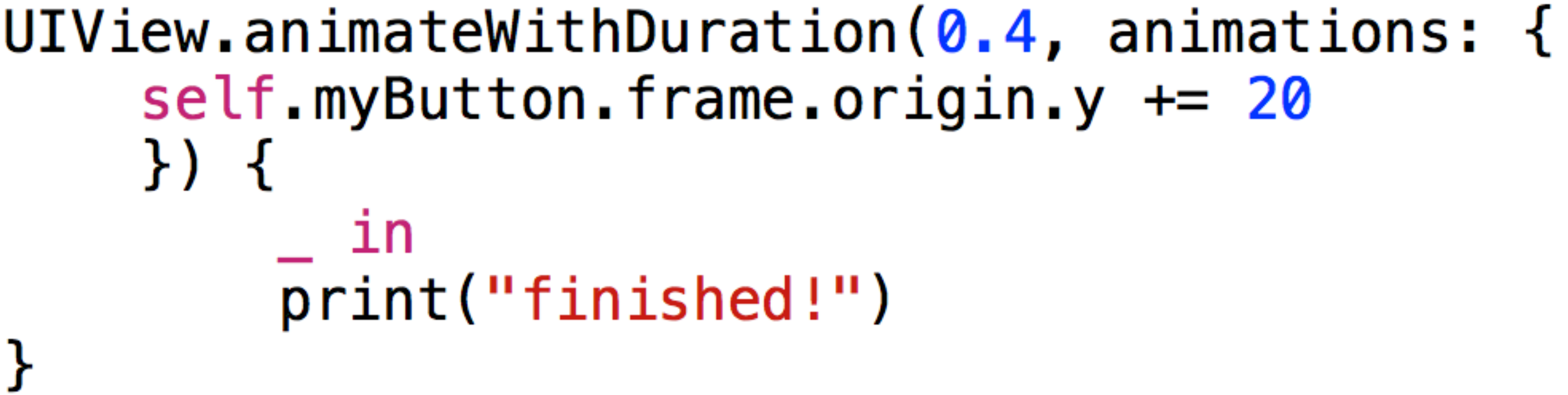
```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        print("finished: \($0)")
})
```

- If the anonymous function body doesn’t need to refer to a parameter, you can substitute an underscore for its name in the parameter list in the in line; in fact, if the anonymous function body doesn’t need to refer to any of the parameters, you can substitute *one underscore for the entire parameter list*



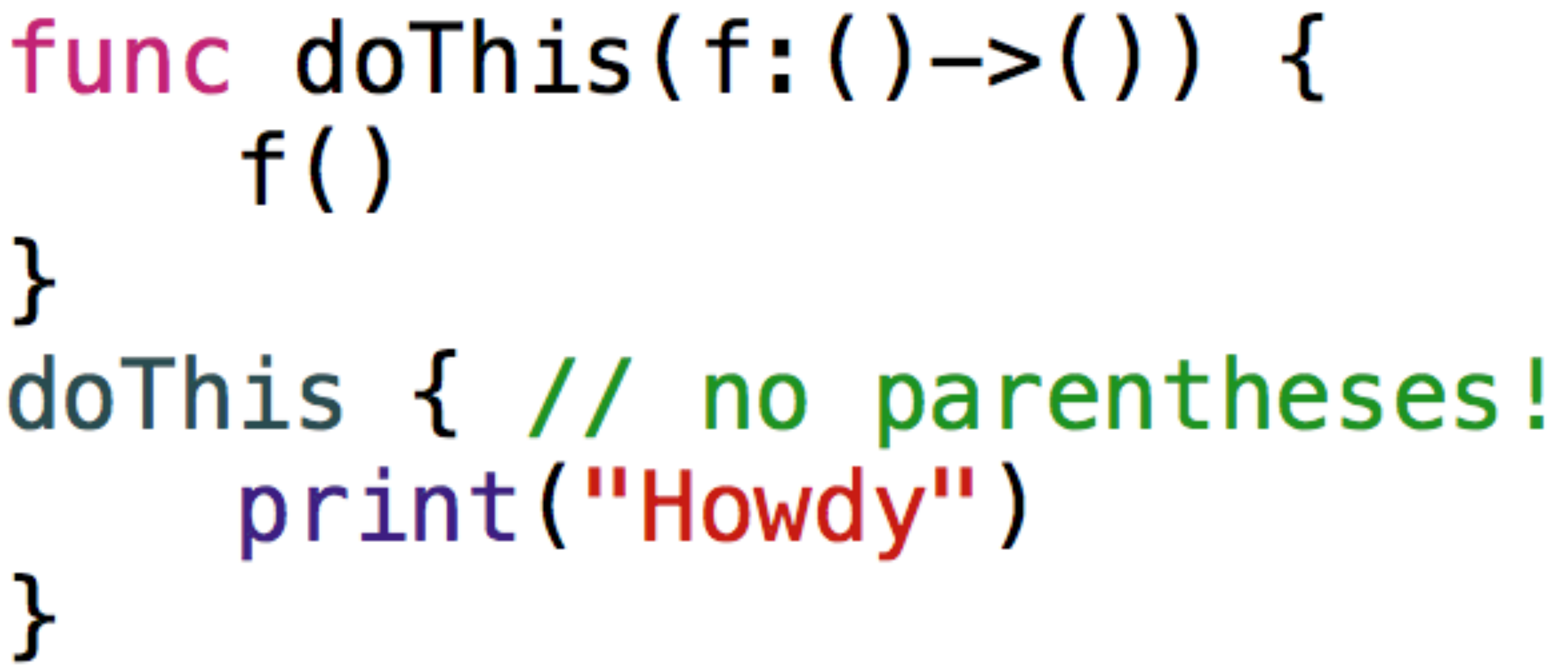
```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        _ in
        print("finished!")
})
```

- If, as will just about always be the case, your anonymous function is the last argument being passed in this function call, you can close the function call with a right parenthesis before this last argument, and then put just the anonymous function body without a label (this is called a *trailing function*)



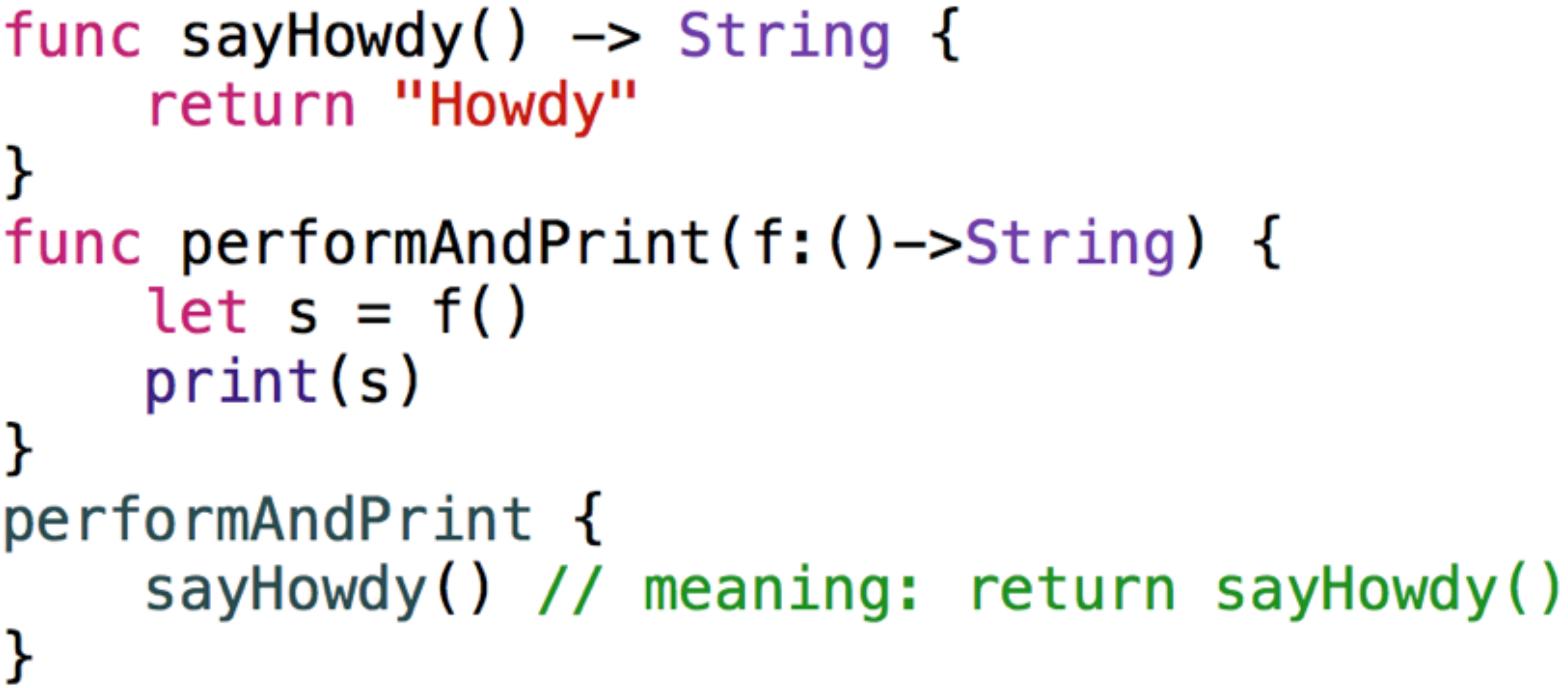
```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }) {
        _ in
        print("finished!")
}
```

- If you use the trailing function syntax, and if the function you are calling takes no parameters other than the function you are passing to it, you can omit the empty parentheses from the call. This is the only situation in which you can omit the parentheses from a function call!



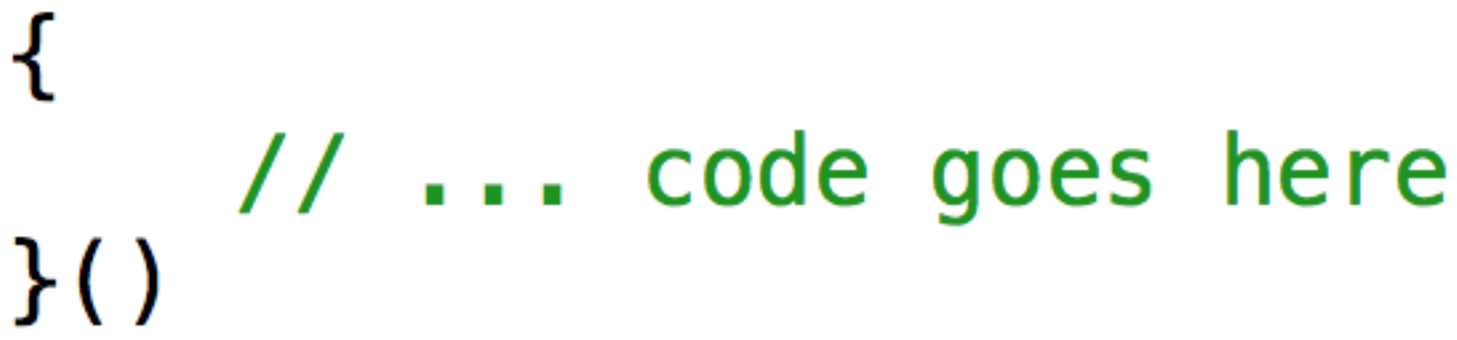
```
func doThis(f:()->()) {
    f()
}
doThis { // no parentheses!
    print("Howdy")
}
```

- If the anonymous function body consists of exactly *one statement* and that statement consists of returning a value with the keyword *return*, the keyword *return* can be omitted

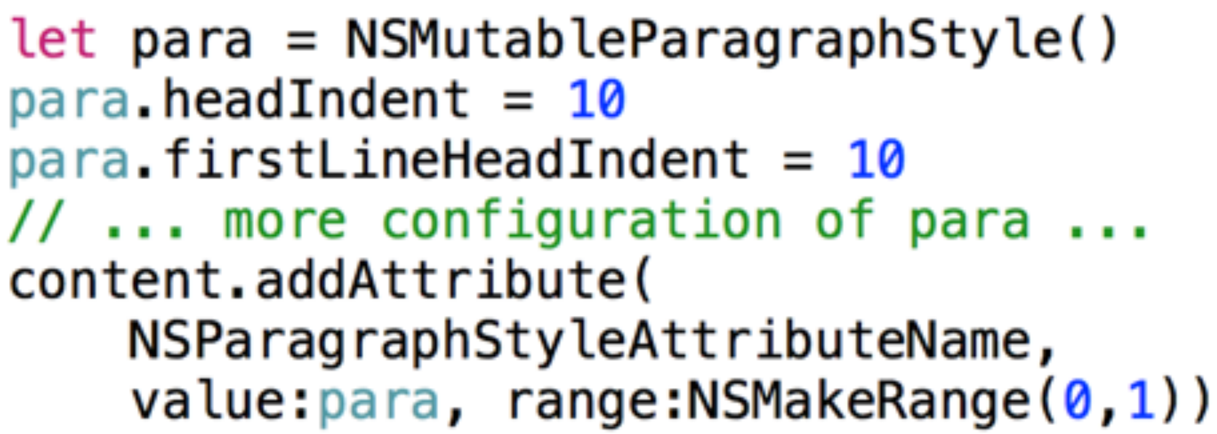


```
func sayHowdy() -> String {
    return "Howdy"
}
func performAndPrint(f:()->String) {
    let s = f()
    print(s)
}
performAndPrint {
    sayHowdy() // meaning: return sayHowdy()
}
```

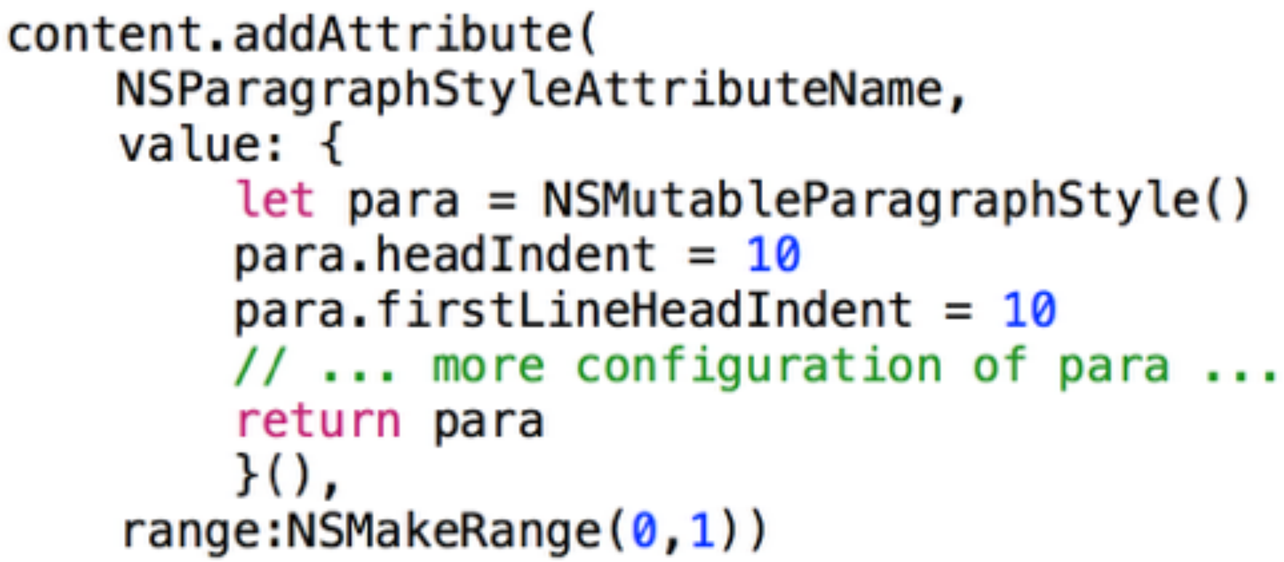
Define-and-Call



```
{
    // ... code goes here
}()
```

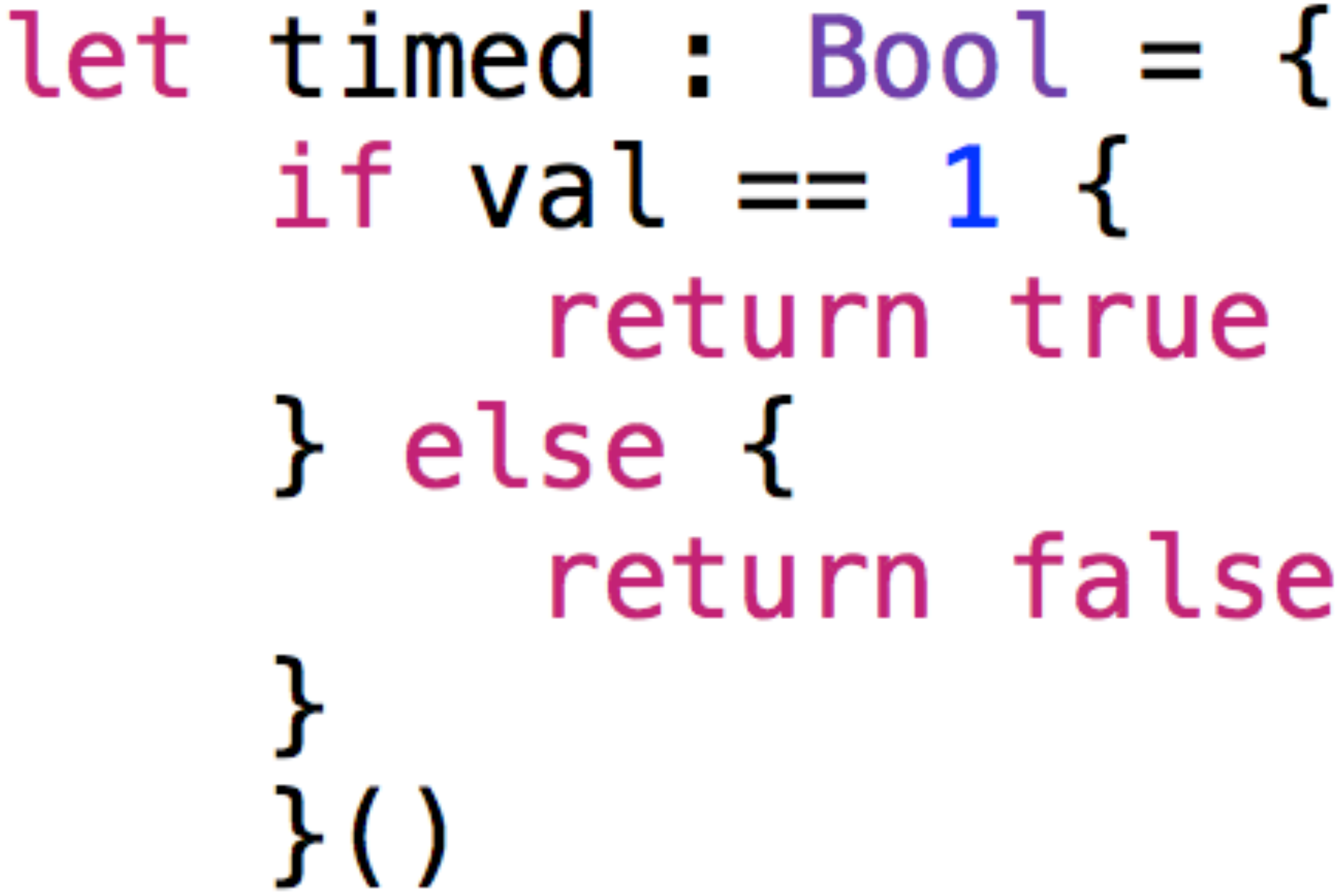


```
let para = NSMutableParagraphStyle()
para.headIndent = 10
para.firstLineHeadIndent = 10
// ... more configuration of para ...
content.addAttribute(
    NSParagraphStyleAttributeName,
    value:para, range:NSMakeRange(0,1))
```



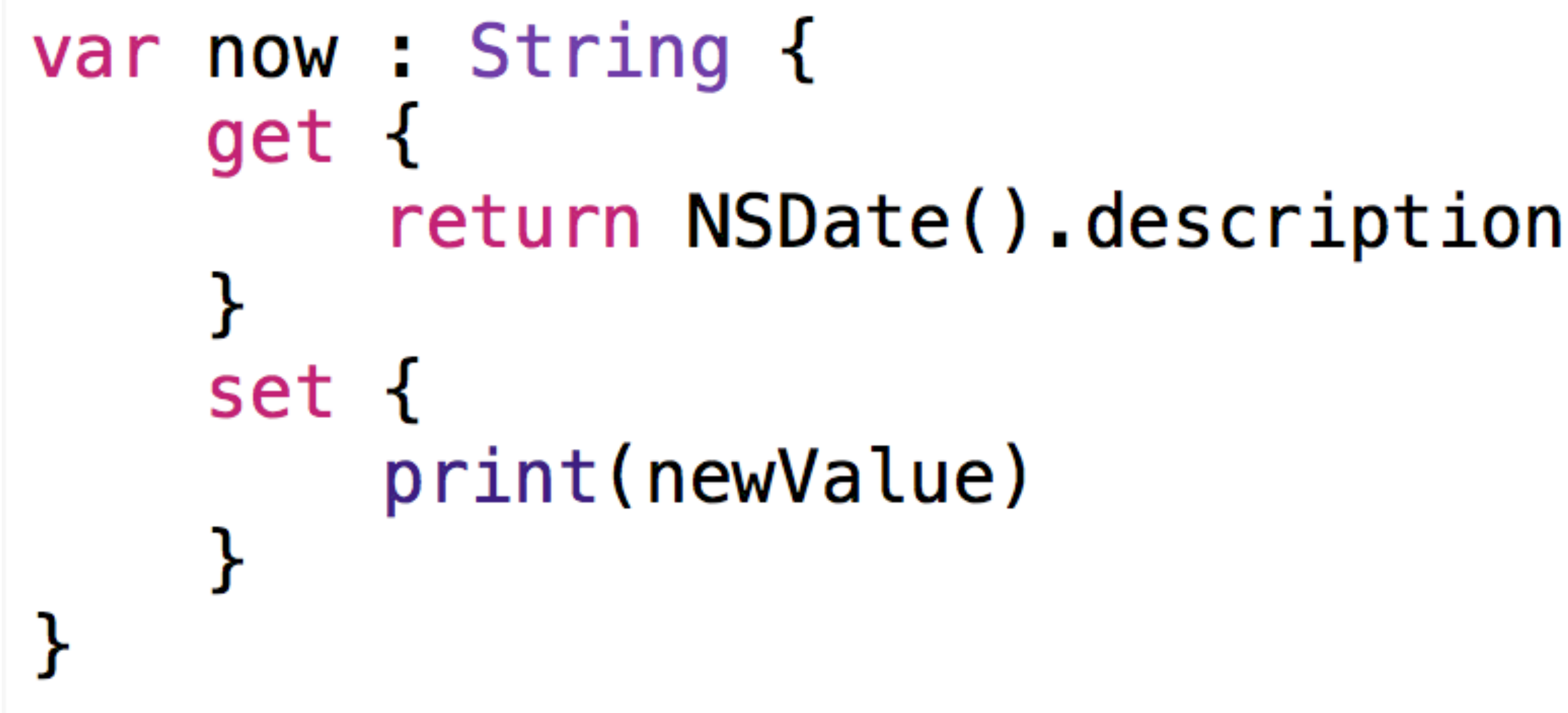
```
content.addAttribute(
    NSParagraphStyleAttributeName,
    value: {
        let para = NSMutableParagraphStyle()
        para.headIndent = 10
        para.firstLineHeadIndent = 10
        // ... more configuration of para ...
        return para
        }(),
    range:NSMakeRange(0,1))
```

Computed Initializer



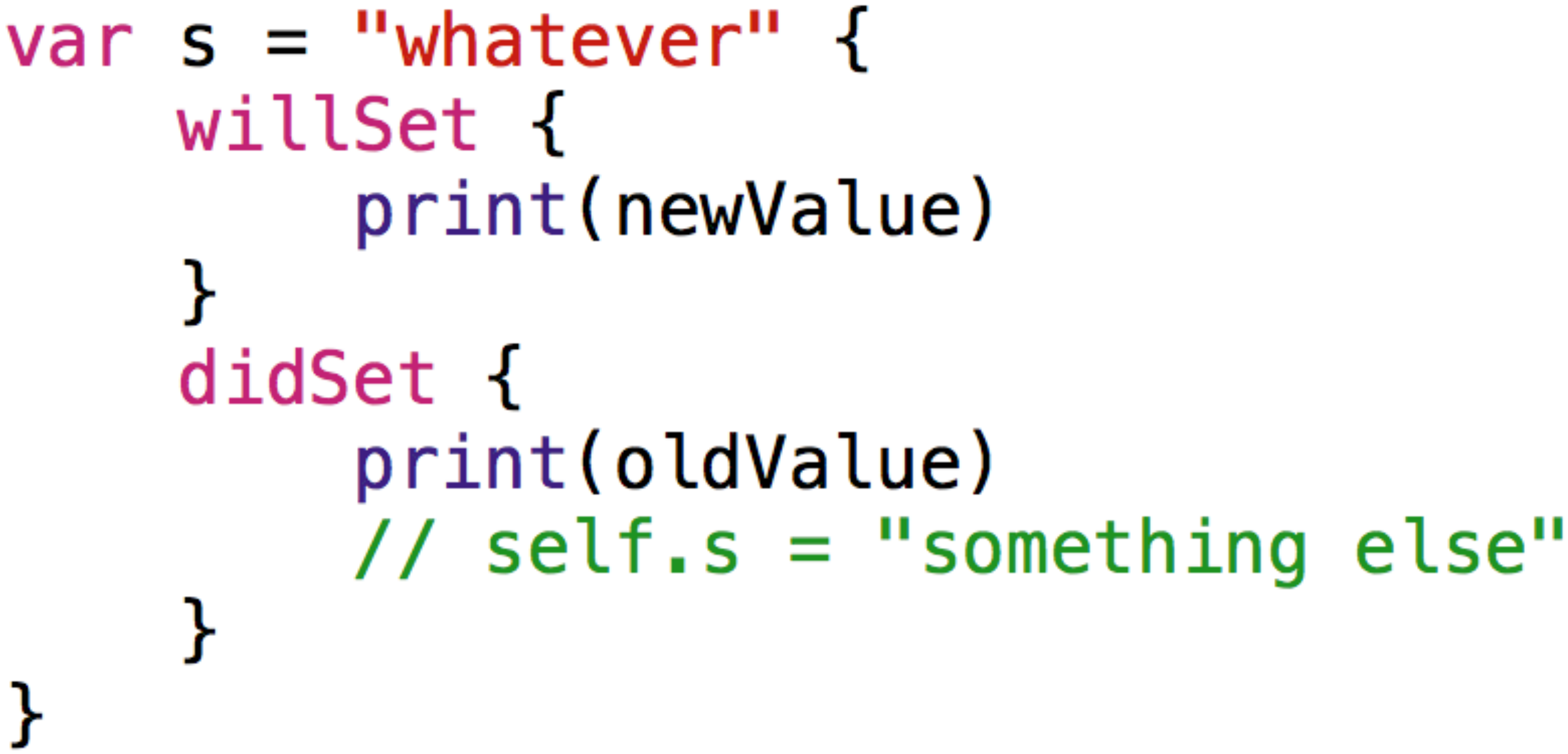
```
let timed : Bool = {
    if val == 1 {
        return true
    } else {
        return false
    }
    }()
```

Computed Variables



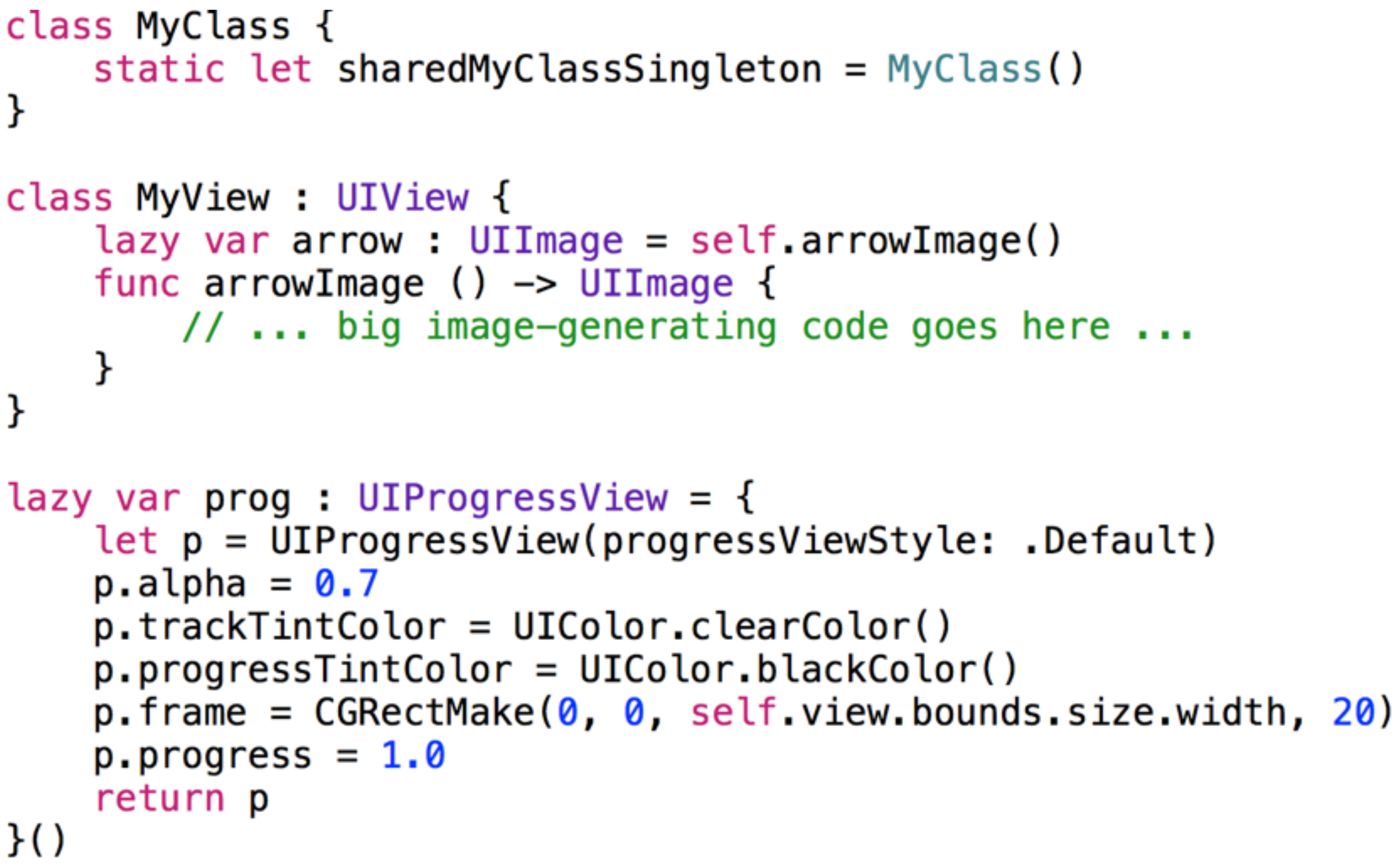
```
var now : String {
    get {
        return NSDate().description
    }
    set {
        print(newValue)
    }
}
```

Setter Observers



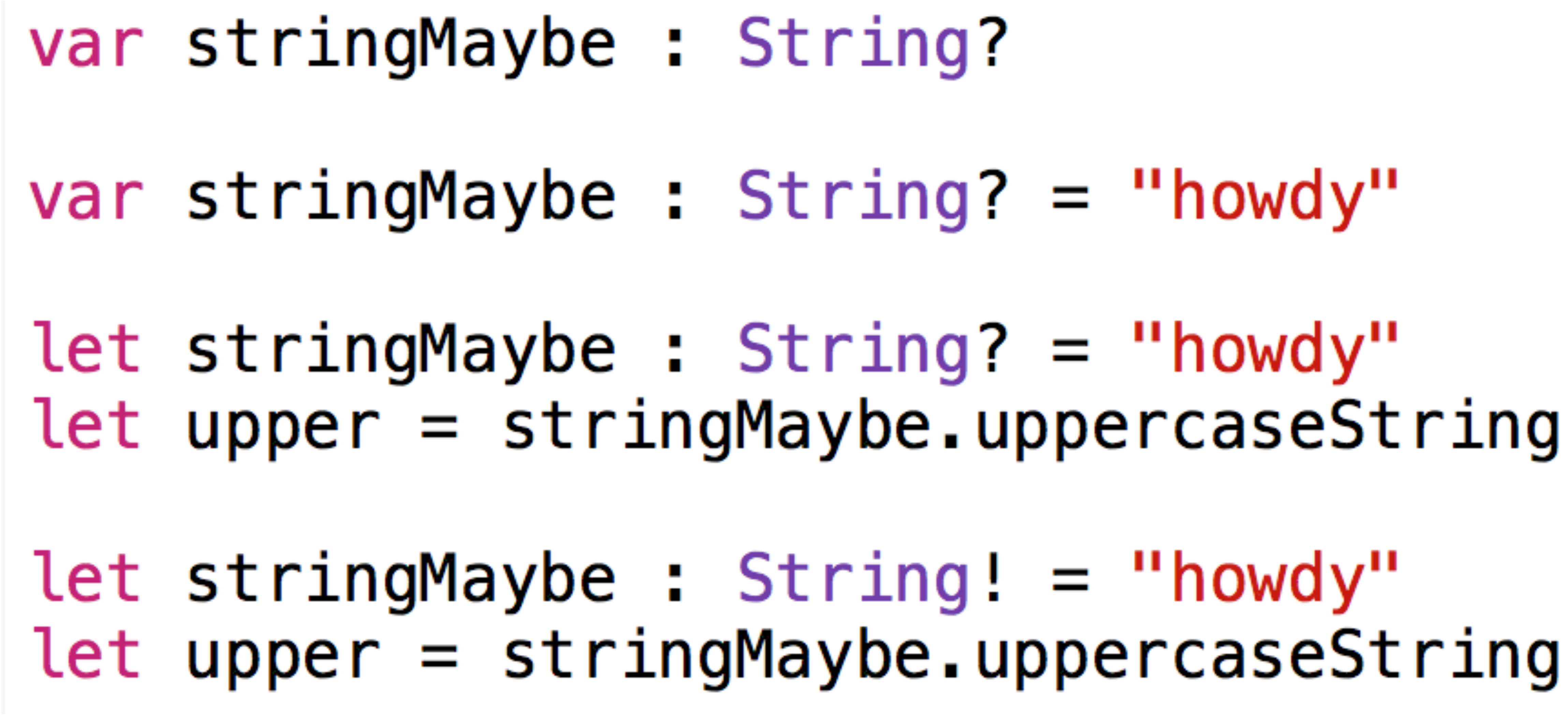
```
var s = "whatever" {
    willSet {
        print(newValue)
    }
    didSet {
        print(oldValue)
        // self.s = "something else"
    }
}
```

Lazy Initialization



```
class MyClass {
    static let sharedMyClassSingleton = MyClass()
}

class MyView : UIView {
    lazy var arrow : UIImage = self.arrowImage()
    func arrowImage () -> UIImage {
        // ... big image-generating code goes here ...
    }
}

lazy var prog : UIProgressView = {
    let p = UIProgressView(progressViewStyle: .Default)
    p.alpha = 0.7
    p.trackTintColor = UIColor.clearColor()
    p.progressTintColor = UIColor.blackColor()
    p.frame = CGRectMake(0, 0, self.view.bounds.size.width, 20)
    p.progress = 1.0
    return p
}()
```

Optionals

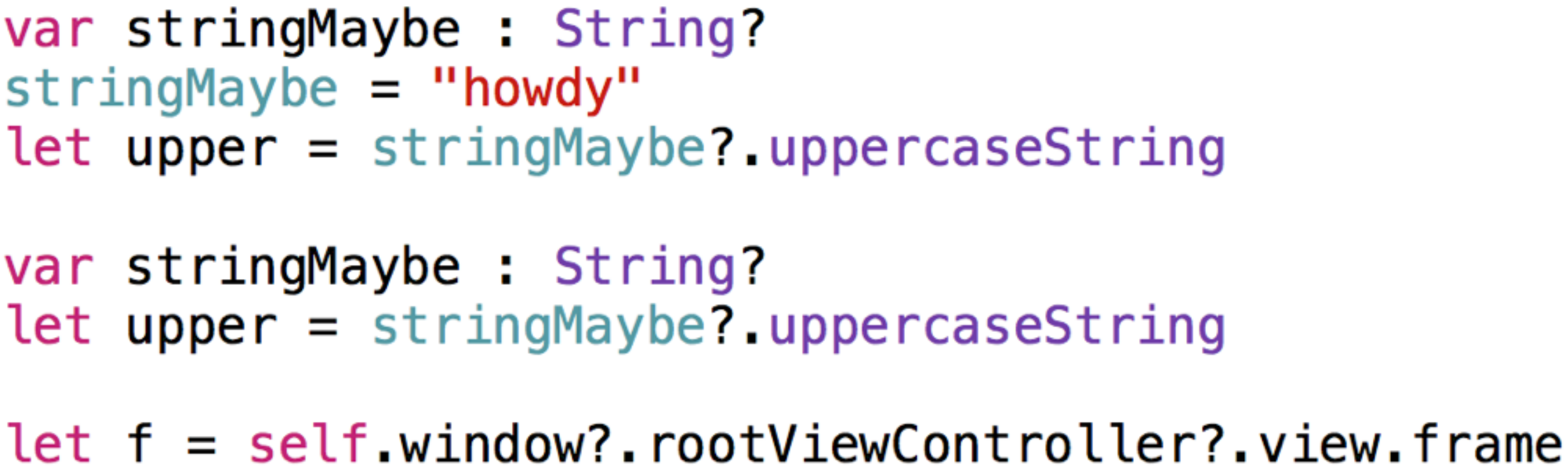


```
var stringMaybe : String?

var stringMaybe : String? = "howdy"

let stringMaybe : String? = "howdy"
let upper = stringMaybe.uppercaseString

let stringMaybe : String! = "howdy"
let upper = stringMaybe.uppercaseString
```

nil

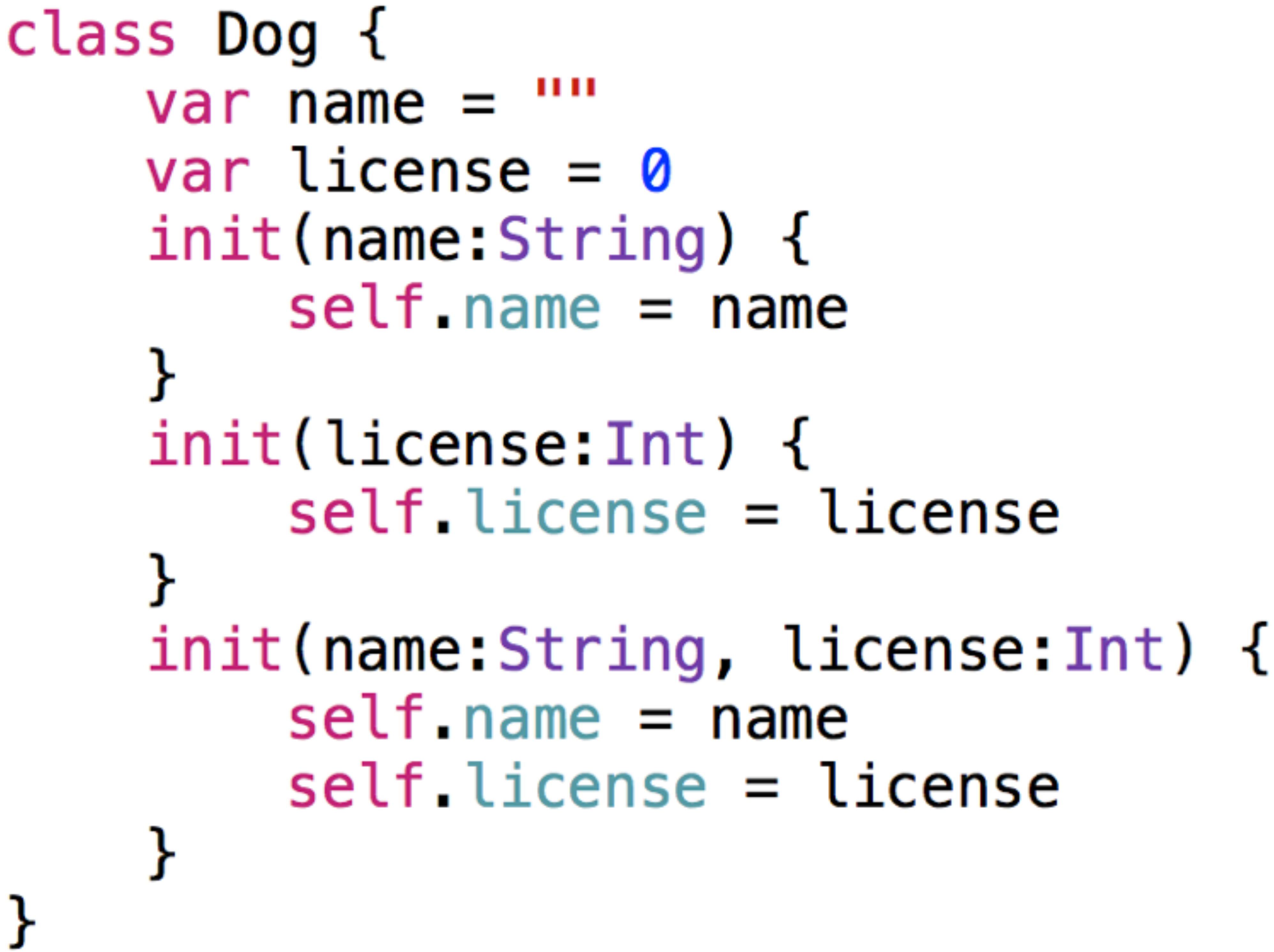
```
var stringMaybe : String? = "Howdy"
print(stringMaybe) // Optional("Howdy")
if stringMaybe == nil {
    print("it is empty") // does not print
}
stringMaybe = nil
print(stringMaybe) // nil
if stringMaybe == nil {
    print("it is empty") // prints
}
```

Optional chains



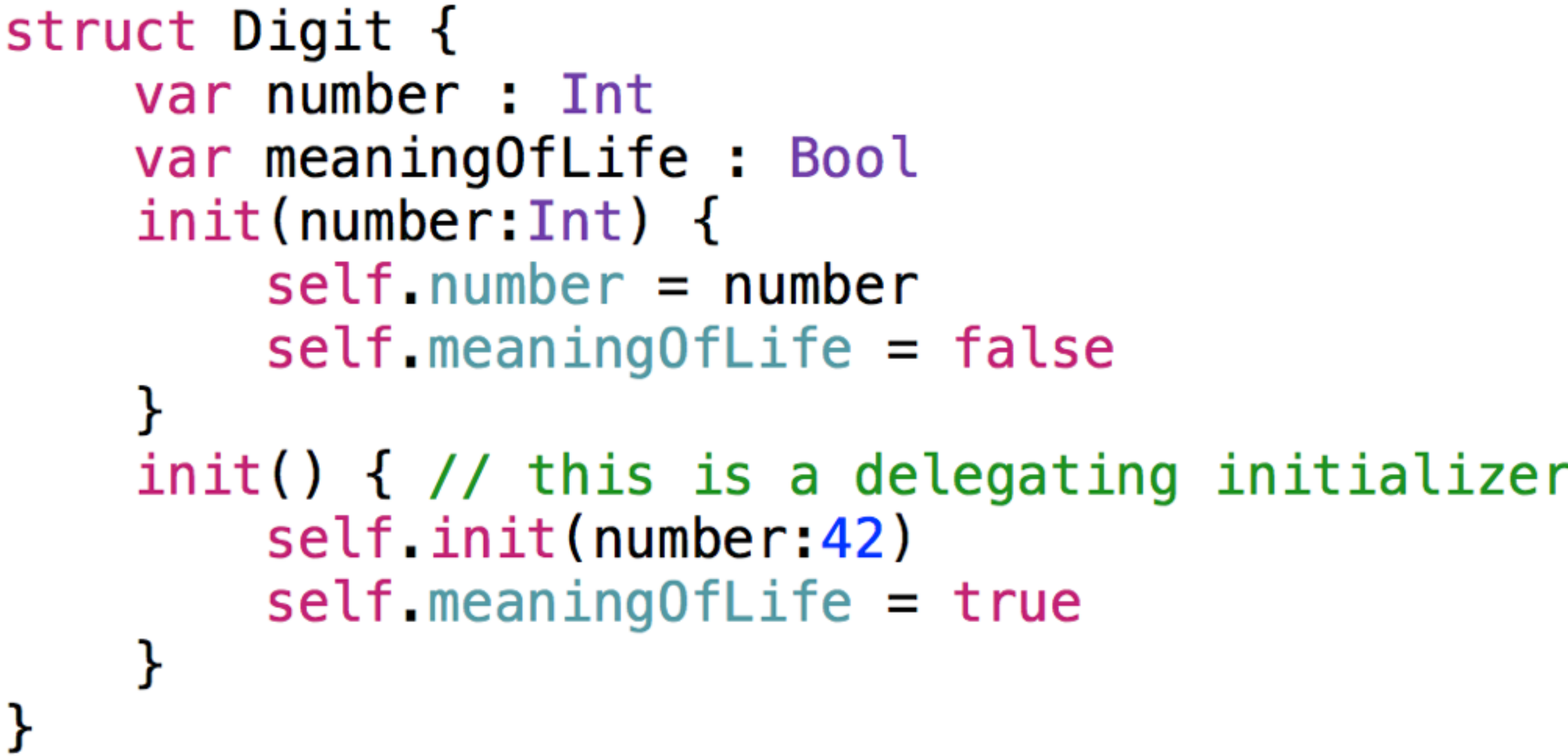
```
var stringMaybe : String?
stringMaybe = "howdy"
let upper = stringMaybe?.uppercaseString

var stringMaybe : String?
let upper = stringMaybe?.uppercaseString

let f = self.window?.rootViewController?.view.frame
```

Initializers



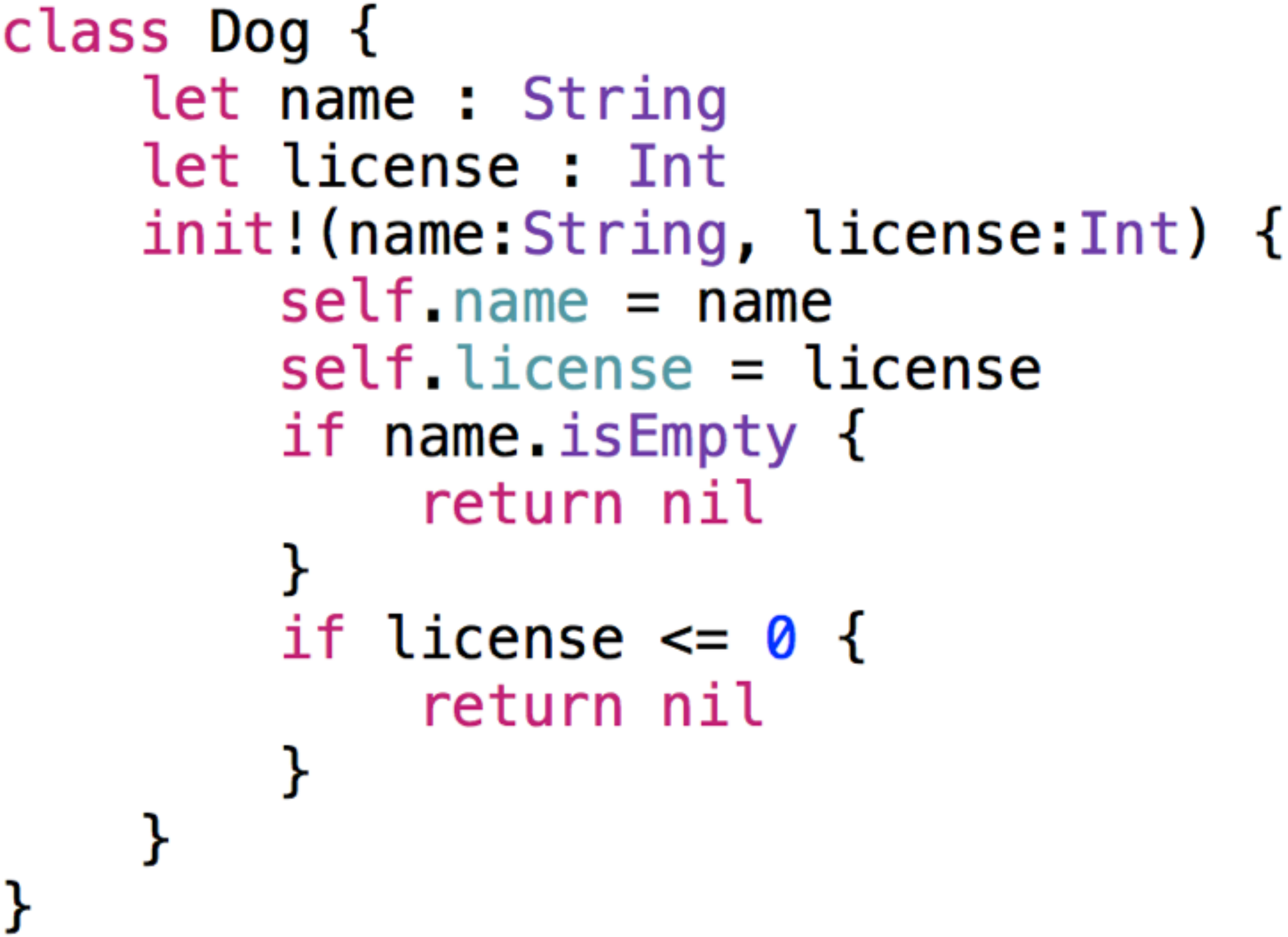
```
class Dog {
    var name = ""
    var license = 0
    init(name:String) {
        self.name = name
    }
    init(license:Int) {
        self.license = license
    }
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

Delegating initializers



```
struct Digit {
    var number : Int
    var meaningOfLife : Bool
    init(number:Int) {
        self.number = number
        self.meaningOfLife = false
    }
    init() { // this is a delegating initializer
        self.init(number:42)
        self.meaningOfLife = true
    }
}
```

Failable initializers



```
class Dog {
    let name : String
    let license : Int
    init!(name:String, license:Int) {
        self.name = name
        self.license = license
        if name.isEmpty {
            return nil
        }
        if license <= 0 {
            return nil
        }
    }
}
```

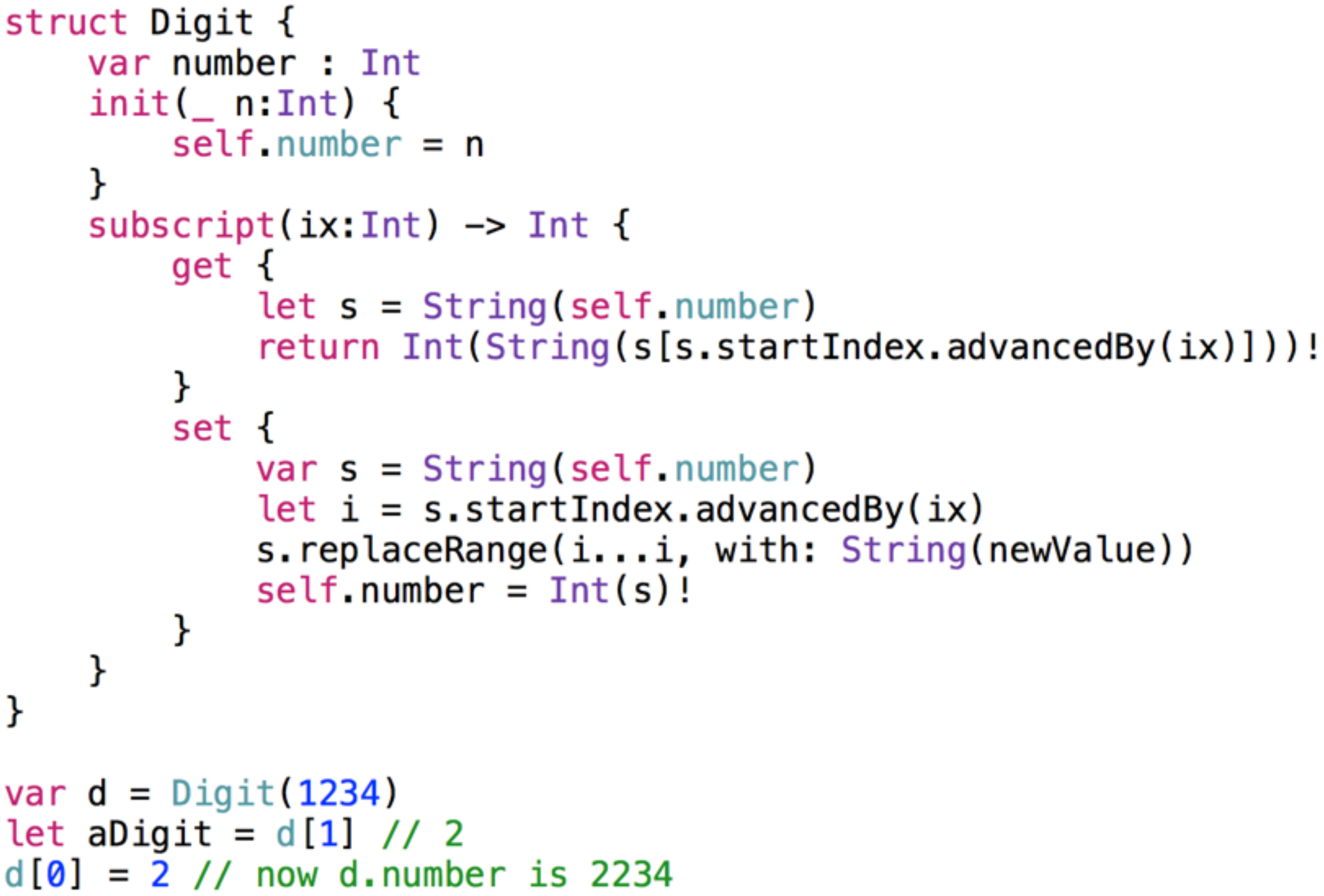
What are you required to do inside init?

- By the time any init is done, all properties must have values (optionals can have the value nil)
- There are two types of inits in a **class**, **convenience** and **designated** (i.e. not convenience)
- A designated init must (and can only) call a designated **init** that is in its immediate **superclass**
- You must initialize all properties <u>introduced by your class</u> before calling a superclass's init
- You must call a superclass's init before you assign a value to an inherited property
- A **convenience** init must (and can only) call a designated init in its own class
- A **convenience** init may call a designated init indirectly (through another **convenience** init)
- A **convenience** init must call a designated init before it can set any property values
- The calling of other inits must be complete before you can access properties or invoke methods

What are you required to do inside init?

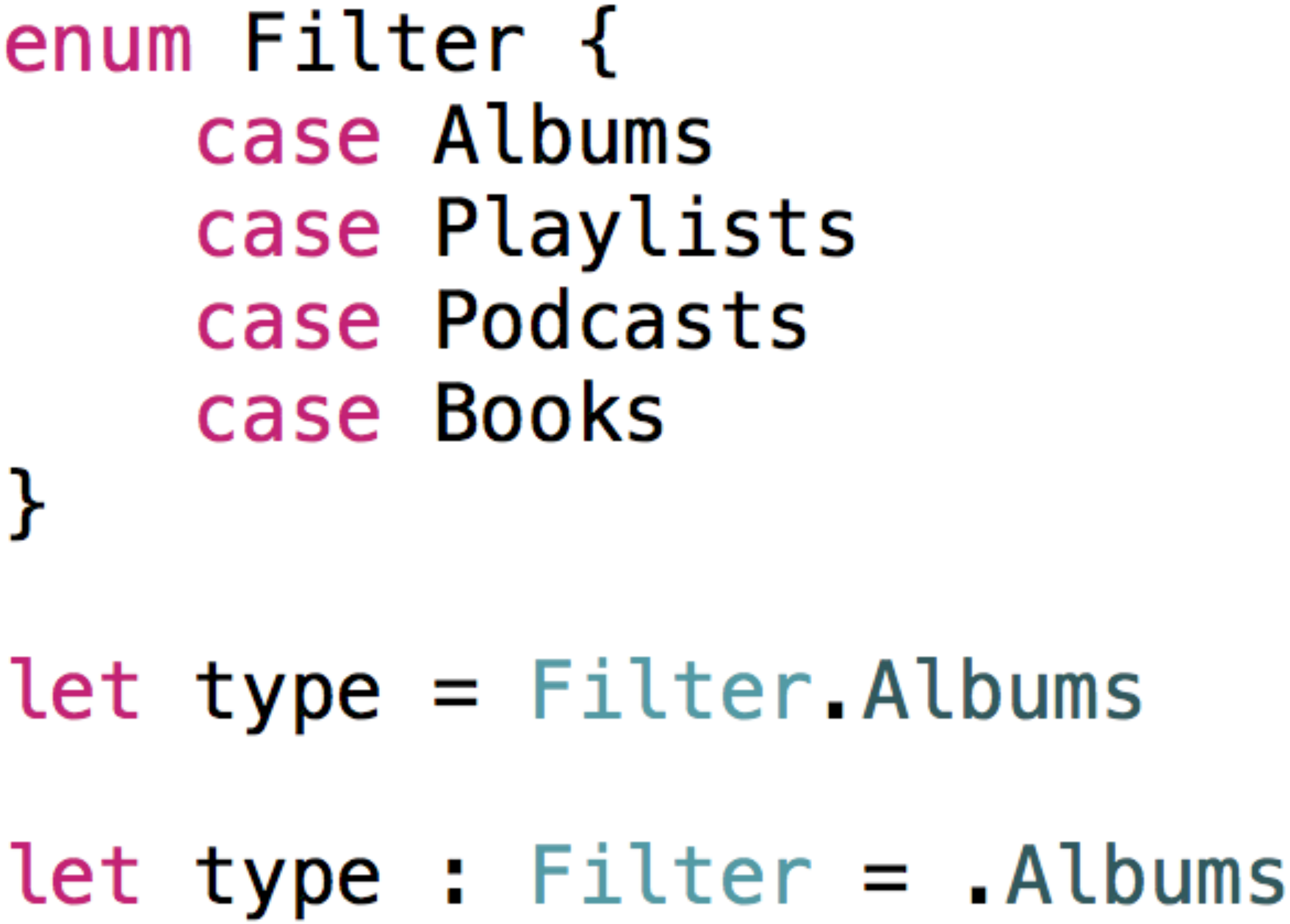
- Inheriting init
  - If you do not implement any designated inits, you'll inherit all of your superclass's designateds
  - If you override all of your superclass's designated inits, you'll inherit all its convenience inits
  - If you implement no inits, you'll inherit all of your superclass's inits
  - Any init inherited by these rules qualifies to satisfy any of the rules on the previous slide
- Required init
  - A class can mark one or more of its init methods as required
  - Any subclass must implement said init methods (though they can be inherited per above rules)

Subscripts



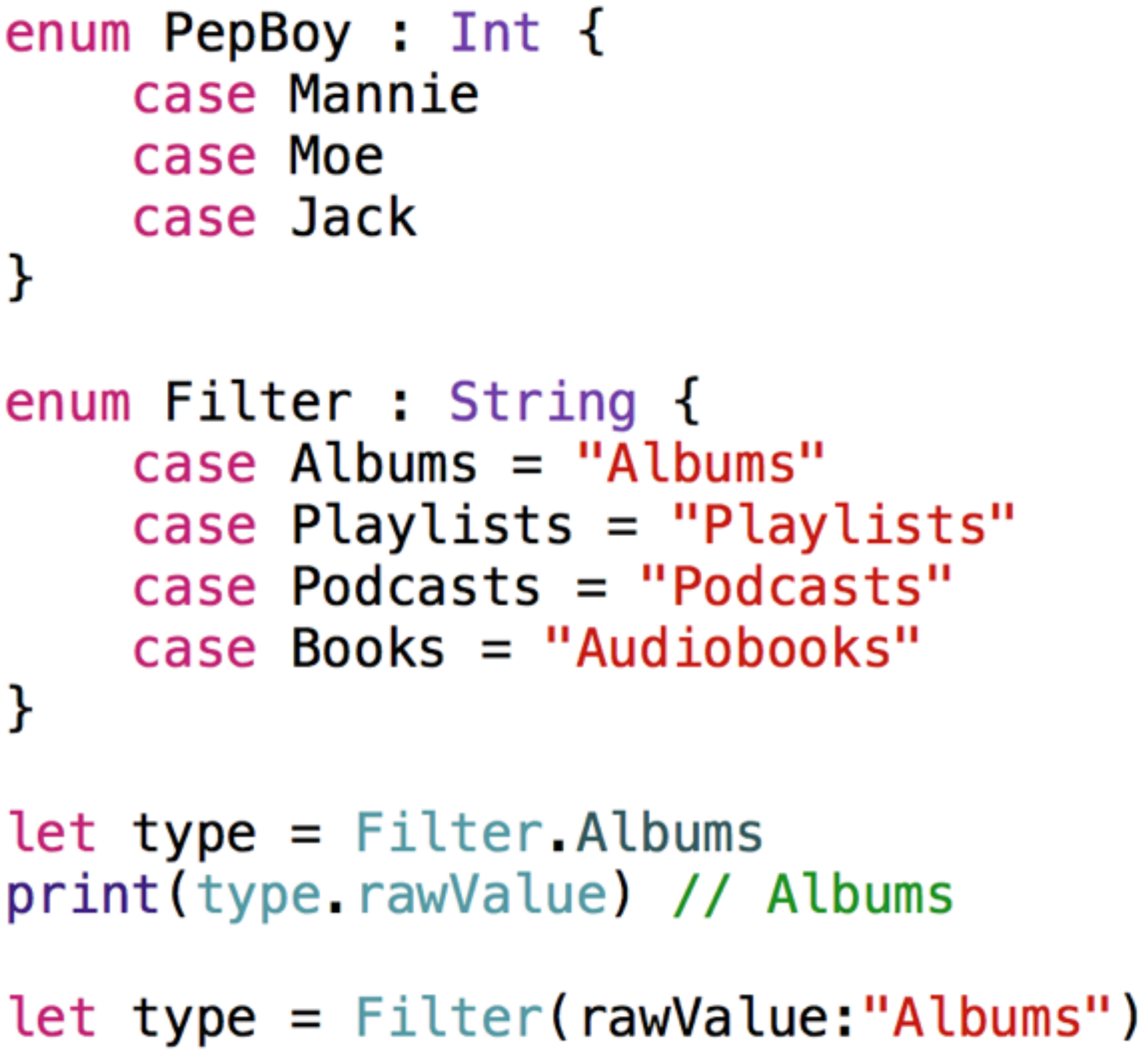
```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
    subscript(ix:Int) -> Int {
        get {
            let s = String(self.number)
            return Int(String(s[s.startIndex.advancedBy(ix)]))!
        }
        set {
            var s = String(self.number)
            let i = s.startIndex.advancedBy(ix)
            s.replaceRange(i...i, with: String(newValue))
            self.number = Int(s)!
        }
    }
}

var d = Digit(1234)
let aDigit = d[1] // 2
d[0] = 2 // now d.number is 2234
```

Enums



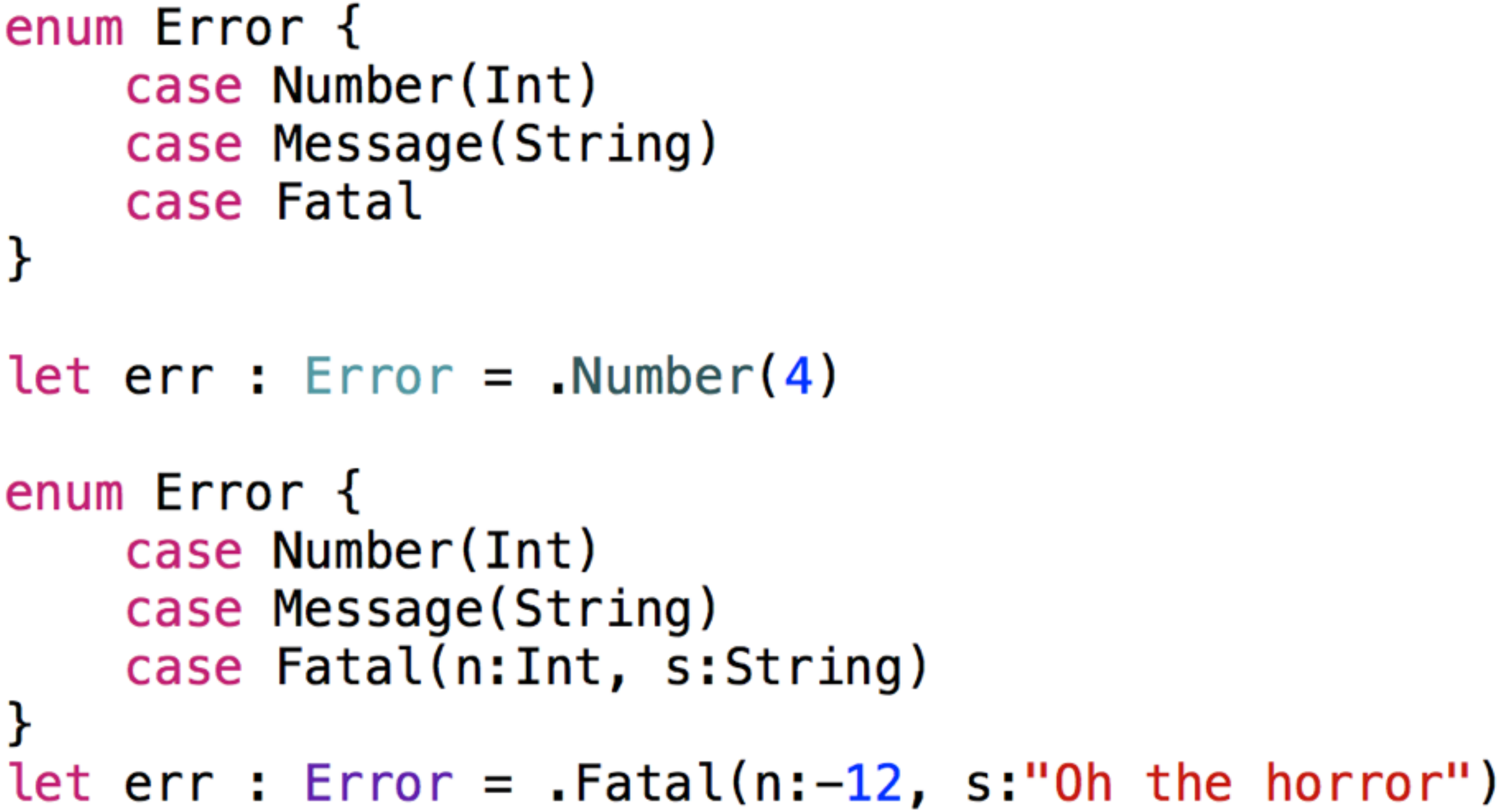
```
enum Filter {
    case Albums
    case Playlists
    case Podcasts
    case Books
}

let type = Filter.Albums

let type : Filter = .Albums
```

Case With Fixed Value



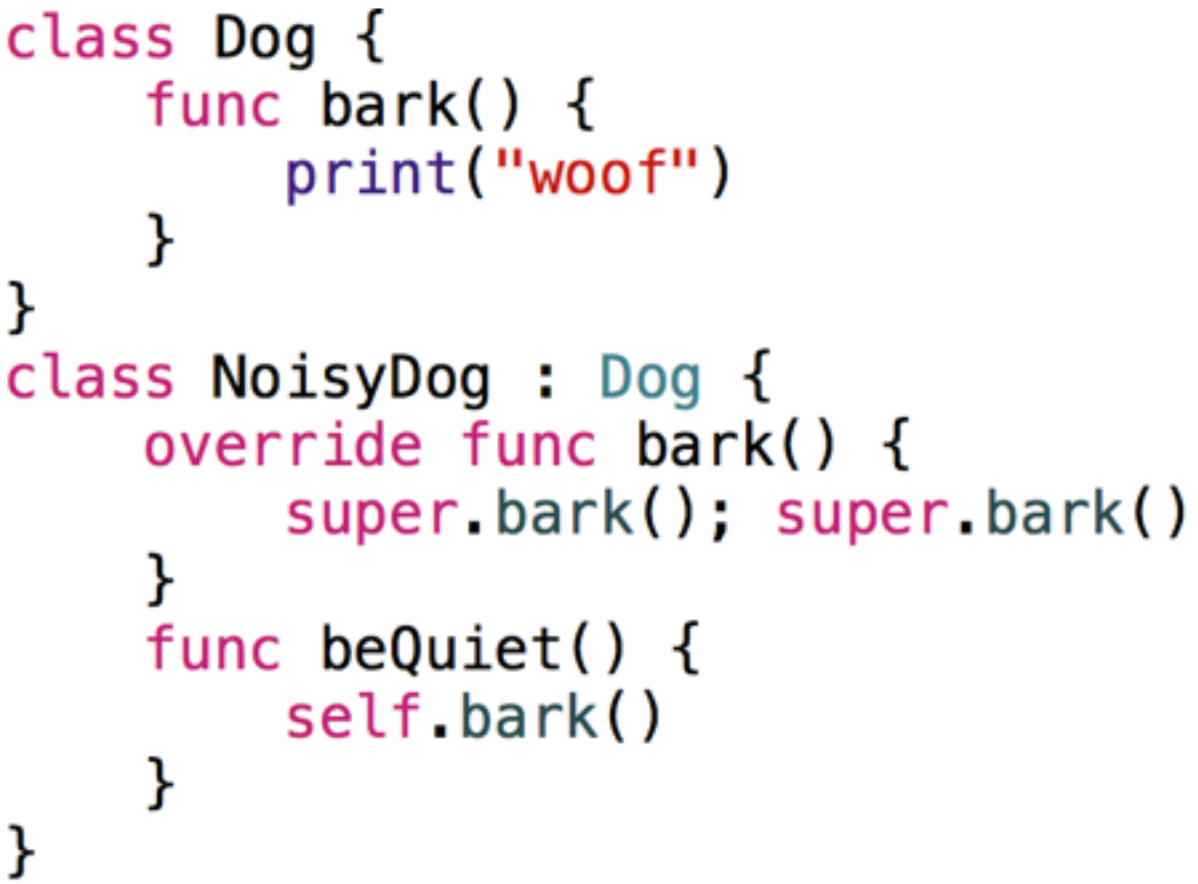
```
enum PepBoy : Int {
    case Mannie
    case Moe
    case Jack
}

enum Filter : String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
}

let type = Filter.Albums
print(type.rawValue) // Albums

let type = Filter(rawValue:"Albums")
```

Case With Typed Value

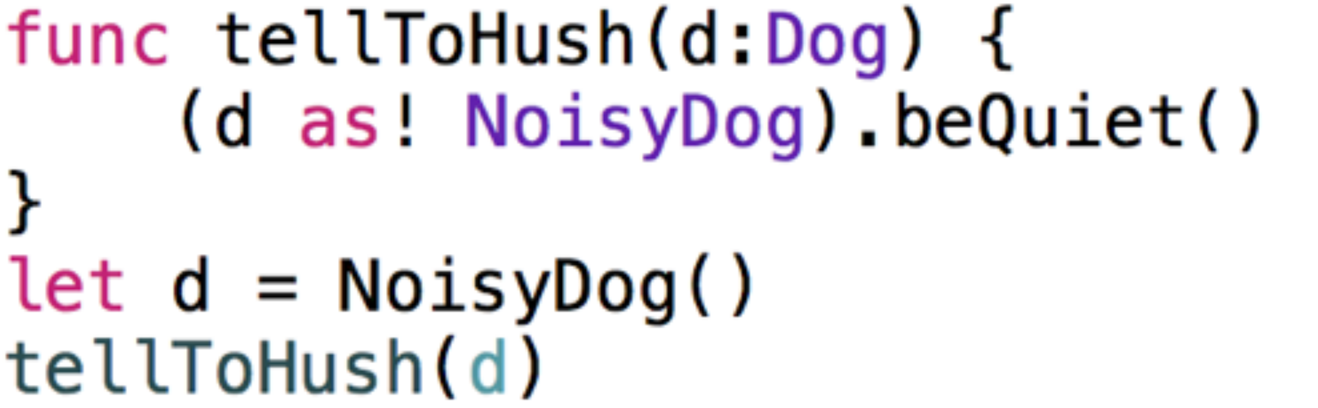


```
enum Error {
    case Number(Int)
    case Message(String)
    case Fatal
}

let err : Error = .Number(4)

enum Error {
    case Number(Int)
    case Message(String)
    case Fatal(n:Int, s:String)
}
let err : Error = .Fatal(n:-12, s:"Oh the horror")
```

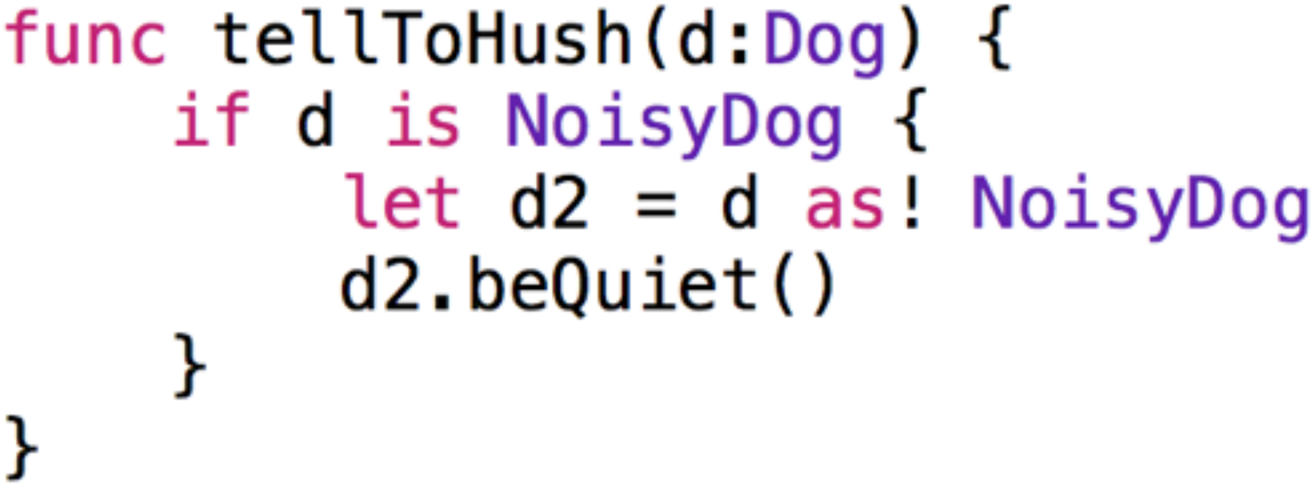
Casting



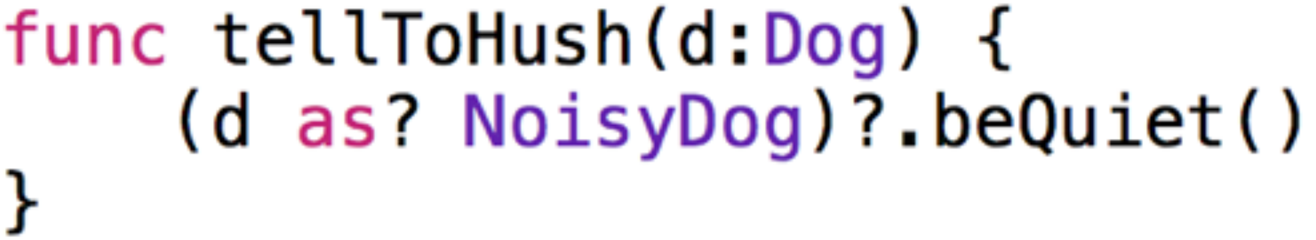
```
class Dog {
    func bark() {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark() {
        super.bark(); super.bark()
    }
    func beQuiet() {
        self.bark()
    }
}
```



```
func tellToHush(d:Dog) {
    (d as! NoisyDog).beQuiet()
}
let d = NoisyDog()
tellToHush(d)
```

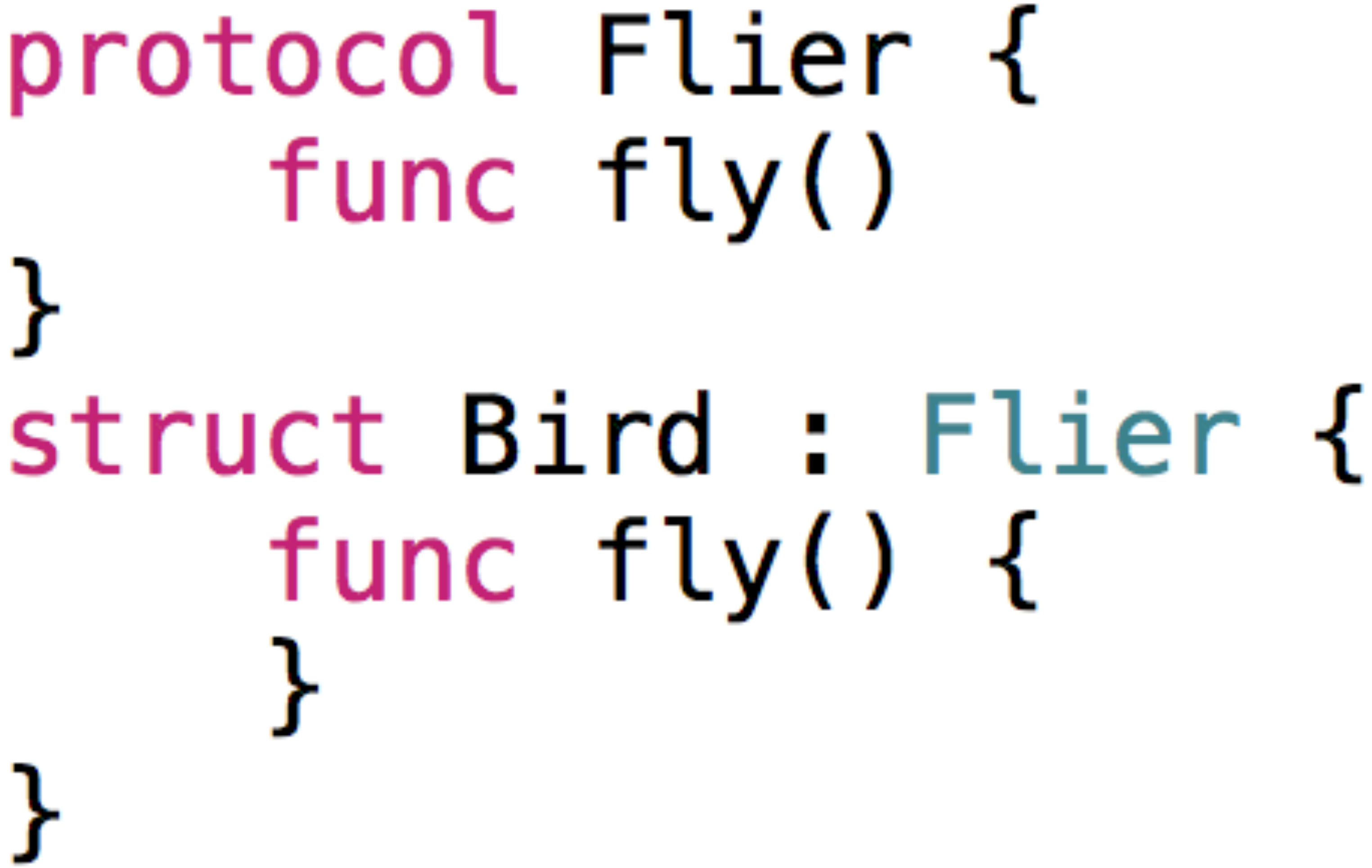


```
func tellToHush(d:Dog) {
    if d is NoisyDog {
        let d2 = d as! NoisyDog
        d2.beQuiet()
    }
}
```



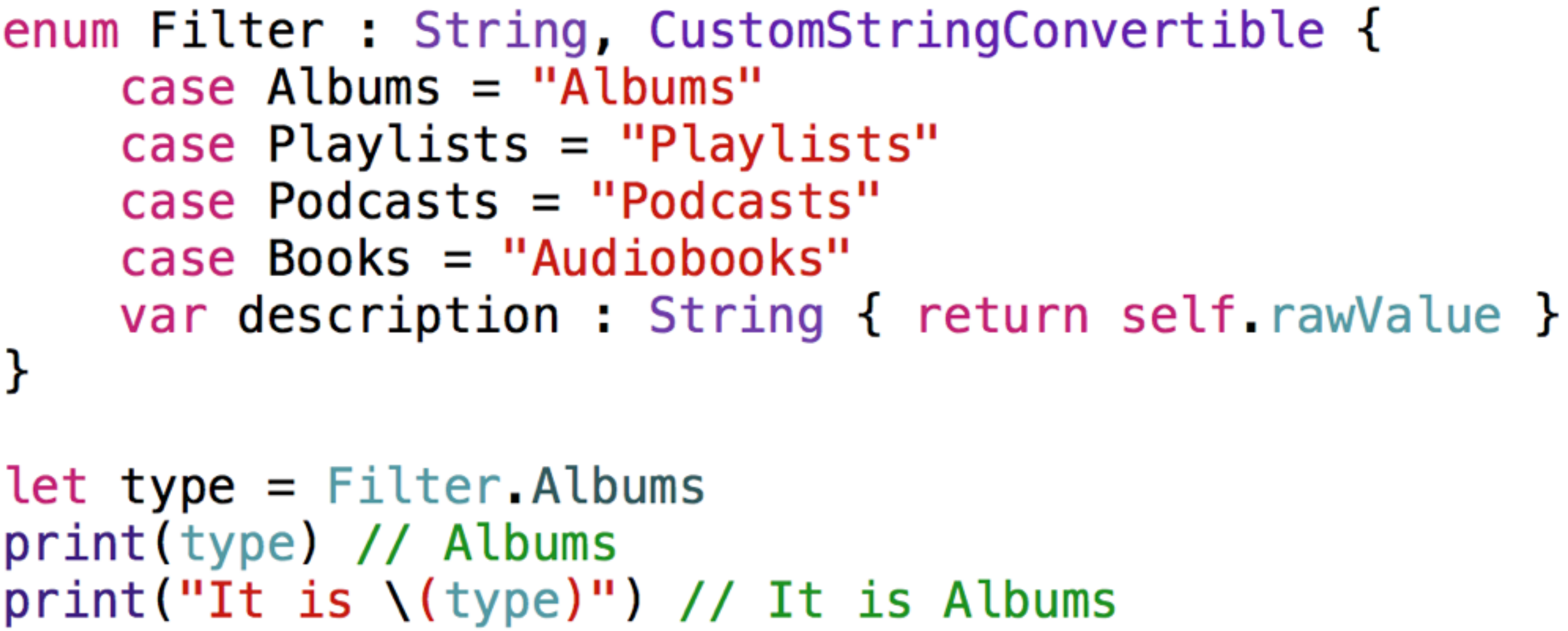
```
func tellToHush(d:Dog) {
    (d as? NoisyDog)?.beQuiet()
}
```

Protocols



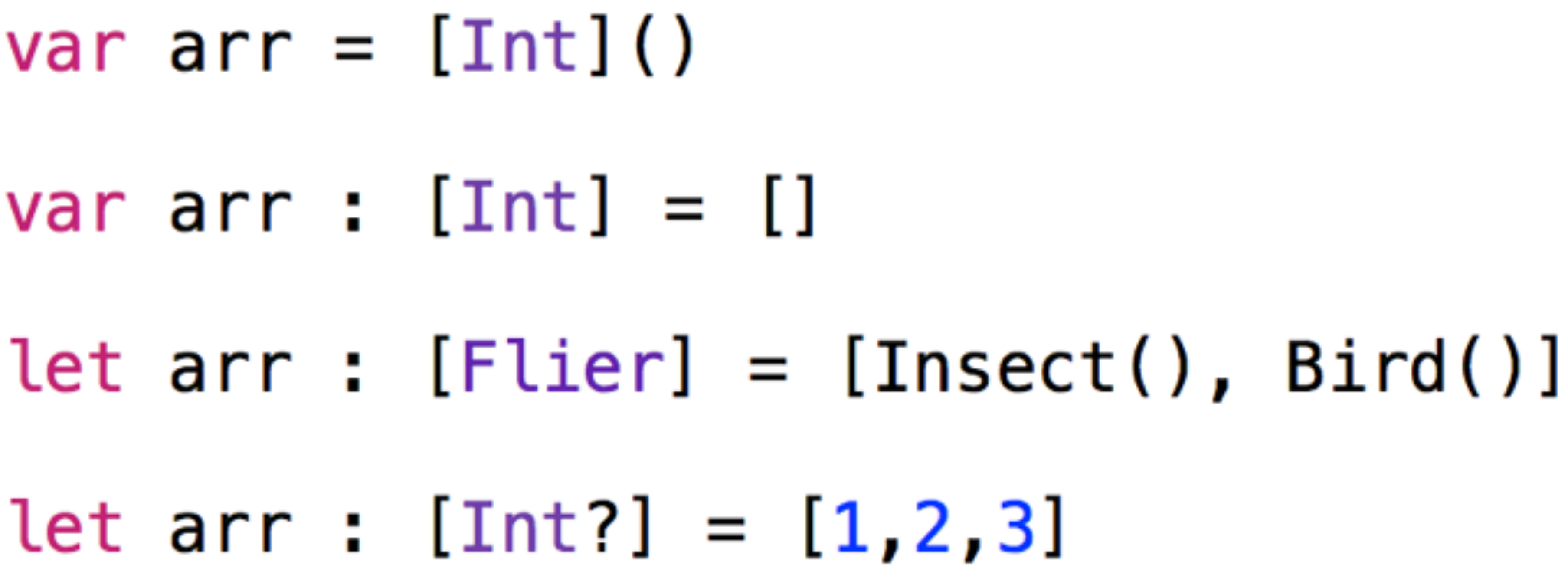
```
protocol Flier {
    func fly()
}
struct Bird : Flier {
    func fly() {
    }
}
```

CustomStringConvertible protocol

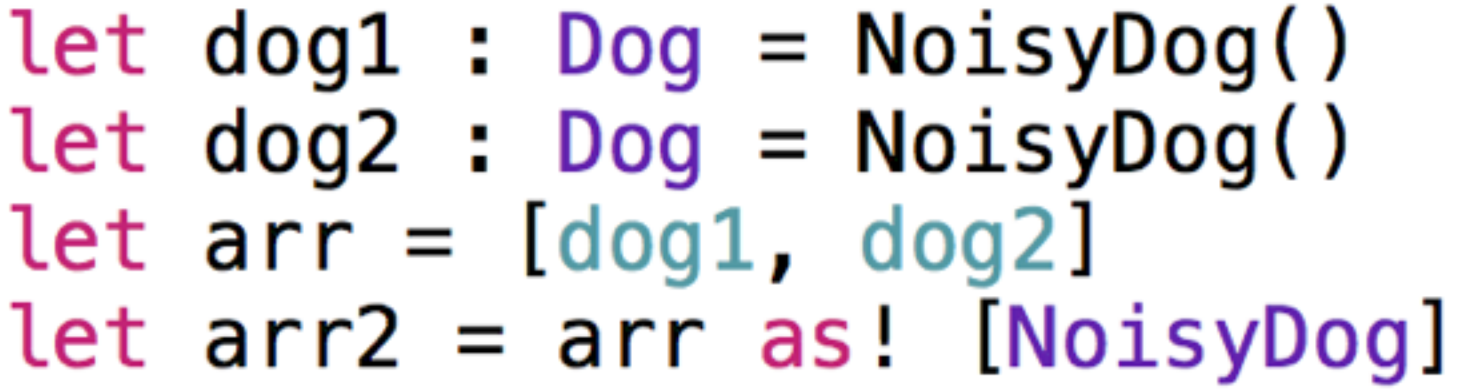


```swift
enum Filter : String, CustomStringConvertible {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    var description : String { return self.rawValue }
}

let type = Filter.Albums
print(type) // Albums
print("It is \(type)") // It is Albums
```

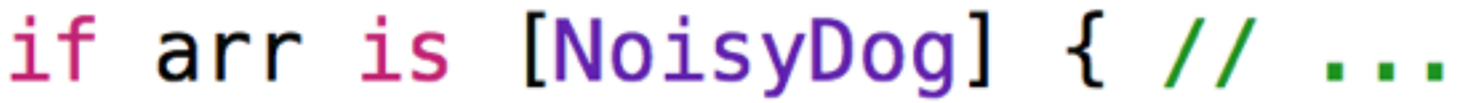
Arrays



```swift
var arr = [Int]()

var arr : [Int] = []

let arr : [Flier] = [Insect(), Bird()]

let arr : [Int?] = [1,2,3]
```

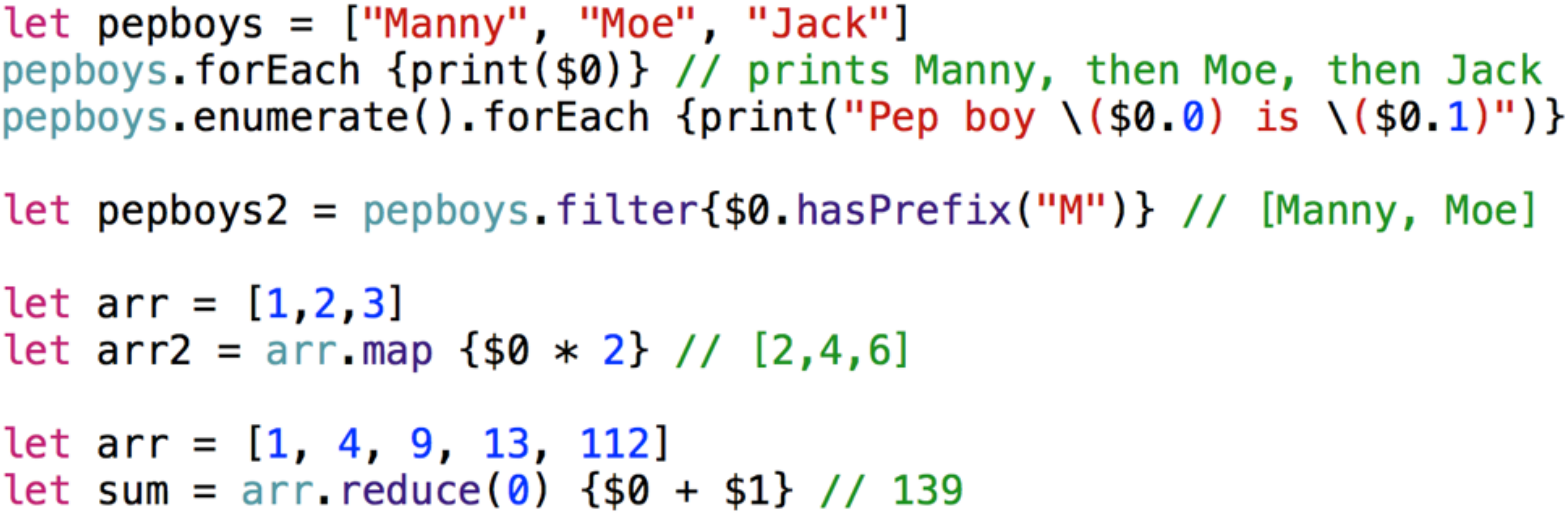


```swift
let dog1 : Dog = NoisyDog()
let dog2 : Dog = NoisyDog()
let arr = [dog1, dog2]
let arr2 = arr as! [NoisyDog]
```



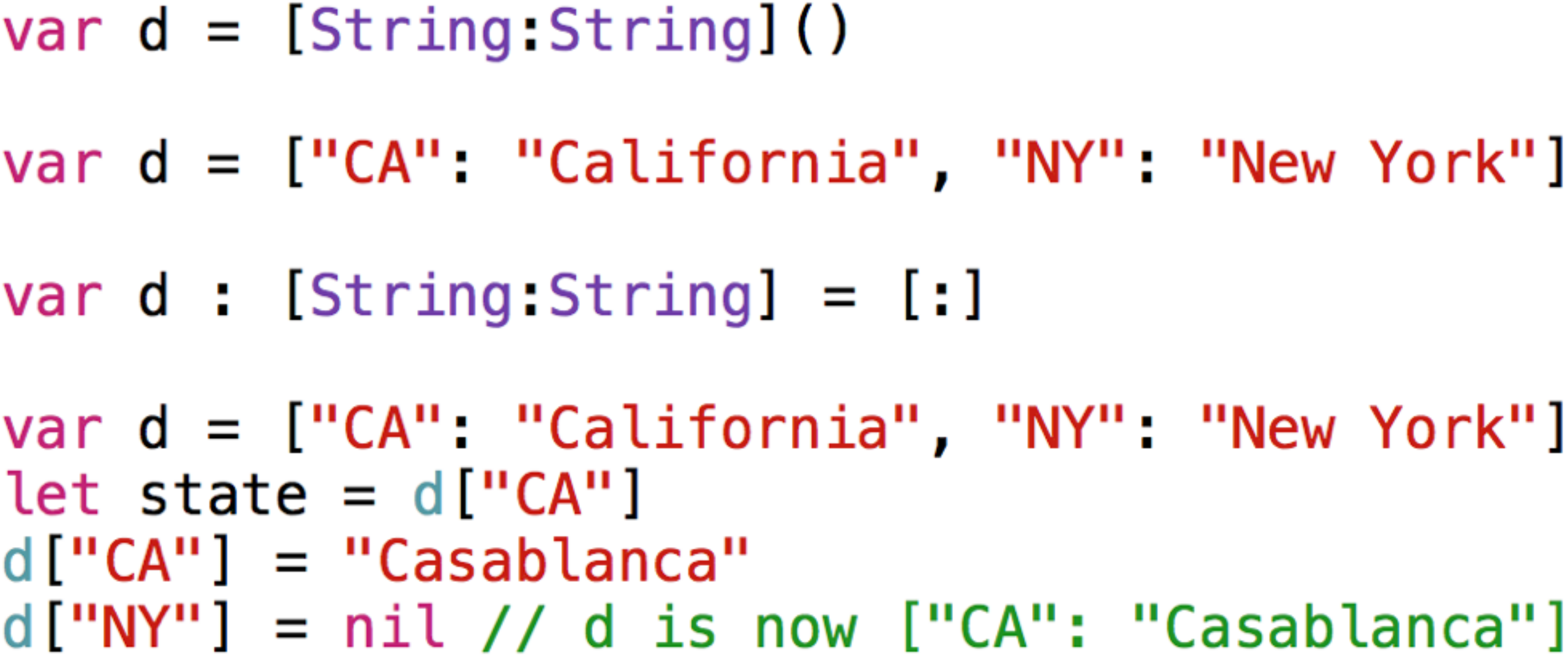
```swift
if arr is [NoisyDog] { // ...
```

Array enumeration and transformation



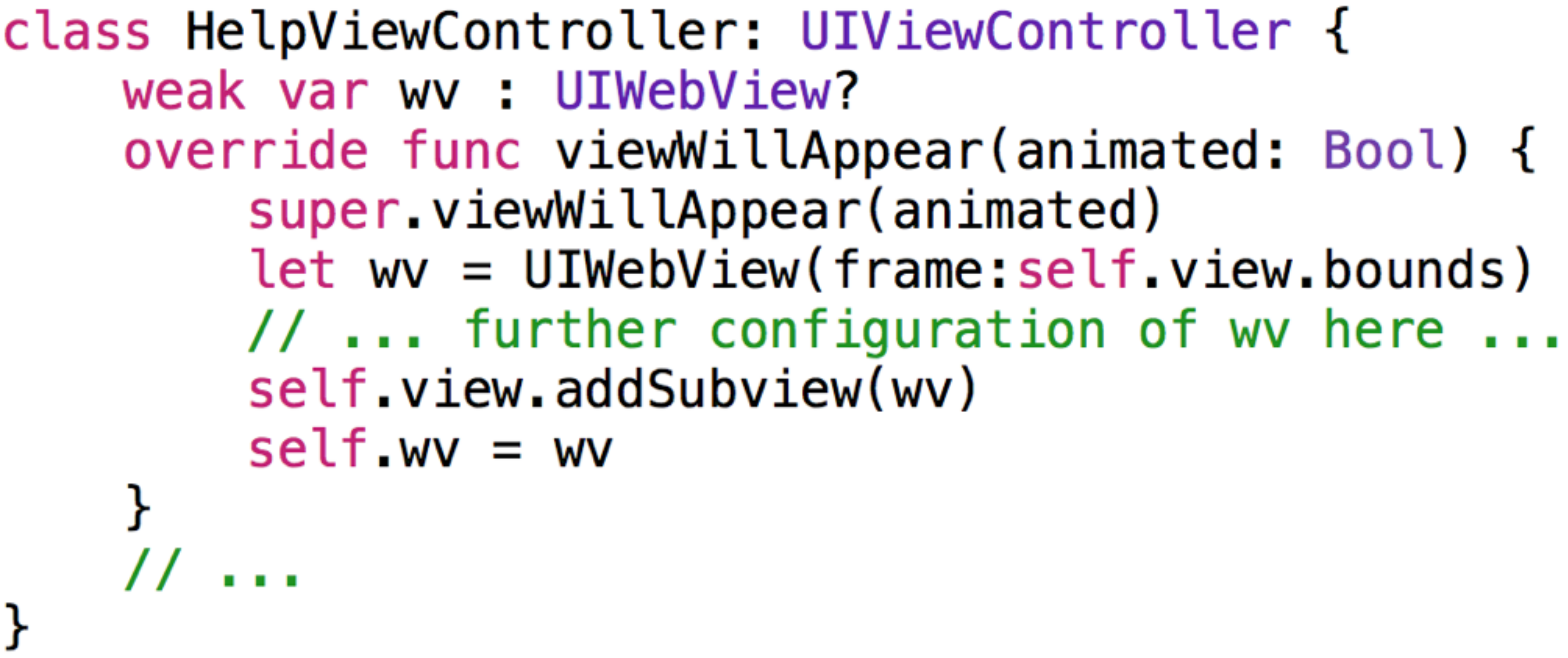
```
let pepboys = ["Manny", "Moe", "Jack"]
pepboys.forEach {print($0)} // prints Manny, then Moe, then Jack
pepboys.enumerate().forEach {print("Pep boy \($0.0) is \($0.1)")}

let pepboys2 = pepboys.filter{$0.hasPrefix("M")} // [Manny, Moe]

let arr = [1,2,3]
let arr2 = arr.map {$0 * 2} // [2,4,6]

let arr = [1, 4, 9, 13, 112]
let sum = arr.reduce(0) {$0 + $1} // 139
```

Dictionary



```
var d = [String:String]()

var d = ["CA": "California", "NY": "New York"]

var d : [String:String] = [:]

var d = ["CA": "California", "NY": "New York"]
let state = d["CA"]
d["CA"] = "Casablanca"
d["NY"] = nil // d is now ["CA": "Casablanca"]
```

Weak References



```swift
class HelpViewController: UIViewController {
    weak var wv : UIWebView?
    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        let wv = UIWebView(frame:self.view.bounds)
        // ... further configuration of wv here ...
        self.view.addSubview(wv)
        self.wv = wv
    }
    // ...
}
```

The Swift Programming Language

Swift 2 Edition

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/