GENERAL ASSEMBLY

MOBILE DEVELOPMENT

OPTIONALS + UITABLEVIEWS

Kishin Manglani

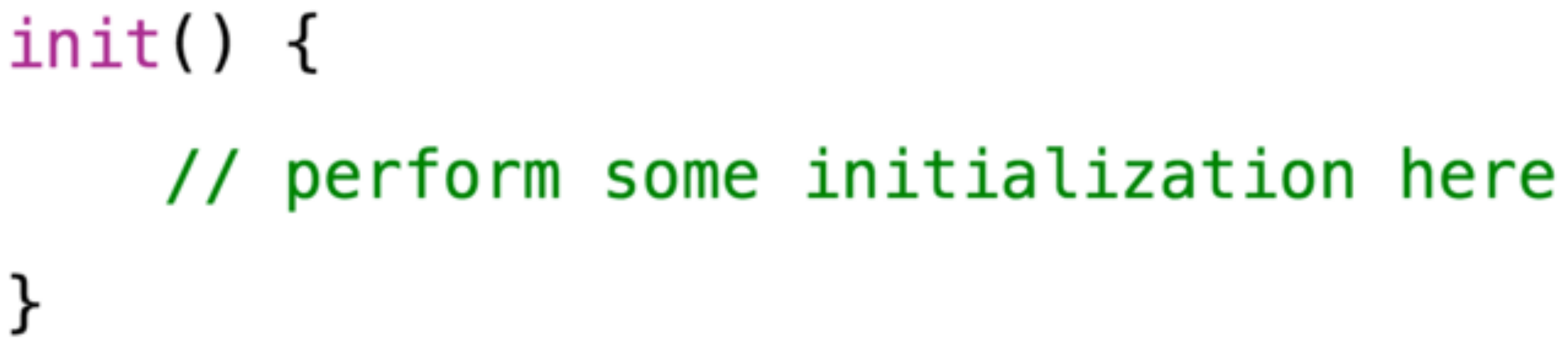
AGENDA

- Recap
- Optionals
- UITableView

RECAP

INITIALIZERS

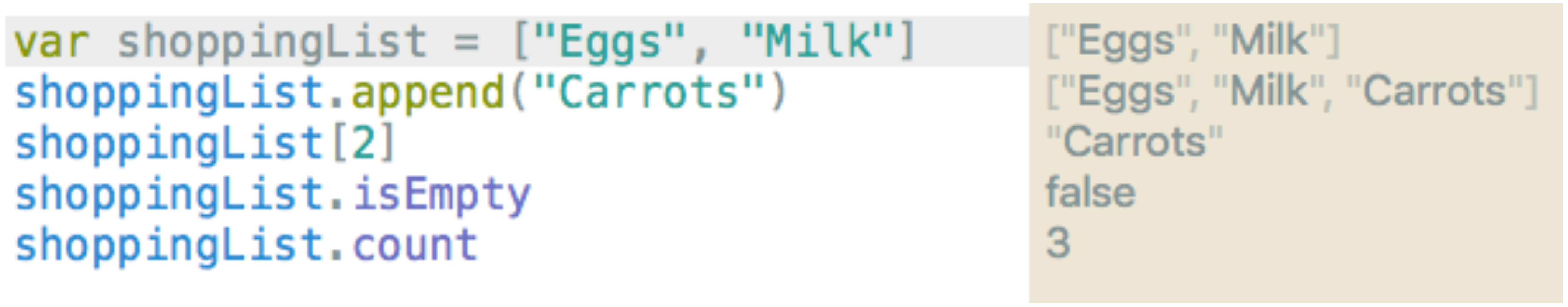
- Initializers are called to create a new instance of a particular type. In its simplest form, an initializer is like an instance method with no parameters, written using the init keyword



```
init() {
    // perform some initialization here
}
```

ARRAY

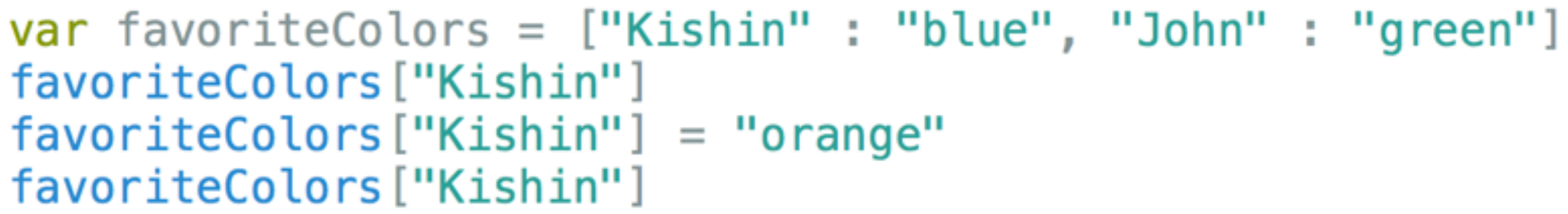
- Arrays have some useful properties
  - count: returns the number (Int) of objects in the array
  - isEmpty: returns a Bool checking if the array is empty (0 objects)
- Append method: adds an object to the array
  - shoppingList.append("carrots")



```
var shoppingList = ["Eggs", "Milk"]          ["Eggs", "Milk"]
shoppingList.append("Carrots")               ["Eggs", "Milk", "Carrots"]
shoppingList[2]                              "Carrots"
shoppingList.isEmpty                         false
shoppingList.count                           3
```

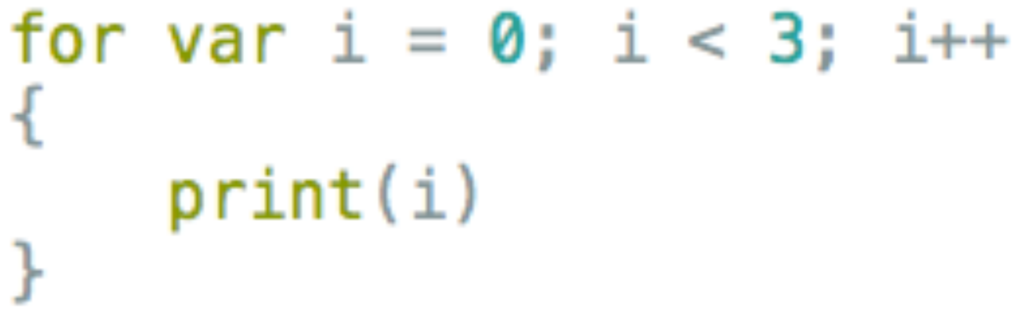
DICTIONARIES

- Associates keys of the same type with values of the same type
- No defined/specified ordering
- Each value is associated with a unique key, which acts as an identifier for that value within the dictionary
- Similar to a traditional dictionary: we use a key (word) to look up a value (definition)
- Provides more meaning than merely index numbers



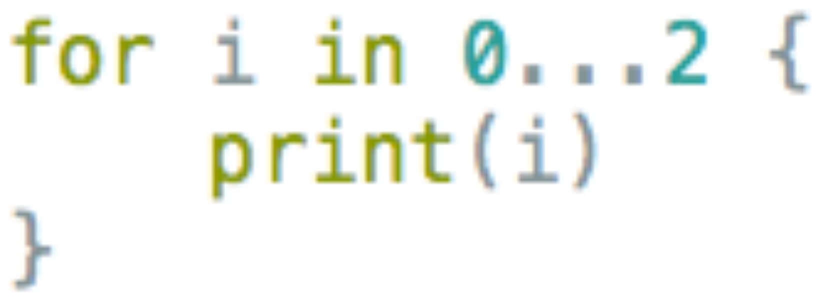
```
var favoriteColors = ["Kishin" : "blue", "John" : "green"]
favoriteColors["Kishin"]
favoriteColors["Kishin"] = "orange"
favoriteColors["Kishin"]
```

FOR LOOPS



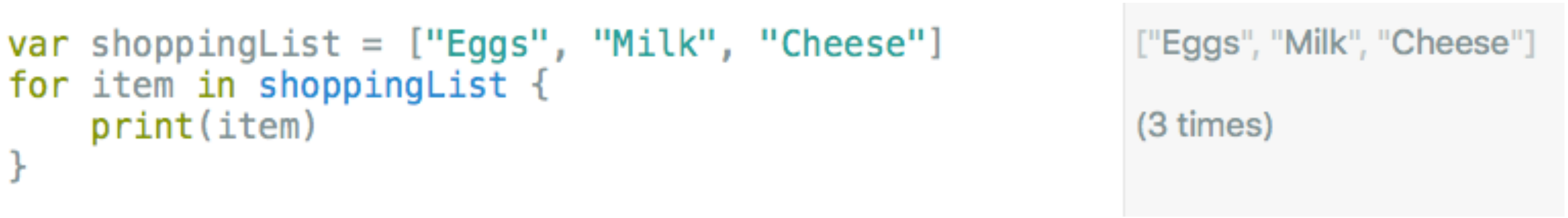
```
for var i = 0; i < 3; i++
{
    print(i)
}
```

FOR LOOPS



```
for i in 0...2 {
    print(i)
}
```

FOR LOOPS



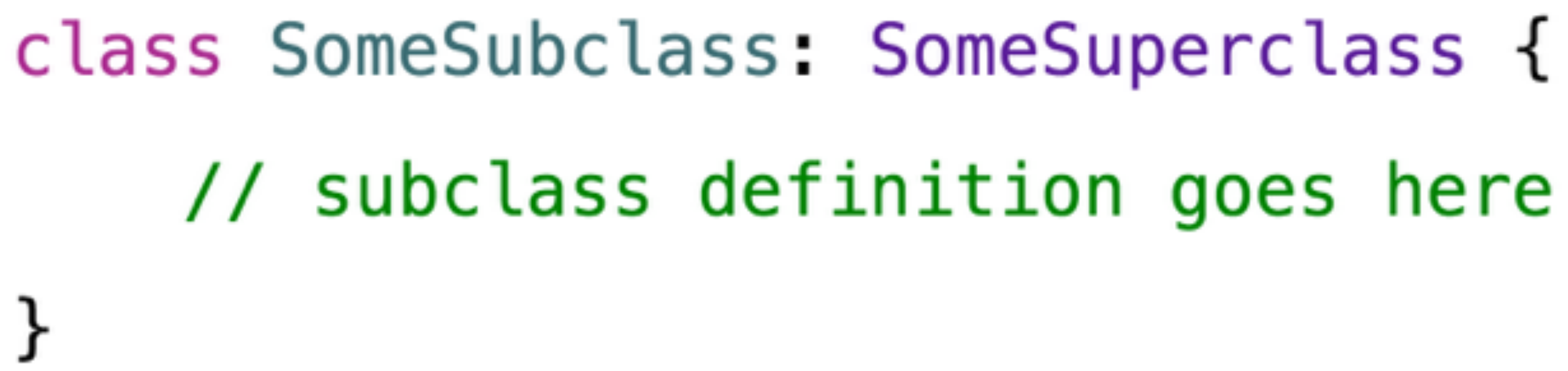
```
var shoppingList = ["Eggs", "Milk", "Cheese"]
for item in shoppingList {
    print(item)
}
```

```
["Eggs", "Milk", "Cheese"]

(3 times)
```

INHERITANCE

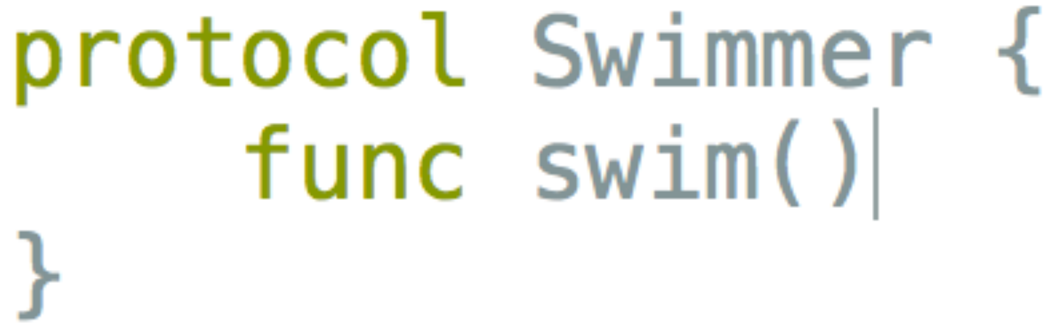
- A class can inherit methods, properties, and other characteristics from another class. When one class inherits from another, the inheriting class is known as a subclass, and the class it inherits from is known as its superclass



```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

PROTOCOLS

- Here’s an example:

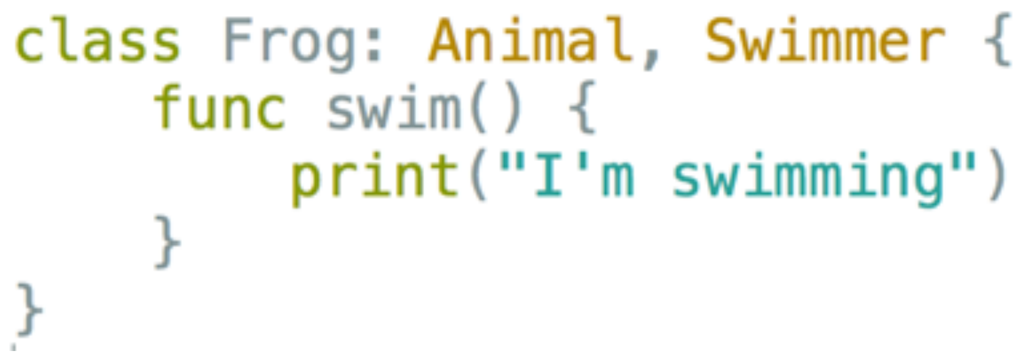


```
protocol Swimmer {
    func swim()
}
```

- Notice how we do not need to provide an implementation (any code for the function) it is just the definition

PROTOCOLS

- Here's how we adopt the protocol



```
class Frog: Animal, Swimmer {
    func swim() {
        print("I'm swimming")
    }
}
```

- When adopting the protocol we need to implement the required methods

SUMMARY

- Variable
- Constant
- Int
- Bool
- String
- Double
- UIView
- UIViewController
- Navigation Controller
- Tab Bar Controller
- Segue
- Storyboard
- IBOutlet
- IBAction
- If/Else
- <, >, <=, >=, ==
- &&, ||
- Class
- Subclass
- Superclass
- Override
- Initializer
- Protocol
- Array
- Set
- Dictionary
- For Loop

OBJECTIVES

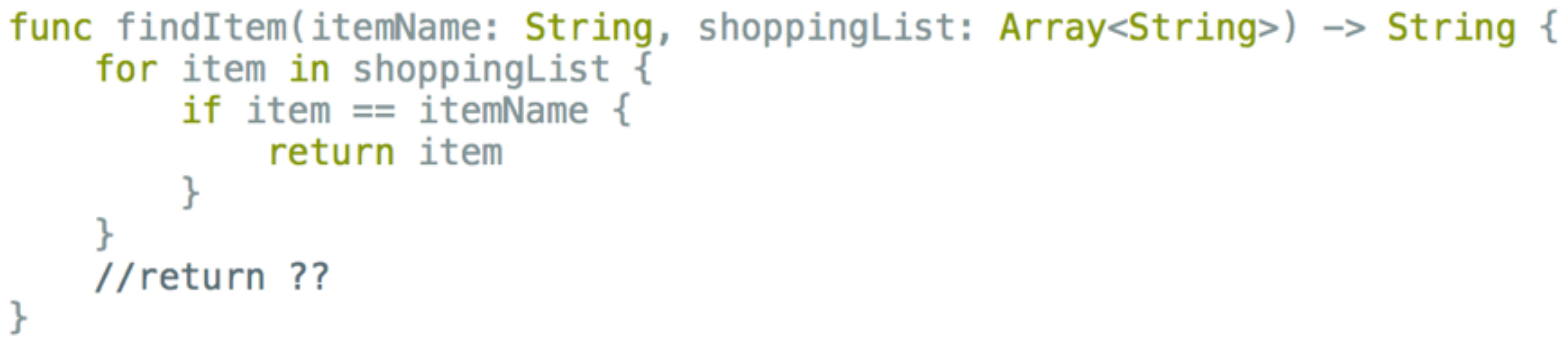
- Learn what optionals are and why we use them
- Start building with UITableView

OPTIONALS

OPTIONALS

- Is zero the same as nothing?

OPTIONALS



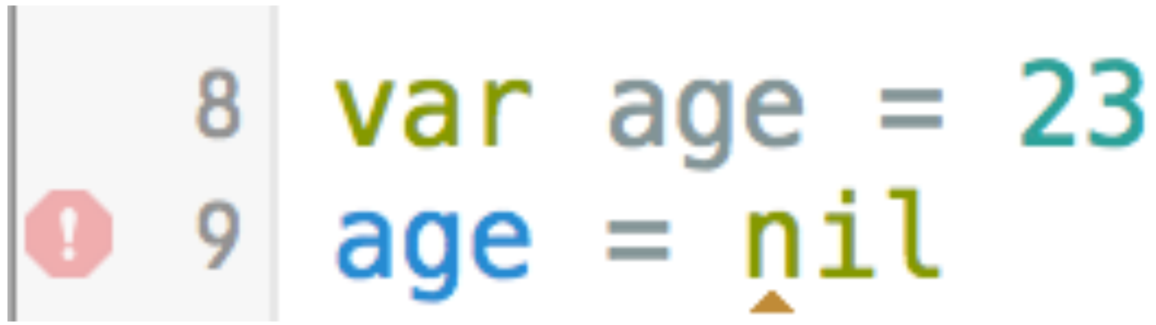
```swift
func findItem(itemName: String, shoppingList: Array<String>) -> String {
    for item in shoppingList {
        if item == itemName {
            return item
        }
    }
    //return ??
}
```

OPTIONALS

- What does optional mean in English? Not required
- An optional in Swift means that value is not required
- Using optionals, we can set a value to nothing, in other words the value is not required
- Sometimes, it's useful to have a value with no value

OPTIONALS

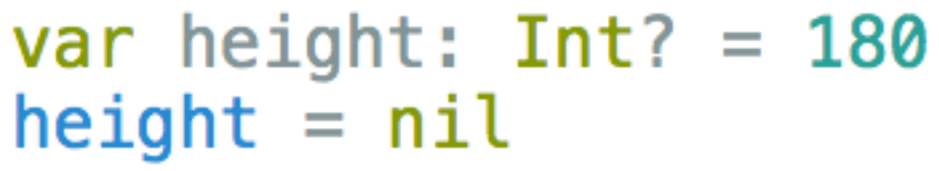
- So we actually need a name or object to represent nothing
- In Swift we use nil to represent nothing
- Before we could not assign nil to a variable



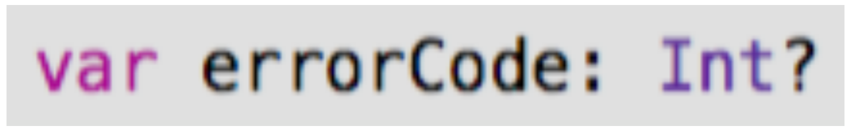
```swift
var age = 23
age = nil
```

OPTIONALS

- Instead, to declare something as optional we can add a ? to the end of the type
- Adding the ? makes it an optional type, meaning that variable can now be assigned nil
- Remember, non-optional types are guaranteed to have an actual value



```swift
var height: Int? = 180
height = nil
```



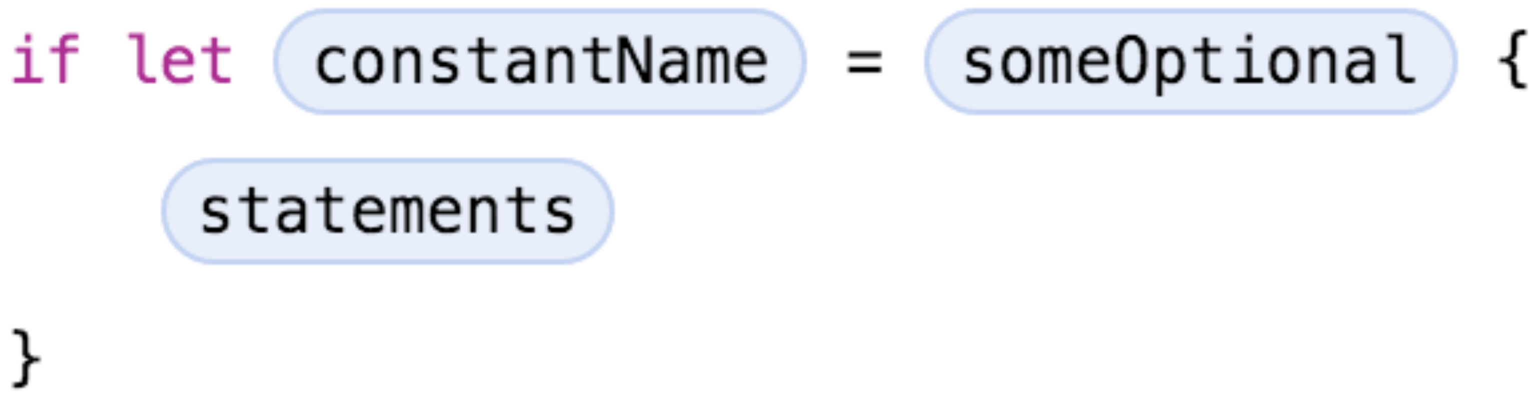
```swift
var errorCode: Int?
```

OPTIONALS

- Optionals say either “there is a value, and it equals x” or “there isn’t a value at all” (nil)
- Something can only be nil if it is an optional
- An optional is like a box:
  - It either contains a value, or it doesn't. When it doesn't contain a value, the box contains nil/nothing. The box itself does still exist, you just need to check what’s inside of the box
- So if we create an “Int?”

OPTIONAL BINDING

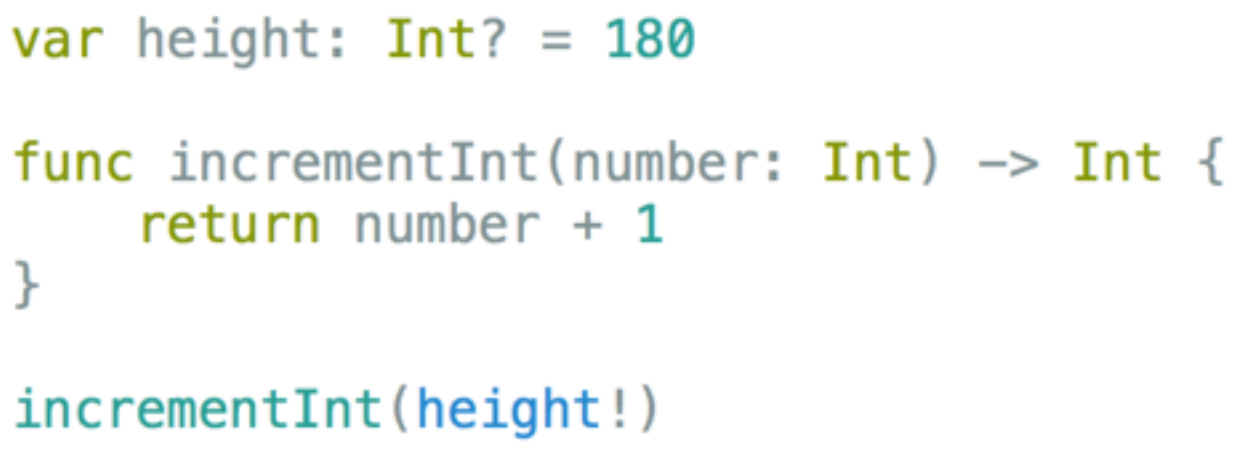
- Here we can check to see if the optional is assigned a value:



```
if let constantName = someOptional {
    statements
}
```

FORCED UNWRAPPING

- Sometimes it’s clear that our optional will ALWAYS have a valuable
- If we know that a value will always have a value, we can use an !
- We can just think of it as Swift saying, “Hey, I know you are an optional, and I know you have a value”
- This can cause errors if the value does not exist/is nil



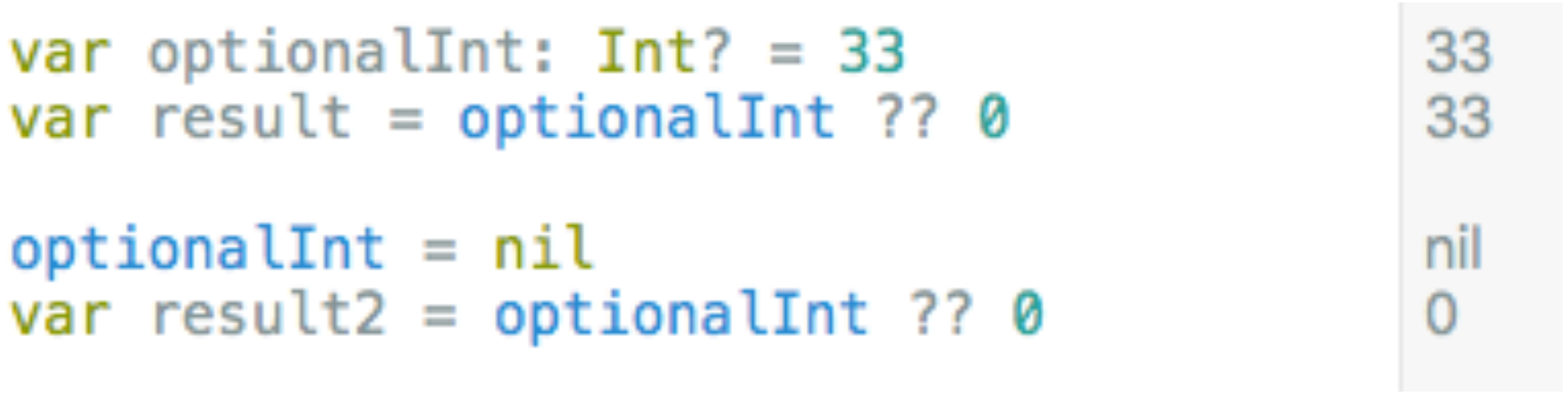
```
var height: Int? = 180

func incrementInt(number: Int) -> Int {
    return number + 1
}

incrementInt(height!)
```

NIL COALESCING

- We can use a ?? to check to see if a value is nil
- If it is nil, we can assign it another value
- We can sort of think of this as a default value



```
var optionalInt: Int? = 33            33
var result = optionalInt ?? 0         33

optionalInt = nil                     nil
var result2 = optionalInt ?? 0        0
```

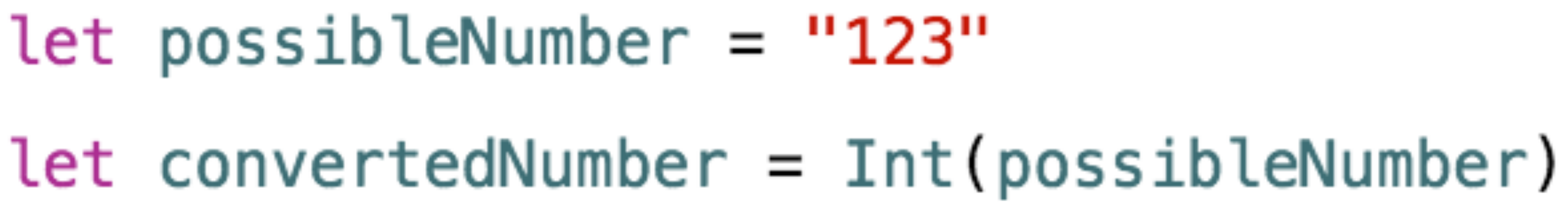
OPTIONAL CHAINING

- If we use ! and the value doesn’t exist our app can crash, that’s why I call it a force unwrap
- When accessing properties we can use a ? instead of a !

```
myObject.employer?.companySize
```

OPTIONALS

- Why would we want to assign an object to nil?
- Sometimes things fail or sometimes things don’t have values because it doesn’t make sense
- What if had a division function? What if we divided by zero? Nil can be a good way to prevent that error



```swift
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
```

OPTIONALS SUMMARY

- Nil represents the absence of a value
- In order to be assigned nil, they need to be of the Optional type. Non-optional variables and constants must have a non-nil value.
- To make something an Optional type we add a ? to the end of the type name
- Optional variables and constants are like boxes that can contain a value or be empty (nil)
- To use the value inside an optional, you must unwrap it from the optional
- If-let binding and nil coalescing are safer to unwrap optionals
- Forced unwrapping can produce a runtime error, so avoid when possible

UITABLEVIEW

TABLE VIEWS

- Table views are a one dimensional list (list view may be a better name)
  - Vocabulary:
    - Section: All table views contain 1 or more sections; these are logical divisions of data
    - Row: Every section has a number of rows, which are entries in that section, each row has a UITableViewCell
    - Index path: The combination of a section and row that is a unique position in a table view
    - Cell: The view that is displayed for an index path (the class UITableViewCell is a subclass of UIView)
- Table views must have a number of sections, a number of cells in each section, and (optionally), the cells themselves
- Table views have a data source and a delegate (these are protocols)
  - Data source: Provides cells, number of cells and sections
  - Delegate: Gets called when things happen to the table view, provides some views (e.g. header and footer)

DATA SOURCE

- We are going to make our view controller adopt the UITableViewDataSource protocol
- That means our UIViewController subclass will NEED to implement a couple of methods (and can implement a few others optionally)
- Two required methods
  - tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
  - tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int
- Cell for row at indexPath: returns a UITableViewCell
- Number of rows in section returns an Int telling the table view how many rows it will have

CODE ALONG

CHALLENGE

LAB

- Challenge 1: Add an array of strings to the view controller and display each item in the array in a row
- Challenge 2: Create a new Swift file called Todo. In the file create a class called Todo with a String property called item and a Bool isDone property.
- Challenge 3: Replace the array of Strings with an array of Todos and display each todo item
- Challenge 4: Add a checkmark to each Todo item (hint: look at cell.accessoryType)
- Challenge 5: implement the didSelectRowAtIndexPath method to toggle the checkmark