CS5530
Mobile/Wireless Systems
Swift

Yanyan Zhuang

Department of Computer Science

http://www.cs.uccs.edu/~yzhuang

cat announce.txt_

- iMacs remote VNC access
  - VNP: http://www.uccs.edu/itservices/services/network-and-internet/vpn.html
  - VNC password: cs5530
  - Please save data to Z
  - Please do not use iMacs in Library
  - IT will upgrade…

Swift

- What is it?
  - A new programming language for Apple products
    - iOS (ipods, iphones, ipads, etc.), macOS, watchOS, tvOS, future…
    - Currently at version 3
      - □ To see your version: xcrun swift -version
      - □ Apple Swift version 3.0.2 (swiftlang-800.0.63 clang-800.0.42.1)
    - Open source
  - Based on Objective-C and C.
    - Classes, instances, properties, methods, inheritance, etc.

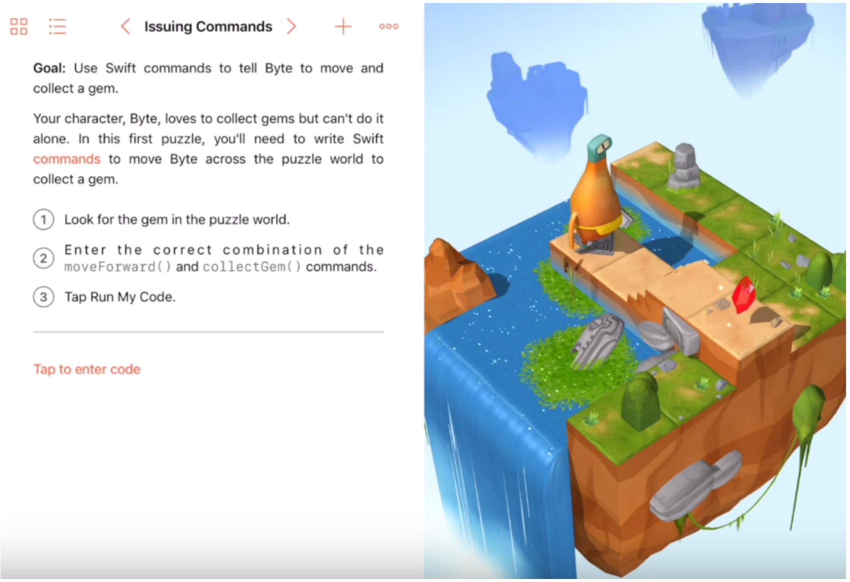
Swift

- Requires an Apple product for development
  - Air, MacBook, MacBook Pro, iMac, iTrashCan (MacPro)

- Requires the ‘Xcode’ development environment, Apple only.
- Resources at:
  - https://developer.apple.com/

Xcode Playground

- An interactive work environment that allows you update values real-time and see results.
- A ‘project’ option in Xcode.
- New for iPad iOS 10!!!



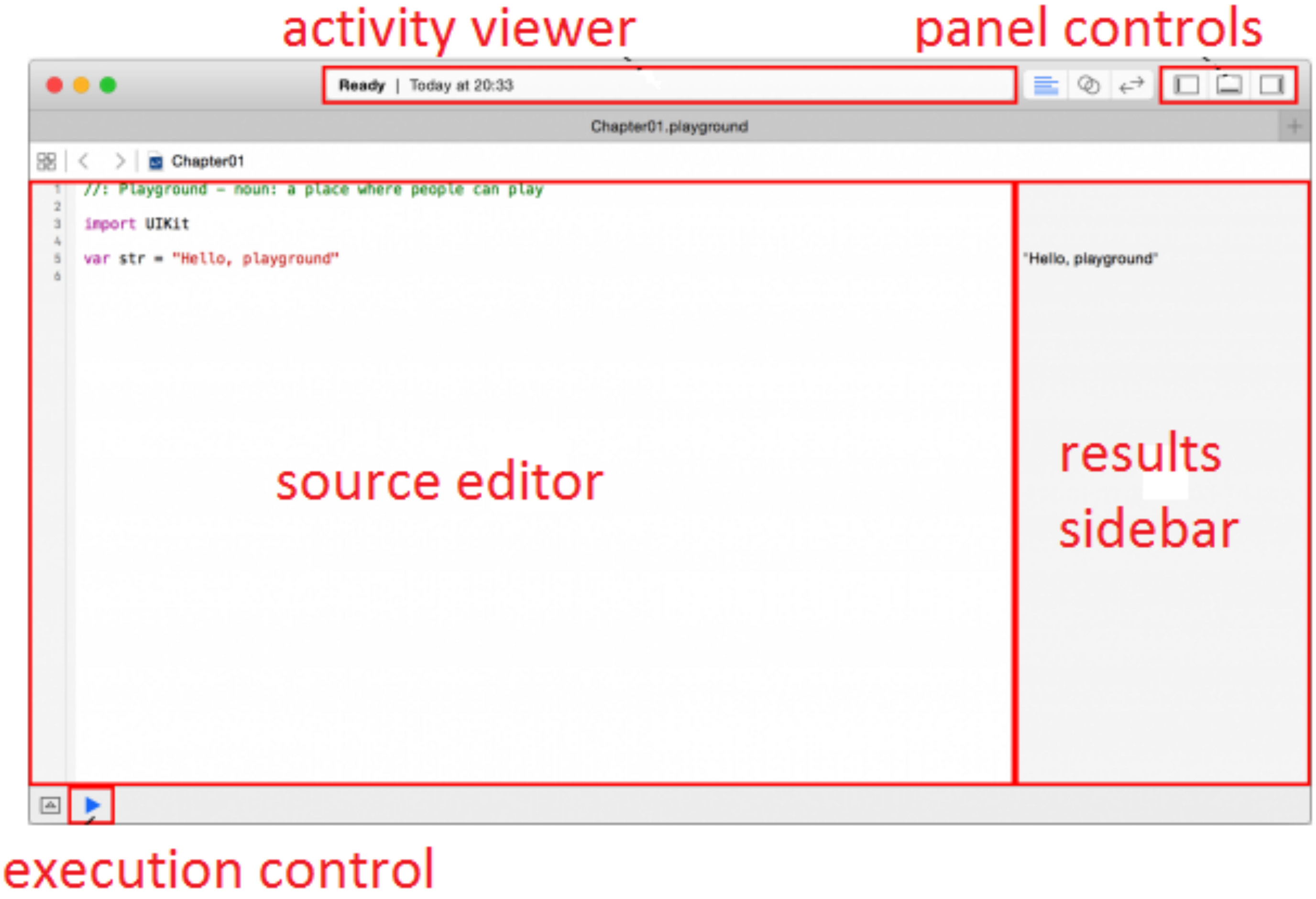
Goal: Use Swift commands to tell Byte to move and collect a gem.

Your character, Byte, loves to collect gems but can't do it alone. In this first puzzle, you'll need to write Swift commands to move Byte across the puzzle world to collect a gem.

1. Look for the gem in the puzzle world.
2. Enter the correct combination of the `moveForward()` and `collectGem()` commands.
3. Tap Run My Code.

Tap to enter code

Xcode Playground



```
//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
```

"Hello, playground"

activity viewer

panel controls

source editor

results sidebar

execution control

Swift

- Quick overview of the language
  - Assignments
  - Control Flow
  - Functions and Closures
  - Objects and Classes
  - Enumerations and Structures
  - Protocols
  - Error Handling

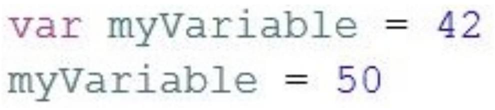
Swift - Overview

- “Don’t need to import a separate library for functionality like input/output or string handling.
- Code written at global scope is used as the entry point for the program, so you don’t need main().
- Don’t need to write semicolons at the end of every statement.”
  - Excerpt From: Apple Inc. “The Swift Programming Language (Swift 3.0.1).” iBooks. https://itun.es/ca/jEUH0.l

Swift - Assignments

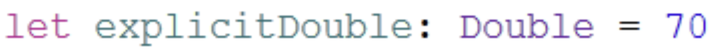
Key word	Description
`let`	Used for **constants**. Does not need to be known at compile time but must be assigned a value exactly once.
`var`	Used for **variables**.

- Types can be ‘inferred’



```
var myVariable = 42
myVariable = 50
```

- Can be explicit



```
let explicitDouble: Double = 70
```

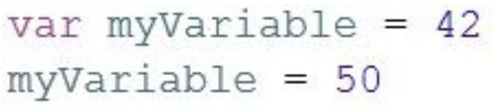
- NO implicit type conversions
  - Values in strings by using a “\”

```
let apples = 3
let applySummary = “I have \(apples) apples.”
```

Swift - Assignments

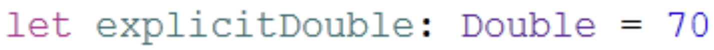
Key word	Description
`let`	Used for **constants**. Does not need to be known at compile time but must be assigned a value exactly once.
`var`	Used for **variables**.

- Types can be 'inferred'



```
var myVariable = 42
myVariable = 50
```

- Can be explicit



```
let explicitDouble: Double = 70
```

- NO implicit type conversions
  - Values in strings by using a "\"

    ```
    let apples = 3
    let applySummary = "I have \(apples) apples."
    ```

  - Values are never implicitly converted to another type. If need to convert a value to a different type, explicitly make an instance of the desired type.

"The Swift Programming Language (Swift 3.0.1)."

Swift - Assignments

- Assignments
  - Dictionaries and arrays use []

```
var shoppingList = [“hp”, “apple”, “microsoft”]
shoppingList[1] = “Lenovo”
var occupations = ["Malcolm": "Captain", "Kaylee": "Mechanic"]
occupations["Jayne"] = "Public Relations”
```

  - Empty arrays or dictionaries

```
let emptyArray = [String]()
Let emptyDictionary = [String: Float]()
```

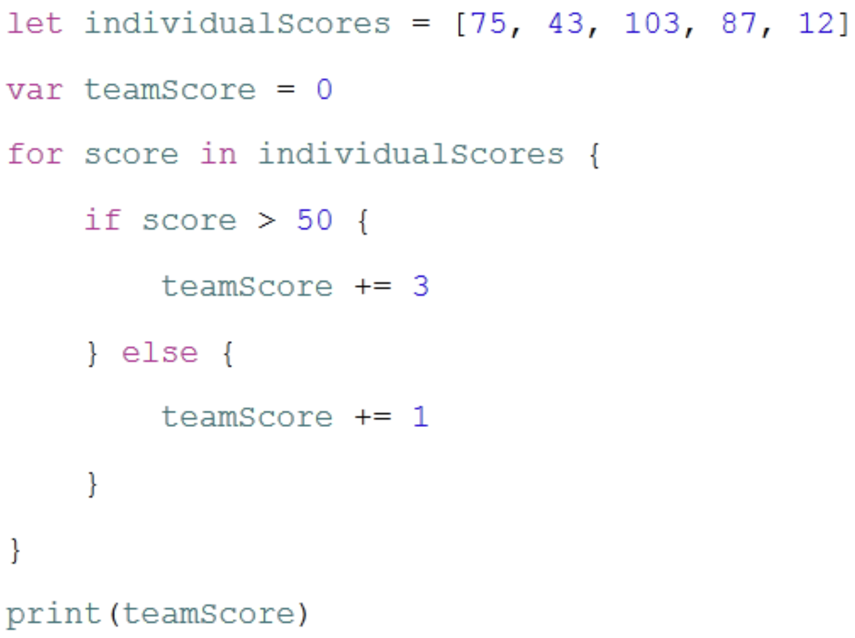
If type information can be inferred, can write an empty array as [] and an empty dictionary as [:]

- Data Types
  - Typical data types available.
    - String, Float, Double, Bool, Int/Uint, Character, Optional

Swift – Control Flow

Keyword	Description
`if, switch`	Used for **conditionals**. Parenthesis around variable are optional. Braces around conditional body are required.
`for-in, for, while, repeat-while`	Used for **loops**. Parenthesis around variable are optional. Braces around loop body are required.

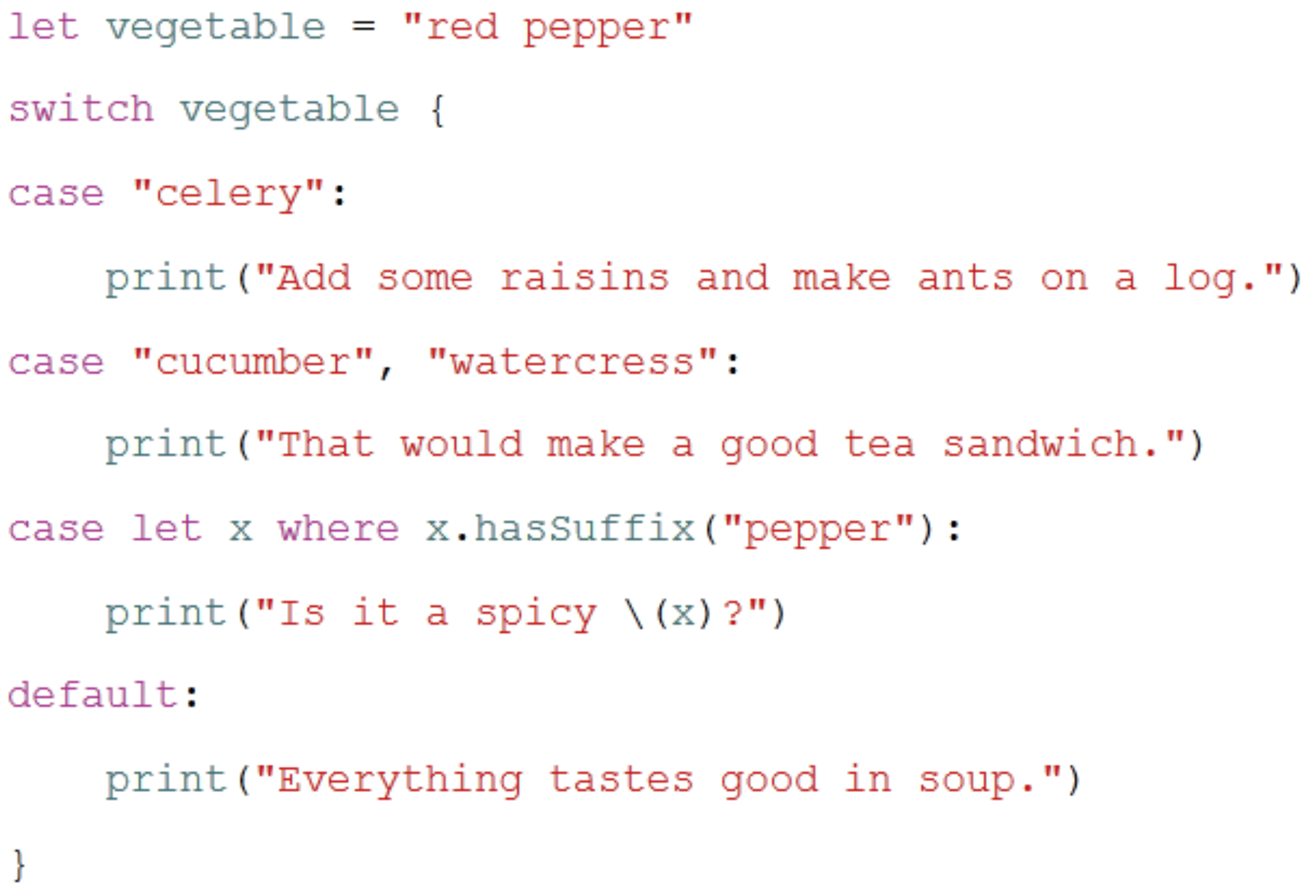
- For/if example
  - If condition must be explicit
  - if score {..} is an error



```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
```

Swift – Control Flow

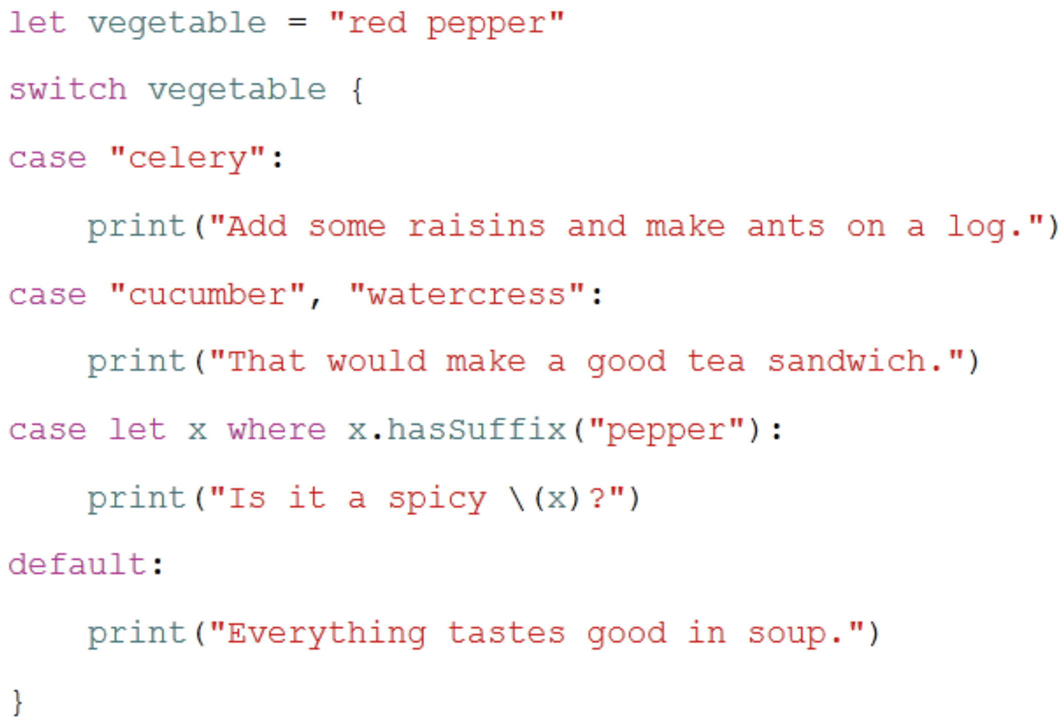
- Switch



```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    print("Add some raisins and make ants on a log.")
case "cucumber", "watercress":
    print("That would make a good tea sandwich.")
case let x where x.hasSuffix("pepper"):
    print("Is it a spicy \(x)?")
default:
    print("Everything tastes good in soup.")
}
```

Swift – Control Flow

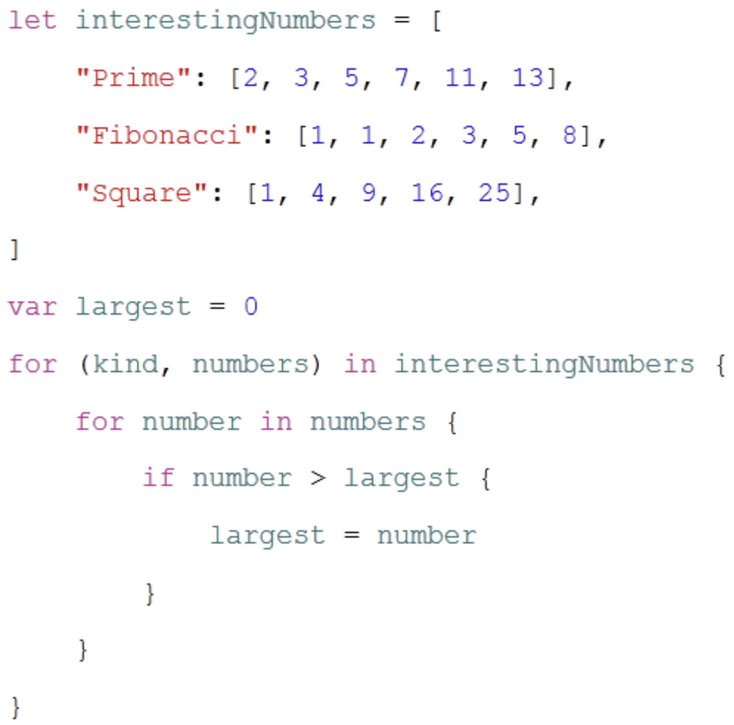
- Switch
  - let can be used in a pattern to assign value
  - No need to break
    - Only one match



```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    print("Add some raisins and make ants on a log.")
case "cucumber", "watercress":
    print("That would make a good tea sandwich.")
case let x where x.hasSuffix("pepper"):
    print("Is it a spicy \(x)?")
default:
    print("Everything tastes good in soup.")
}
```

Swift – Control Flow

- for-in
  - Iterate over items in a dictionary by providing a pair of names to use for each key-value pair.
  - Dictionaries are unordered!



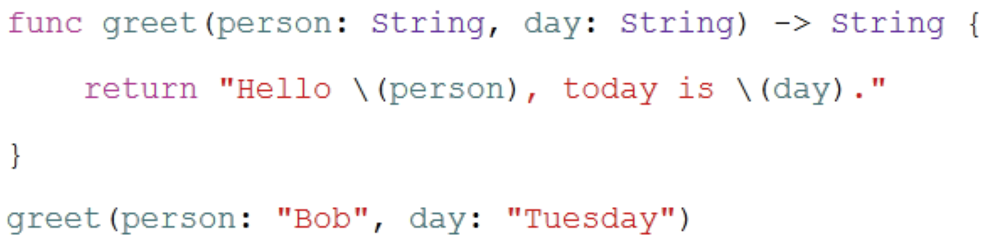
```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
```

Swift – Control Flow

- While & repeat-while
  - Same as C or Java's while & do-while.
  - repeat { … } while *some-condition*
- For loops still the same
  - Though you can use `..<` or `...` to make ranges.
    - □ `0..<7` non-inclusive upper bound.
      - □ `for i in 0..<7 { … }`
    - □ `0...7` inclusive upper bound
      - □ `for i in 0...7 { … }`

Swift – Functions & Closures

- Use `func` to declare a function
  - → to indicate return type

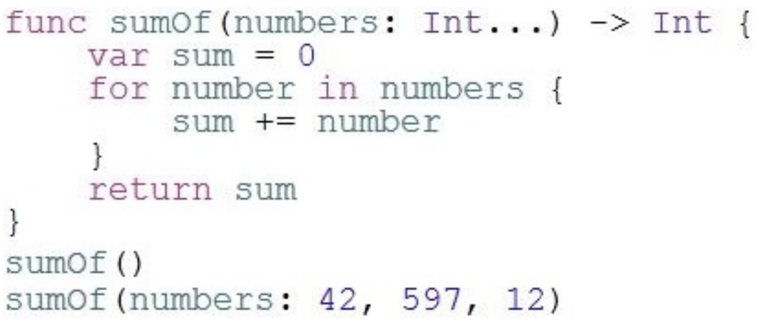


```
func greet(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet(person: "Bob", day: "Tuesday")
```

  - Use a tuple to make a compound value: return multiple values from a function
    - Elements of a tuple can be referred to by name or by number
    - Defined as …… → `(min: Int, max: Int, sum: Int)`
    - Access as `results.sum, or results.2`

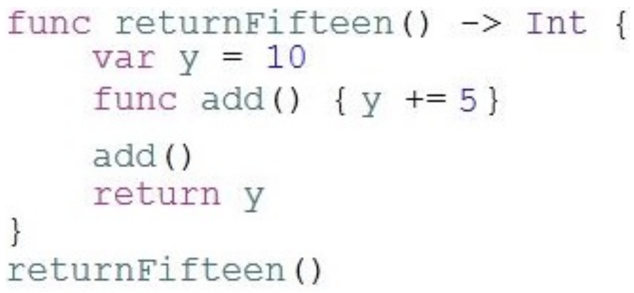
Swift – Functions & Closures

- Can take variable arguments, collects into an array for you.



```
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}
sumOf()
sumOf(numbers: 42, 597, 12)
```

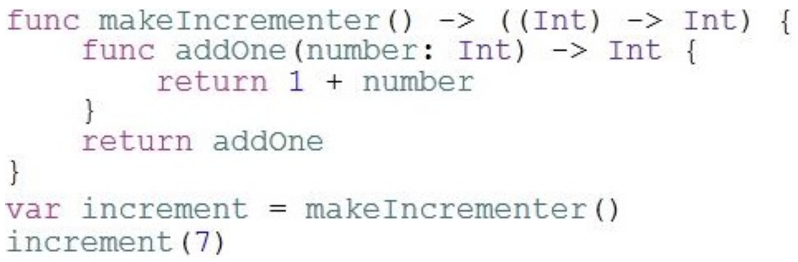
- Can be nested.



```
func returnFifteen() -> Int {
    var y = 10
    func add() { y += 5}

    add()
    return y
}
returnFifteen()
```

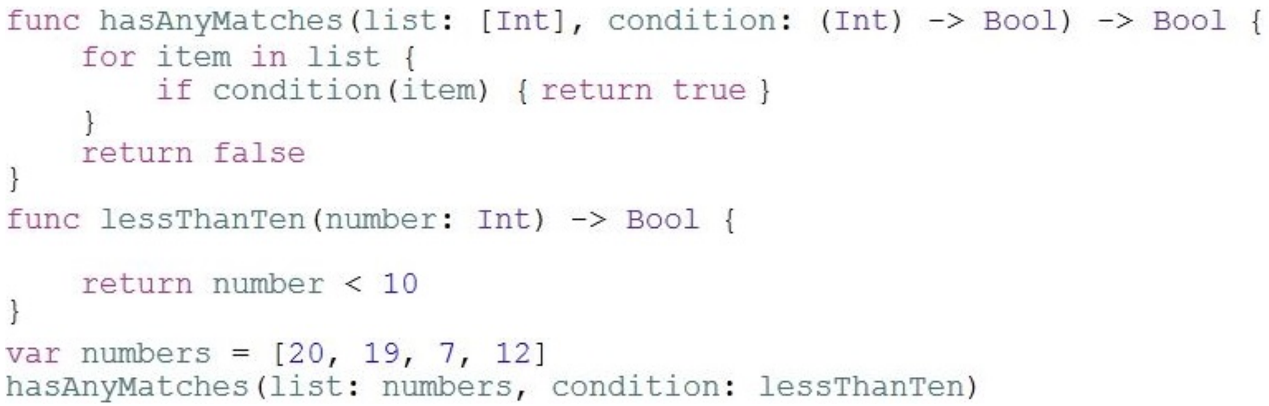
Swift – Functions & Closures

- Functions are first-class types: they can return another function as a return-value



```
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

- Can take another function as one of its arguments



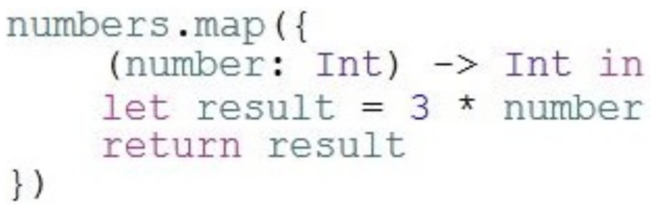
```
func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
    for item in list {
        if condition(item) { return true }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {

    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(list: numbers, condition: lessThanTen)
```

Swift – Functions & Closures

- A closure is a block of code that can be called later (anonymous function)
- Code in a closure has access to
  - Variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is executed
  - You can write a closure without a name (function name)
    - Surround code with braces {}
    - Use ‘in’ to separate the arguments and return type from the body
      - Indicates that definition of closure’s parameters and return type has finished, and the body of the closure is about to begin

Syntax:

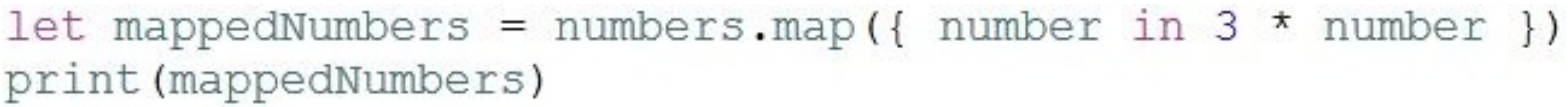
```
{ (parameters) -> return type in
      statements
 }
```



```
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

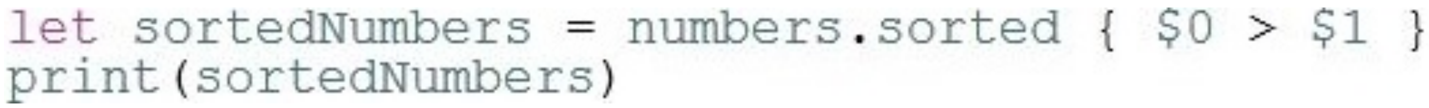
Swift – Functions & Closures

- Concise 1: if type already known, you can omit types of parameters and/or return type.



```
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
```

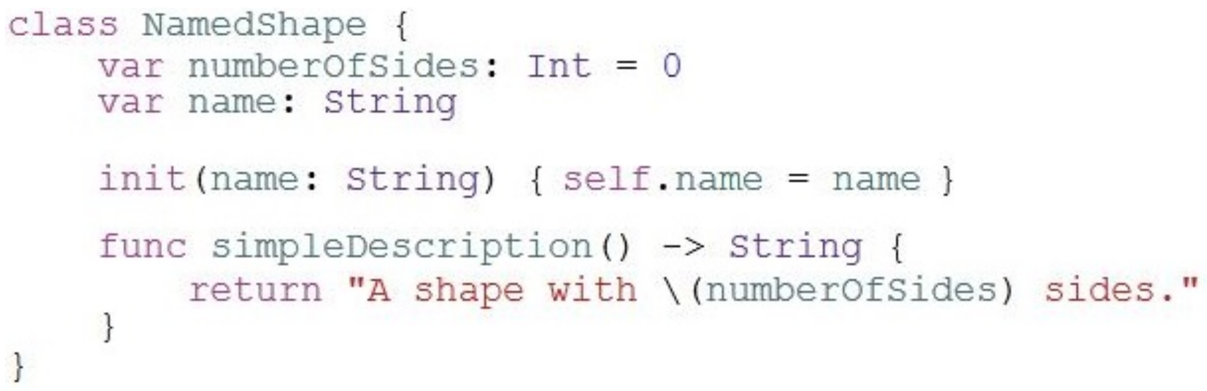
- Concise 2: can refer to parameters by number instead of name



```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
```

Swift – Objects & Classes

- Classes
  - As we'd expect.
  - Use '`init`' as initializer / constructor.



```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) { self.name = name }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

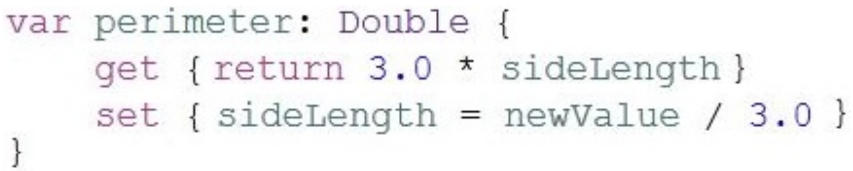
  - Use '`deinit`' as deinitializer / destructor
  - Instantiation by referencing class name followed by ()
    - var shape = Shape()

Swift – Objects & Classes

- Classes
  - To inherit, subclasses include their super classes name after their class name, separated by a :
    - class Square: Shape
    - class ViewController: UIViewController, UITextFieldDelegate
  - Methods in a subclass that override the superclass's implementation are marked with override
    - Overriding a method by accident, without override, is detected by the compiler as an error

Swift – Objects & Classes

- **Properties** can have ‘getter’ and ‘setter’ methods.
  - Similar to Java, C#, VB.Net
  - Note ‘`newValue`’ is implicitly defined for us as the new value (see code example)

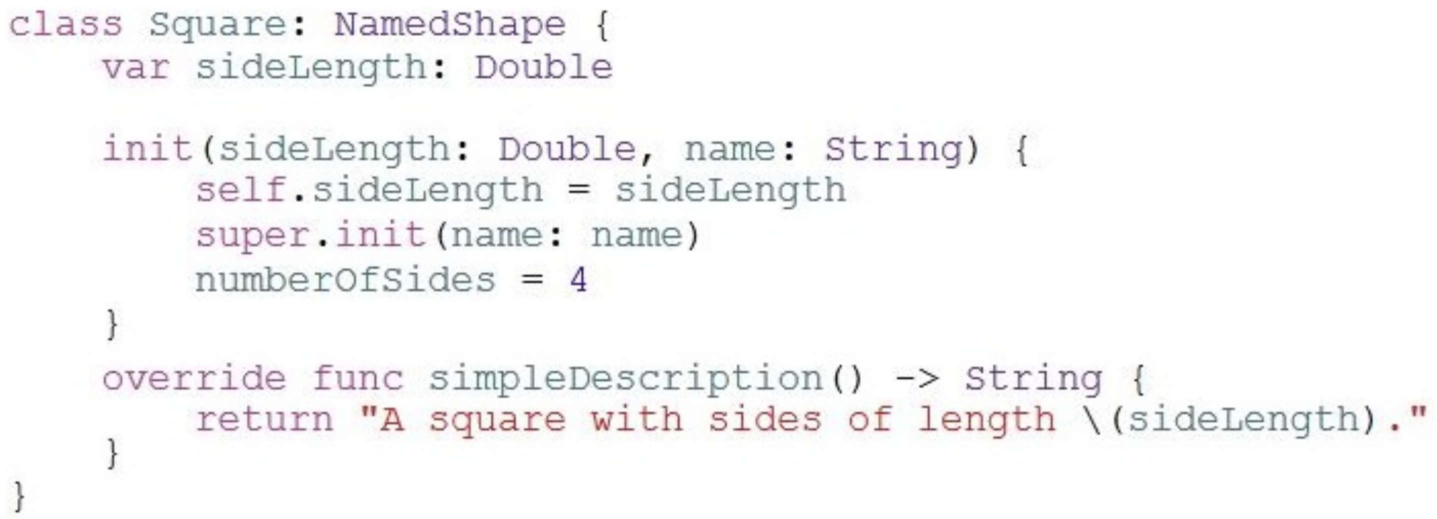


```
var perimeter: Double {
    get {return 3.0 * sideLength}
    set {sideLength = newValue / 3.0 }
}
```

  - Can be explicit by declaring the setter as:
    - □ `set(<parameter_name>)`
    - □ `set( mySide ) { ... }`
    - □ There is no type declaration needed because the property defined it.

Swift – Objects & Classes

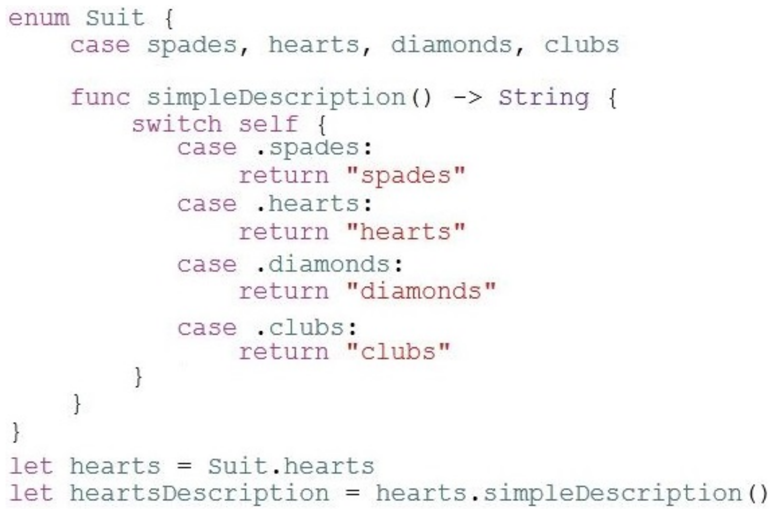
- Inheritance
  - Class: parent
  - Over ride with ‘`override`’ keyword.
  - Call parent methods with ‘`super.`’ keyword.



```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }
    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
```

Swift – Enumerations & Structures

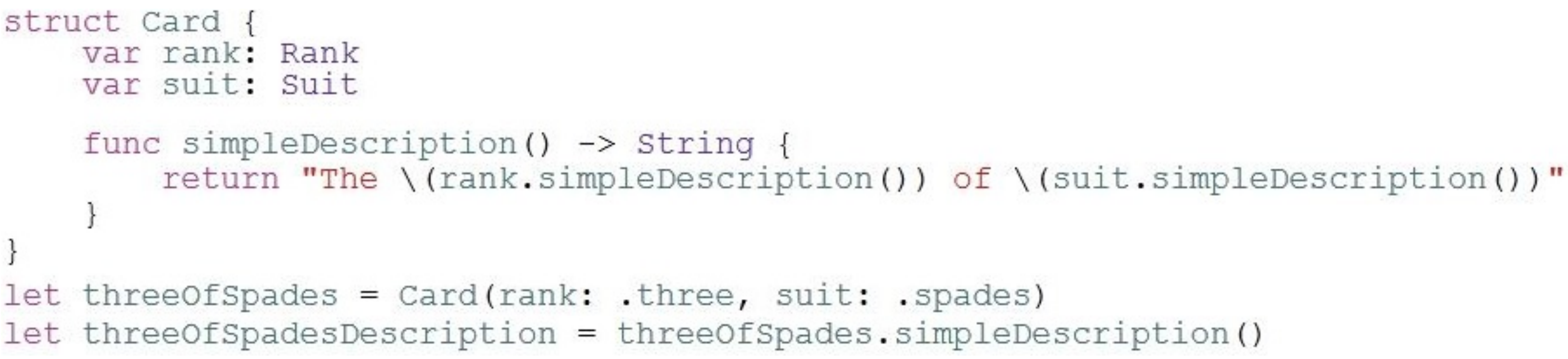
- Enumerations
  - Use ‘`enum`’ to create an enumeration
    - Swift assigns raw values starting at zero and increments by 1, but can change this by explicitly specifying values
  - Can have methods associated with them.



```
enum Suit {
    case spades, hearts, diamonds, clubs

    func simpleDescription() -> String {
        switch self {
           case .spades:
               return "spades"
           case .hearts:
               return "hearts"
           case .diamonds:
               return "diamonds"
           case .clubs:
               return "clubs"
        }
    }
}
let hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()
```

Swift – Enumerations & Structures

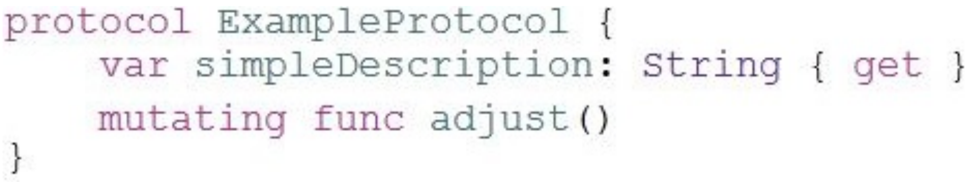
- Structures
  - Use ‘`struct`’ to create a structure.
  - Support many of the same behaviors as classes, including methods & initializers.
  - Structures are passed by value! (classes by reference)



```
struct Card {
    var rank: Rank
    var suit: Suit

    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .three, suit: .spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

Swift – Protocols & Extensions

- Protocols
  - It’s basically an ‘interface’ from other OO languages.
  - Use ‘`protocol`’ to declare a protocol.

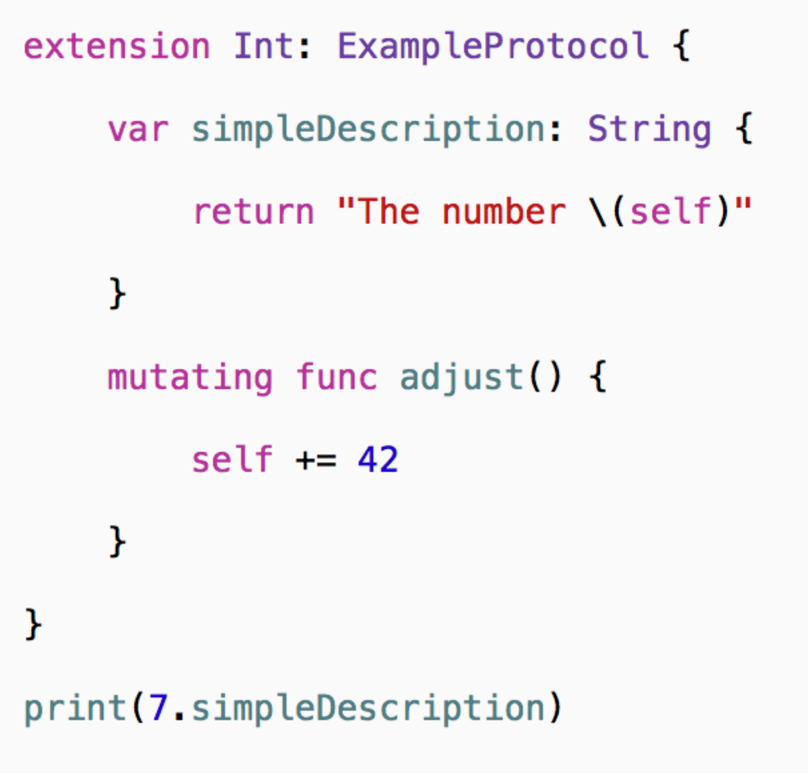


```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

  - ‘`mutating`’ indicates a function changing the struct.
    - Not needed in class redefinitions as class methods can always modify the class.
    - Needed in structures to indicate that the method will modify the structure.
  - Classes, enumerations and structs can all adopt protocols.

Swift – Protocols & Extensions

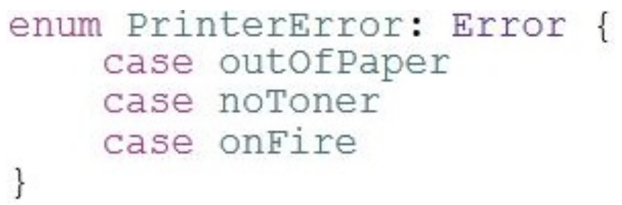
- Use extensions to add functionality to an existing type



```
extension Int: ExampleProtocol {

    var simpleDescription: String {

        return "The number \(self)"

    }

    mutating func adjust() {

        self += 42

    }

}

print(7.simpleDescription)
```

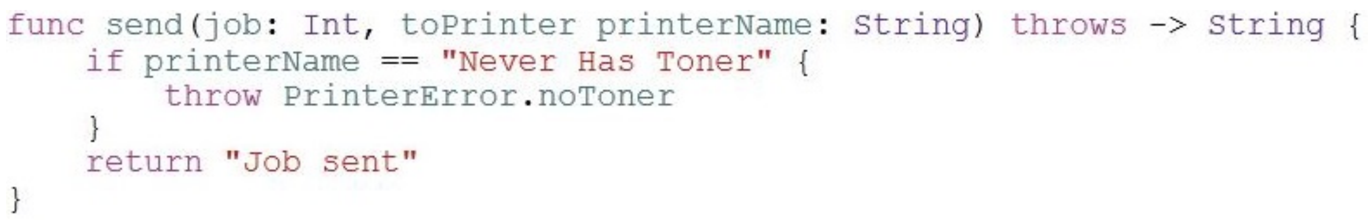
Swift – Error Handling

- Error Handling
  - Represent errors using any type that adopts the `Error` protocol.



```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

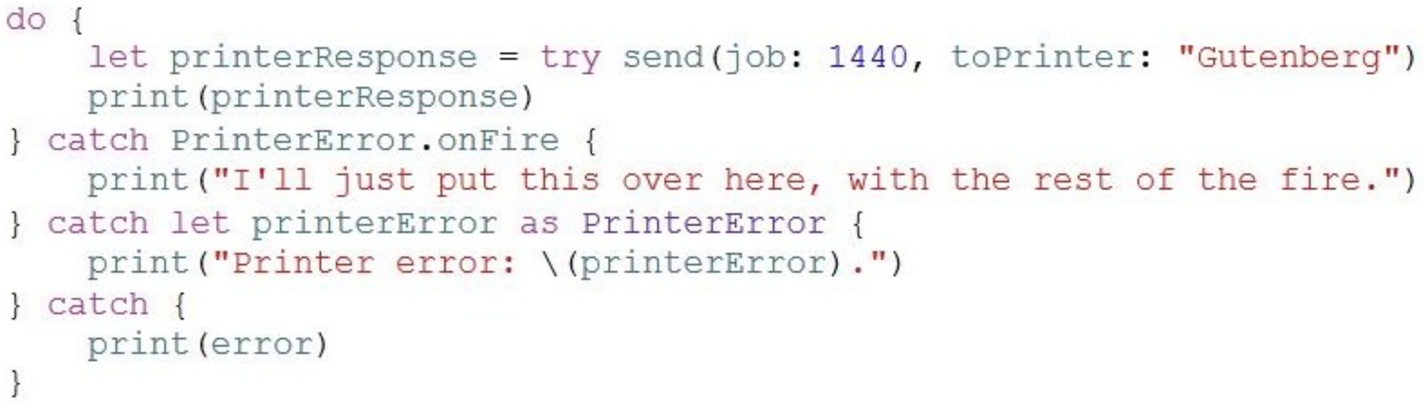
  - Use ‘`throw`’ to throw an error and ‘`throws`’ to denote a function that can throw an error.



```
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

Swift – Error Handling

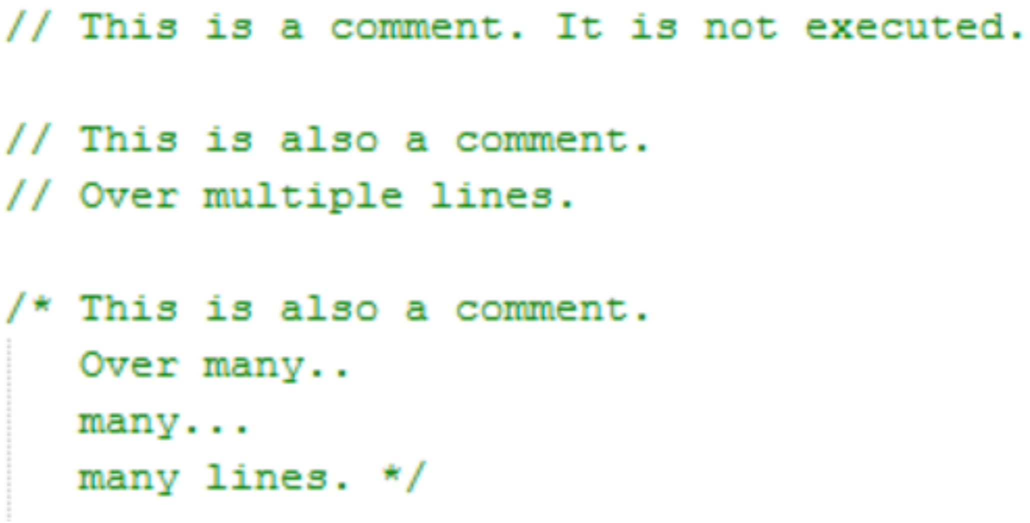
- Error Handling
  - do / catch



```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
```

    - In do block, mark code that can throw an error by writing **try** in front
    - In catch block, the error is automatically given the name error unless you give it a different name
    - Can provide multiple catch blocks that handle specific errors

Swift – Comments



```
// This is a comment. It is not executed.

// This is also a comment.
// Over multiple lines.

/* This is also a comment.
   Over many..
   many...
   many lines. */
```

Let’s Practice!

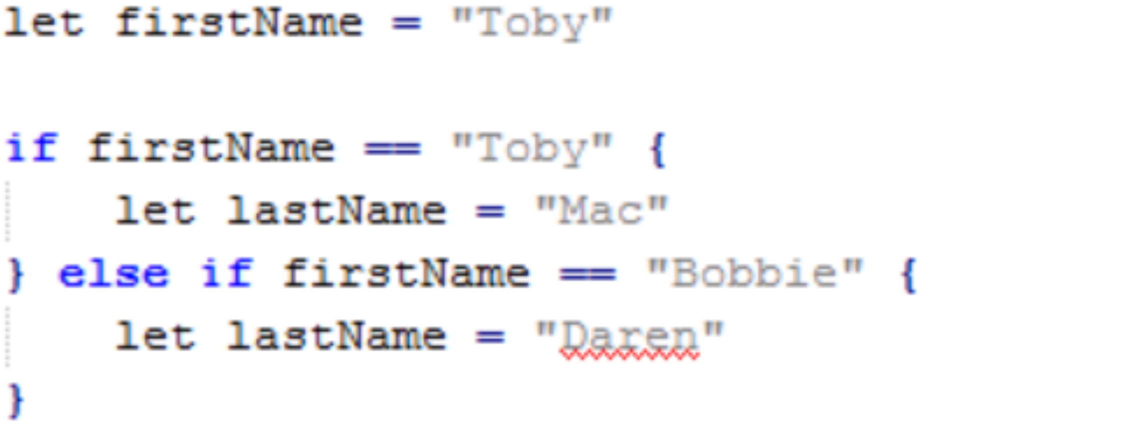
- Print strings (use terminator:”” to disable \n)
  - let label = "The width is "
  - let width = 94
  - print(label+String(width))
  - // compare with print(label+String(width), terminator:””)

  - let apples = 3
  - let appleSummary = "I have \(apples) apples."

  - let oranges = 5
  - let fruitSummary = "I have \(apples+oranges) pieces of fruit."

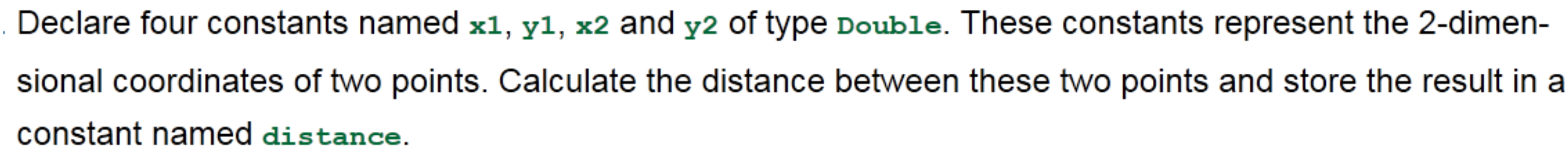
Let’s Practice!

- Q1: What’s wrong with the following code?



```
let firstName = "Toby"

if firstName == "Toby" {
    let lastName = "Mac"
} else if firstName == "Bobbie" {
    let lastName = "Daren"
}
let fullName = firstName + " " + lastName
```

- Q2:



Declare four constants named x1, y1, x2 and y2 of type Double. These constants represent the 2-dimensional coordinates of two points. Calculate the distance between these two points and store the result in a constant named distance.

Swift Resources

- Content was used from these web sites where appropriate. These sites contain quite a bit more information and would make a great resource for you.

Developer

https://developer.apple.com/

https://www.hackingwithswift.com/read

https://www.hackingwithswift.com/example-code

https://itunes.apple.com/us/book/the-swift-programming-language/id881256329?mt=11