Swift

What is Swift?

- Swift is a powerful and intuitive programming language for macOS, iOS, watchOS and tvOS.

- Swift is the result of the latest research on programming languages, combined with decades of experience building Apple platforms.

So Why Swift?

- Objective C is complicated and confusing, especially for new programmers. It is viewed as clunkier than some modern programming languages. Swift is simpler and easier than Objective-C with respect to syntax and code writing.

- If developers are happy with easy and simple coding style, new programmers will automatically prefer iOS and Mac development.

Why Swift is not Objective-C ?

- C is classic high-level programming language, and Objective-C is one of its flavors that has traditionally been Apple’s language of choice.
- Swift can replace entire lines of code with fewer characters using naming conventions brought directly from Objective-C.
- Swift is expressed with clean syntax that makes API’s easy to read and maintain. It offers the same power and flexibility as lower-level languages.

Let’s jump into some swift basics

Let and var

- You declare constants with the ‘let’ keyword and variables with the ‘var’ keyword.

```
let maximumNumberOfLoginAttempts = 10

var currentLoginAttempt = 0
```

Classes vs structures

Classes and structures in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

Classes vs structures

Classes have additional capabilities that structures do not:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

Syntax

- Classes and structures have a similar definition syntax. You introduce classes with the class keyword and structures with the struct keyword.

```
1  class SomeClass {
2      // class definition goes here
3  }
4  struct SomeStructure {
5      // structure definition goes here
6  }
```

Declaration and initialization

Here's an example of a structure definition and a class definition:

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

The syntax for creating instances is very similar for both structures and classes:

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Accessing properties

You can access the properties of an instance using *dot syntax*. In dot syntax, you write the property name immediately after the instance name, separated by a period (`.`), without any spaces:

```
print("The width of someResolution is \(someResolution.width)")
// Prints "The width of someResolution is 0"
```

In this example, `someResolution.width` refers to the `width` property of `someResolution`, and returns its default initial value of `0`.

You can drill down into sub-properties, such as the `width` property in the `resolution` property of a `VideoMode`:

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is 0"
```

Functions

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
```

Functions Without Parameters

Functions are not required to define input parameters. Here's a function with no input parameters, which always returns the same `String` message whenever it is called:

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Prints "hello, world"
```

Functions

Functions With Multiple Parameters

Functions can have multiple input parameters, which are written within the function's parentheses, separated by commas.

This function takes a person's name and whether they have already been greeted as input, and returns an appropriate greeting for that person:

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

Multiple return values

Functions with Multiple Return Values

You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.

The example below defines a function called `minMax(array:)`, which finds the smallest and largest numbers in an array of `Int` values:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Strings and Characters

String Literals

You can include predefined `String` values within your code as *string literals*. A string literal is a sequence of characters surrounded by double quotation marks (").

Use a string literal as an initial value for a constant or variable:

```
let someString = "Some string literal value"
```

Note that Swift infers a type of `String` for the `someString` constant because it's initialized with a string literal value.

Strings and Characters

Multiline String Literals

If you need a string that spans several lines, use a multiline string literal—a sequence of characters surrounded by three double quotation marks:

```
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
```

Strings and Characters

Special Characters in String Literals

String literals can include the following special characters:

- The escaped special characters `\0` (null character), `\\` (backslash), `\t` (horizontal tab), `\n` (line feed), `\r` (carriage return), `\"` (double quotation mark) and `\'` (single quotation mark)
- An arbitrary Unicode scalar, written as `\u{`*n*`}`, where *n* is a 1–8 digit hexadecimal number with a value equal to a valid Unicode code point (Unicode is discussed in Unicode below)

The code below shows four examples of these special characters. The `wiseWords` constant contains two escaped double quotation marks. The `dollarSign`, `blackHeart`, and `sparklingHeart` constants demonstrate the Unicode scalar format:

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
let dollarSign = "\u{24}"        // $,  Unicode scalar U+0024
let blackHeart = "\u{2665}"      // ♥,  Unicode scalar U+2665
let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
```

Strings and characters

String Mutability

You indicate whether a particular `String` can be modified (or *mutated*) by assigning it to a variable (in which case it can be modified), or to a constant (in which case it can't be modified):

```
1  var variableString = "Horse"
2  variableString += " and carriage"
3  // variableString is now "Horse and carriage"
4
5  let constantString = "Highlander"
6  constantString += " and another Highlander"
7  // this reports a compile-time error - a constant string cannot be modified
```

Protocols

You define protocols in a very similar way to classes, structures

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Custom types state that they adopt a particular protocol by placing the protocol's name after the type's name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}
```

If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

OOP VS POP

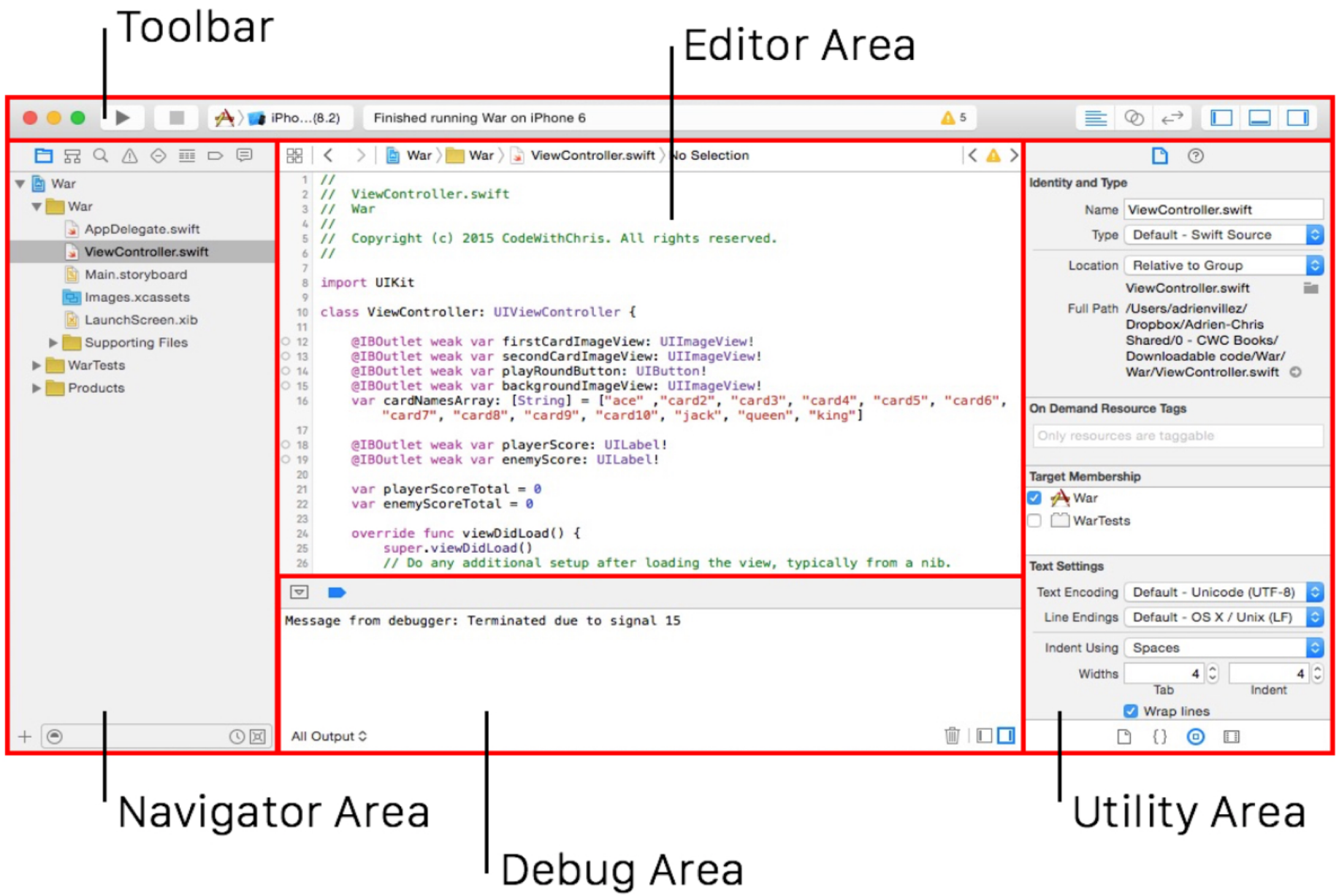
- The primary aspect of POP over OOP is that is prefers composition over inheritance. There are several benefits to this.
- In large inheritance hierarchies, the ancestor classes *tend* to contain most of the (generalized) functionality, with the leaf subclasses making only minimal contributions. The issue here is that the ancestor classes end up doing a lot of things.
- Swift offers multiple features to achieve this, but the most important by far are protocol extensions. They allow implementation of a protocol to exist separate of its implementing class, so that many classes may simply implement this protocol and instantly gain its functionality.

Let’s get familiar with Xcode

1. Where Can I Download Xcode?

The easiest way to get Xcode is through the Mac App Store. Click the link to go to the listing.

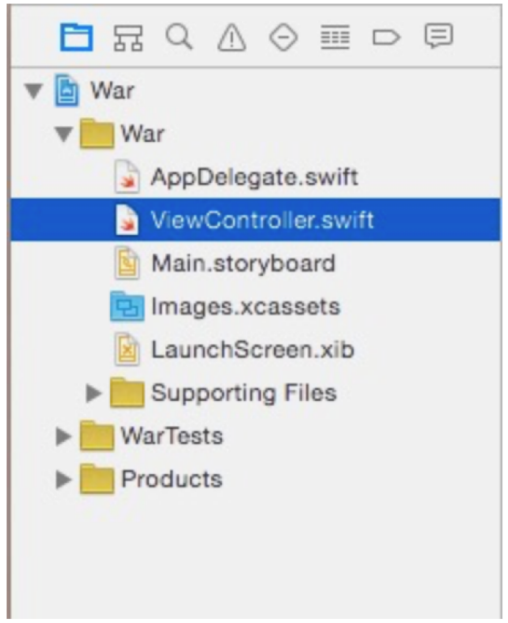
You can also download it manually if you don't have the Mac App Store. Just visit the Apple Developer page for Xcode.


Toolbar
Editor Area
Navigator Area
Debug Area
Utility Area
iPho...(8.2)
Finished running War on iPhone 6
War
War
AppDelegate.swift
ViewController.swift
Main.storyboard
Images.xcassets
LaunchScreen.xib
Supporting Files
WarTests
Products
War
War
ViewController.swift
No Selection
//
// ViewController.swift
// War
//
// Copyright (c) 2015 CodeWithChris. All rights reserved.
//
import UIKit
class ViewController: UIViewController {
@IBOutlet weak var firstCardImageView: UIImageView!
@IBOutlet weak var secondCardImageView: UIImageView!
@IBOutlet weak var playRoundButton: UIButton!
@IBOutlet weak var backgroundImageView: UIImageView!
var cardNamesArray: [String] = ["ace" ,"card2", "card3", "card4", "card5", "card6", "card7", "card8", "card9", "card10", "jack", "queen", "king"]
@IBOutlet weak var playerScore: UILabel!
@IBOutlet weak var enemyScore: UILabel!
var playerScoreTotal = 0
var enemyScoreTotal = 0
override func viewDidLoad() {
super.viewDidLoad()
// Do any additional setup after loading the view, typically from a nib.
Message from debugger: Terminated due to signal 15
All Output
Identity and Type
Name ViewController.swift
Type Default - Swift Source
Location Relative to Group
ViewController.swift
Full Path /Users/adrienvillez/Dropbox/Adrien-Chris Shared/0 - CWC Books/Downloadable code/War/War/ViewController.swift
On Demand Resource Tags
Only resources are taggable
Target Membership
War
WarTests
Text Settings
Text Encoding Default - Unicode (UTF-8)
Line Endings Default - OS X / Unix (LF)
Indent Using Spaces
Widths 4 Tab 4 Indent
Wrap lines

Navigator area

Project Navigator

This is where you'll see all the files associated with your project.



Navigator area

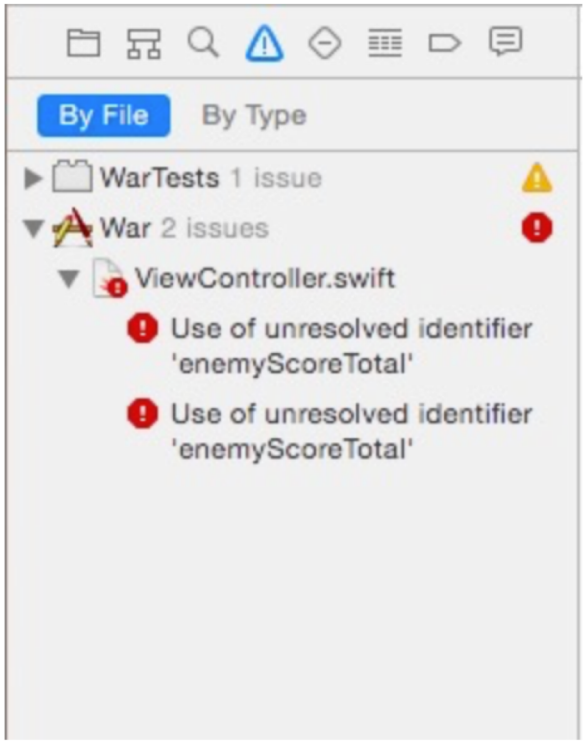
Search Navigator

With the search navigator tab, you can easily look for pieces of text in your project.

Navigator area

Issue Navigator

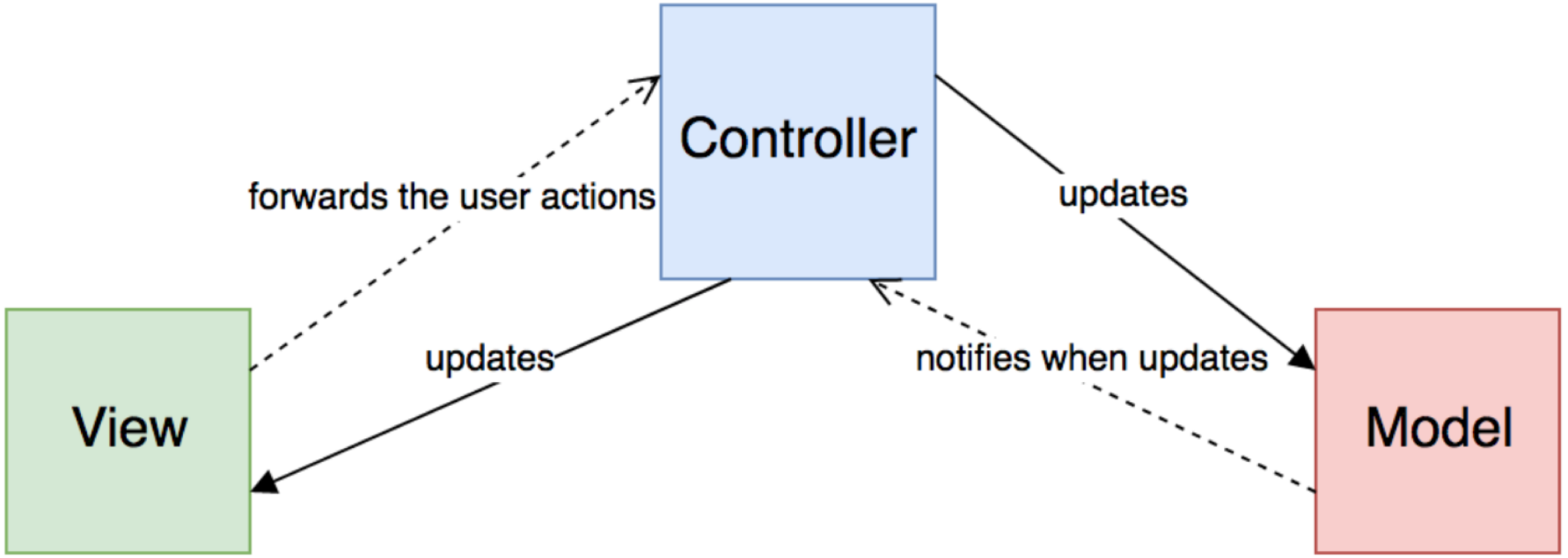
The issue navigator shows you all the problems with your app.



https://codewithchris.com/xcode-tutorial/

iOS Design Patterns

Model View Controller



Apple's MVC representation

Problems with MVC

MVC causes MVC?