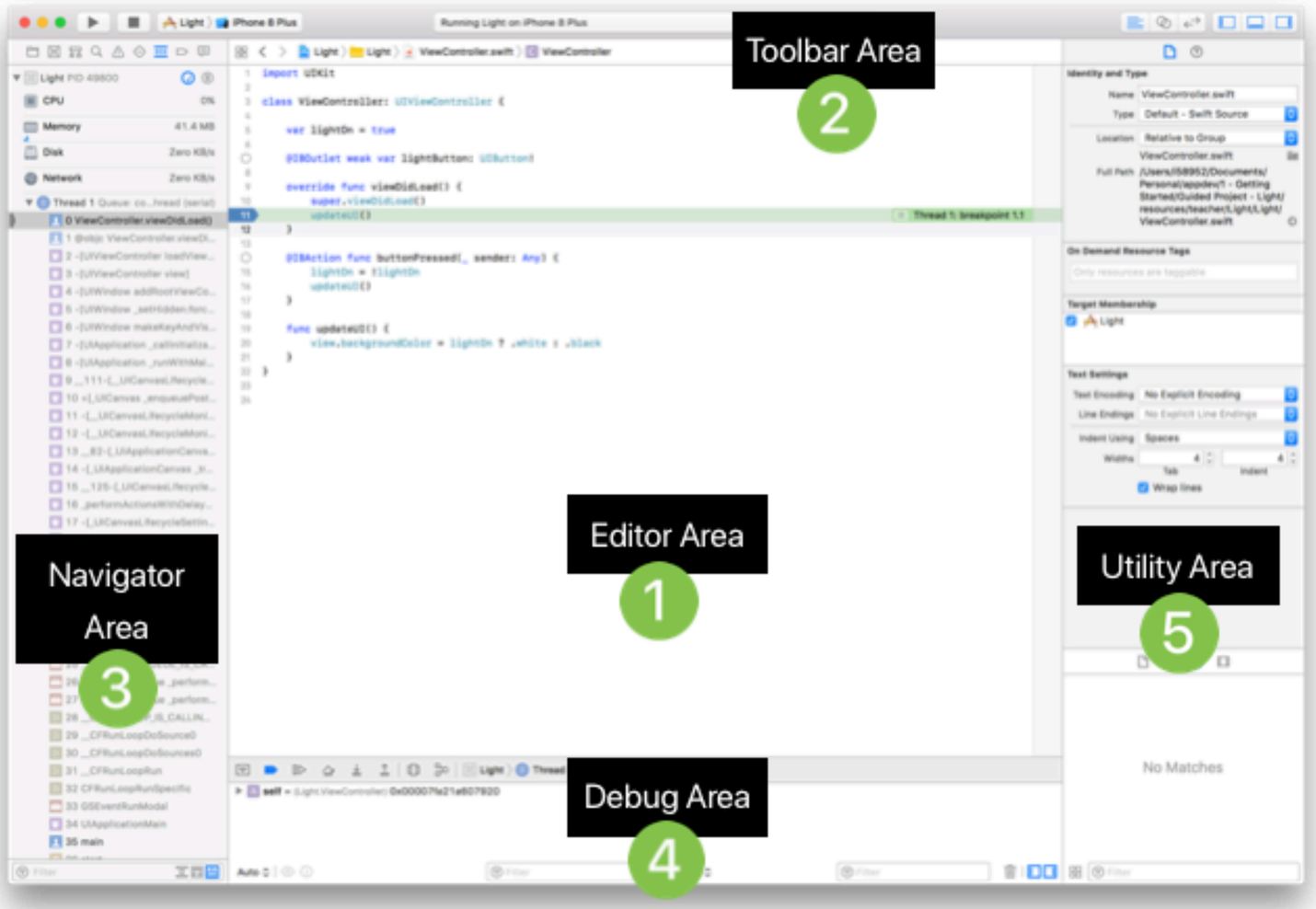
iOS Development Intro

Swift 5 + Xcode 10

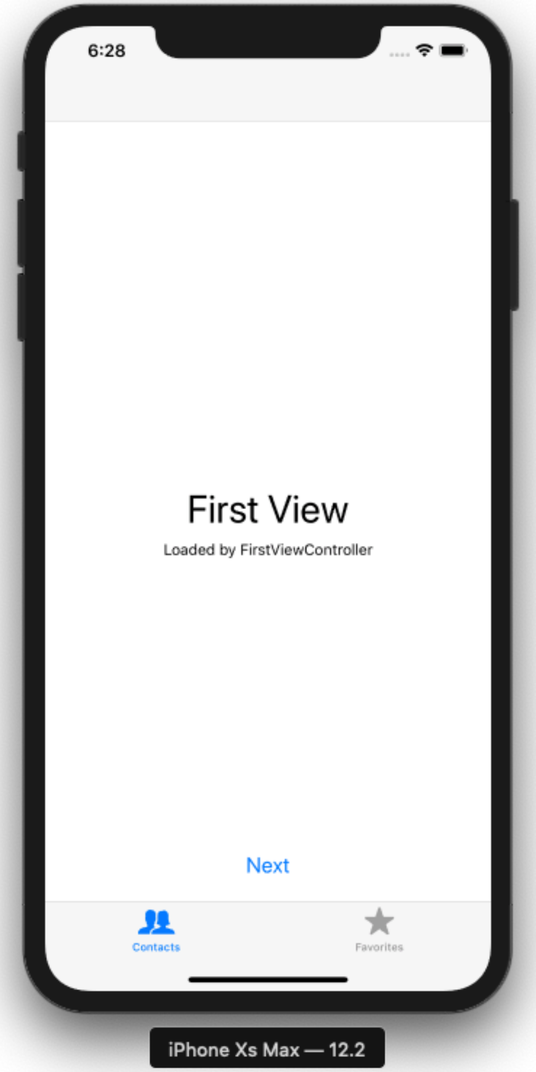
Xcode

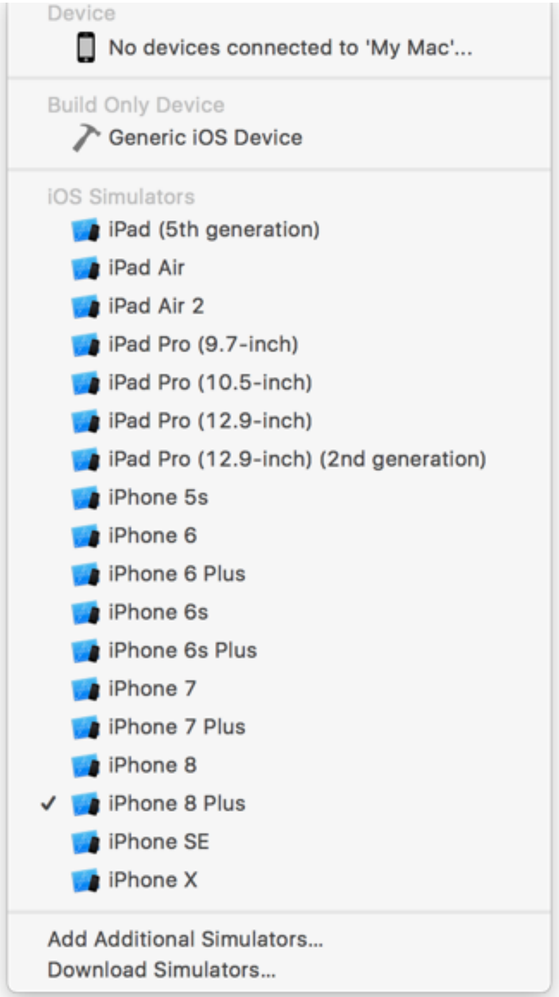

Toolbar Area
2
Editor Area
1
Navigator Area
3
Debug Area
4
Utility Area
5

Components of Xcode

- Xcode IDE
  - Integrated development environment (IDE) that enables you to manage, edit, debug your projects.
- iOS Simulator
  - Provides a software simulator to simulate an iPhone or an iPad on your Mac.
- Interface Builder
  - Visual editor for designing user interfaces for your iPhone and iPad applications.

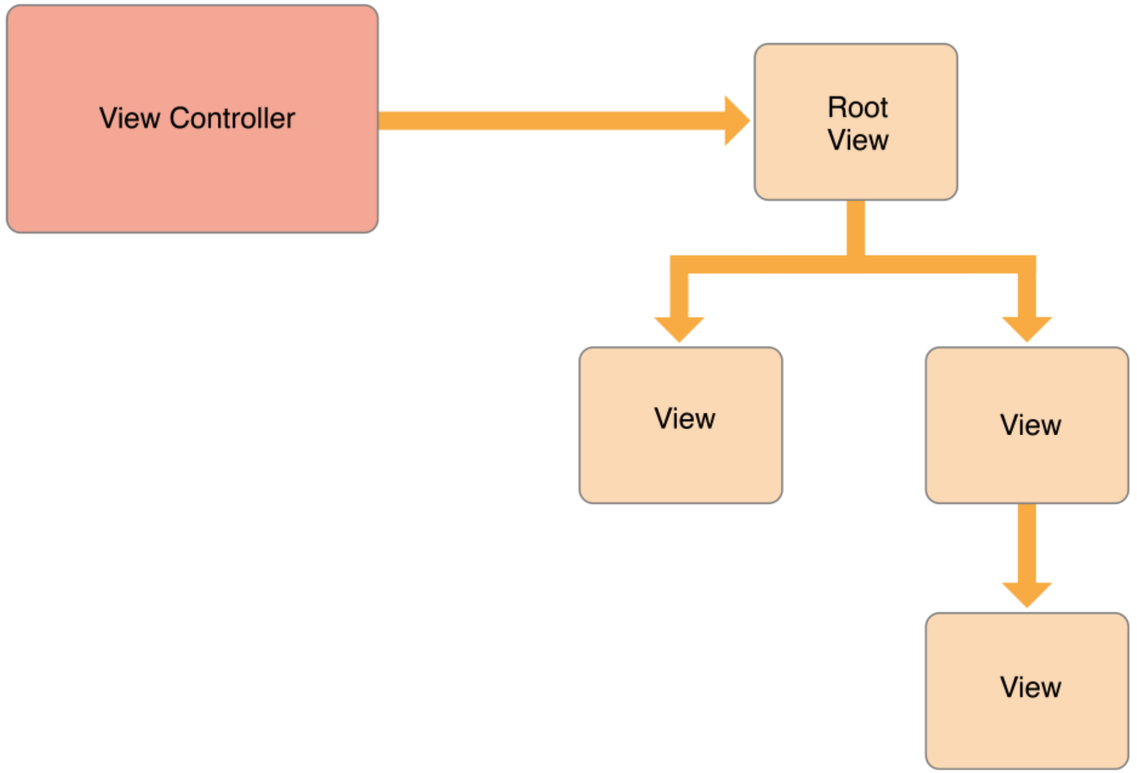
Simulator


6:28
First View
Loaded by FirstViewController
Next
Contacts
Favorites
iPhone Xs Max — 12.2

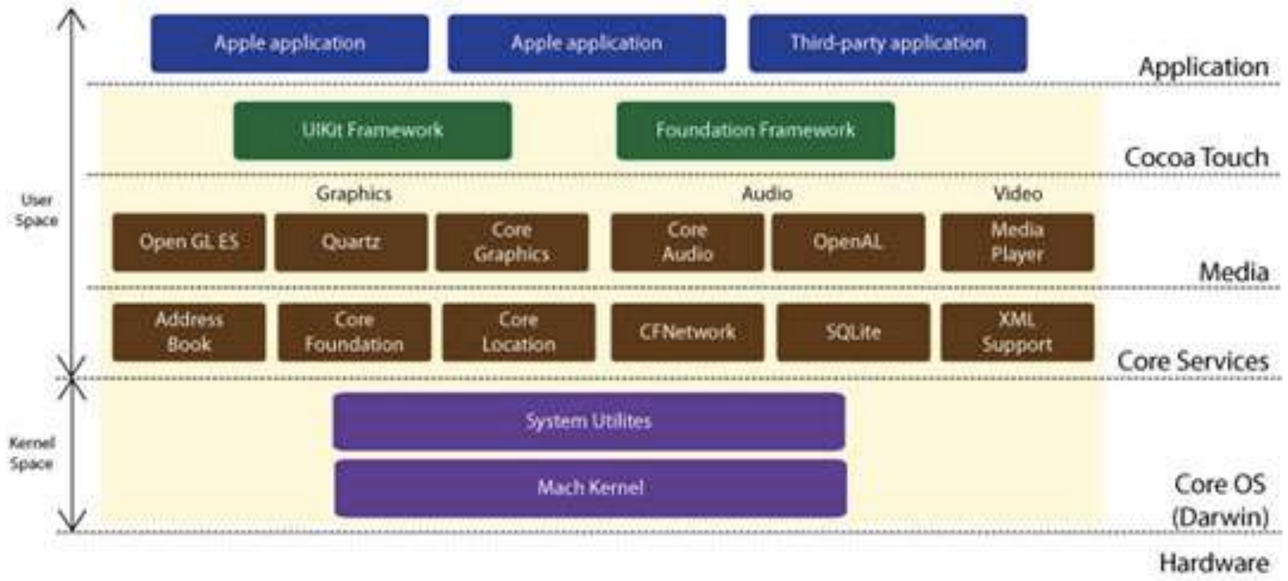

Device
No devices connected to 'My Mac'...
Build Only Device
Generic iOS Device
iOS Simulators
iPad (5th generation)
iPad Air
iPad Air 2
iPad Pro (9.7-inch)
iPad Pro (10.5-inch)
iPad Pro (12.9-inch)
iPad Pro (12.9-inch) (2nd generation)
iPhone 5s
iPhone 6
iPhone 6 Plus
iPhone 6s
iPhone 6s Plus
iPhone 7
iPhone 7 Plus
iPhone 8
iPhone 8 Plus
iPhone SE
iPhone X
Add Additional Simulators...
Download Simulators...

View controller

The most important role of a view controller is to manage a hierarchy of views. Every view controller has a single root view that encloses all of the view controller’s content.



iOS Architecture in-depth

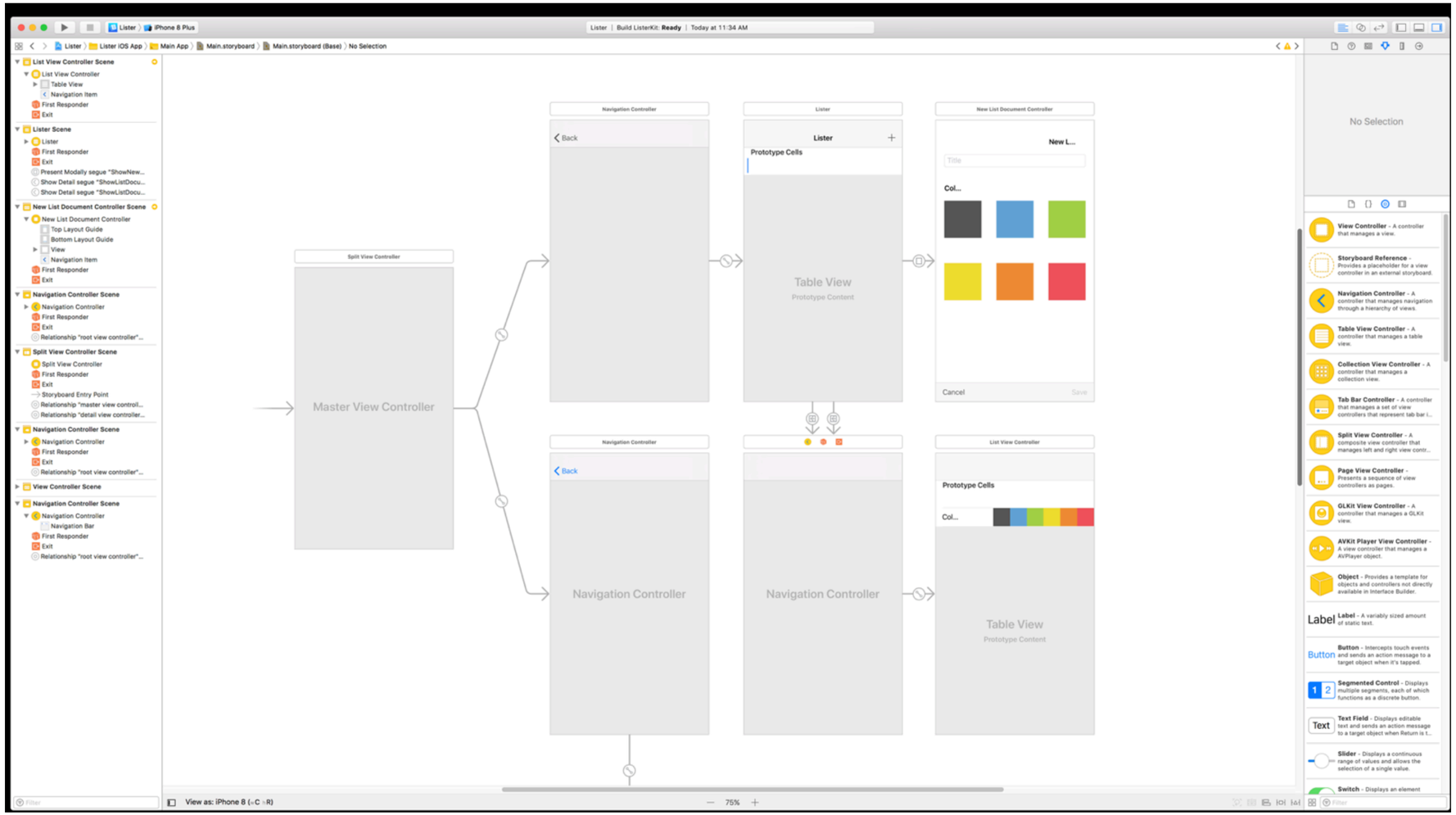


- Lower layers written in C
- Higher layers are written in Objective-C and Swift
- Higher layers are for object- oriented abstractions for lower layer constructs

iOS Architecture

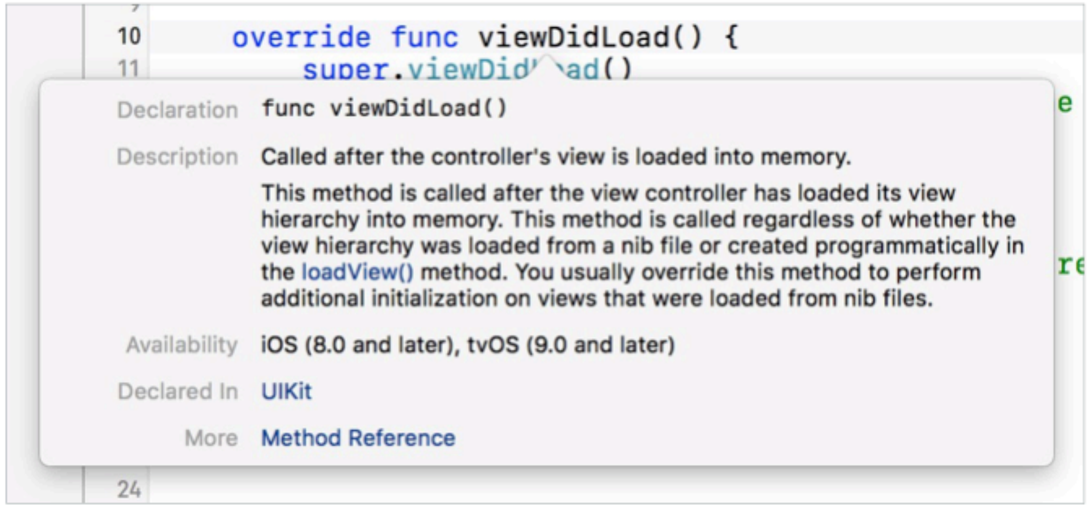
- Cocoa Touch
  - API for running applications on iOS devices.
- Media
  - The graphics, audio, and video technologies.
- Core Services
  - Provides the fundamental data types and essential services that underlie both the Cocoa and Carbon environments for both Mac OSX and iOS.
- Core OS

Storyboard


Master View Controller
Navigation Controller
Table View
Navigation Controller
Navigation Controller
Table View
Lister
Prototype Cells
New List Document Controller
Cancel
List View Controller
Prototype Cells
No Selection

Documentation

Option+Click on function to open documentation



Swift features

- Clean syntax
- Optionals
- Type inference
- Type safety
- Automatic Reference Counting (ARC) for memory management
- Tuples and multiple return values
- Generics

Constants & Variables

You define constants in Swift using the **let** keyword.

```
let name = "John"
```

You define variables in Swift using the **var** keyword.

```
var age = 29
```

This won't work:

```
let name = "John"
name = "James"
```

This will work:

```
var age = 29
age = 30
```

Common types

Type name	Symbol	Purpose	Example
Integer	`Int`	Represents whole numbers, or integers.	`4`
Double	`Double`	Represents numbers requiring decimal points.	`13.45`
Boolean	`Bool`	Represents `true` or `false` values.	`true`
String	`String`	Represents text.	`"Once upon a time..."`

Custom types & funcs

```
struct Person {
  let firstName: String
  let lastName: String

  func sayHello() {
    print("Hello there! My name is \(firstName) \(lastName).")
  }
}
```

Usage:

```
let aPerson = Person(firstName: "Jacob", lastName: "Edwards")
let anotherPerson = Person(firstName: "Candace", lastName:
"Salinas")

aPerson.sayHello()
anotherPerson.sayHello()
```

Logical operators

Operator	Description
==	Two items must be equal.
!=	The values must not be equal to each other.
>	Value on the left must be greater than the value on the right.
>=	Value on the left must be greater than or equal to the value on the right.
<	Value on the left must be less than the value on the right.
<=	Value on the left must be less than or equal to the value on the right.
&&	AND—The conditional statement on the left and right must be true.
||	OR—The conditional statement on the left or right must be true.
!	NOT—Returns the opposite of the conditional statement immediately following the operator.

Switch statement

```
let numberOfWheels = 2
switch numberOfWheels {
case 1:
    print("Unicycle")
case 2:
    print("Bicycle")
case 3:
    print("Tricycle")
case 4:
    print("Quadcycle")
default:
    print("That's a lot of wheels!")
}
```

```
switch distance {
case 0...9:
    print("Your destination is close.")
case 10...99:
    print("Your destination is a medium distance from here.")
case 100...999:
    print("Your destination is far from here.")
default:
    print("Are you sure you want to travel this far?")
}
```

```
let character = "z"

switch character {
case "a", "e", "i", "o", "u" :
    print("This character is a vowel.")
default:
    print("This character is a consonant.")
}
```

Key development parts

- Closures
- Enums
- Protocols & Delegation
- Optionals
- Value & Reference types
- ARC & Weak Reference
- Extensions
- Computed properties & Property observers
- Objective C relations

Enumeration

```swift
enum LoadingState: Int {
    case idle       = 1
    case inProgress = 3
    case error      = 5
    case success    = 10
}
```

Raw values

```swift
let state = LoadingState(rawValue: 3)
```

```swift
enum LoadingState {
    case idle
    case inProgress
    case error(message: String)
    case success(User)

    mutating func toSuccess(with user: User) {
        guard case .idle = self else { return }

        self = .success(user)
    }

}
```

Associated values

```swift
var state = LoadingState.error(message: "No network")

switch state {
case .error(let message):
    print(message)
default:
    state.toSuccess(with: User())
}
```

Optionals

And Optional value either contains value or nil

```
var a: Int? = 2 // Optional(Int)
a = 0
a = nil // Can be set to nil
```

Optionals can be unwrapped in multiple ways

```
let c = a! // Force unwrap
print(c)

if a != nil {
    print(a!)
}

if let b = a {
    print(b) // b can be used only inside if statement
}

guard let b = a else { return }
print(b) // b can be used after guard statement
```

Closures

Closures are block of functionality that can be passed around

```
let closure: (Int, Int) -> Int = { a, b in
    return a + b
}

let closureResult = closure(1, 2)
print(closureResult) // 3
```

Closure can be passed as argument in functions

```
func calculateArray(_ array: [Int], using calculation: (Int, Int) -> Int) {
    // use calculation closure
}

calculateArray([1,2,3], using: closure)
```

Protocols

- A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.
- The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements.
- Any type that satisfies the requirements of a protocol is said to *conform* to that protocol.

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

Protocols + Extensions

```
protocol TextRepresentable {
    var textForm: String { get }
}

struct Hamster {
    var name = "Ham"
}
```

Declaring Protocol Adoption with an Extension

```
struct Hamster {
    var name = "Ham"
}

extension Hamster: TextRepresentable {
    var textForm: String {
        return "This is \(name)"
    }
}
```

Protocol's default implementation

```
struct Hamster: TextRepresentable {
    var name = "Ham"
}

extension TextRepresentable {
    var textForm: String {
        return "This is somebody"
    }
}
```

Extensions

Extensions add new functionality to an existing class, structure, enumeration, or protocol type.

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

Example:

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt is now 9
```