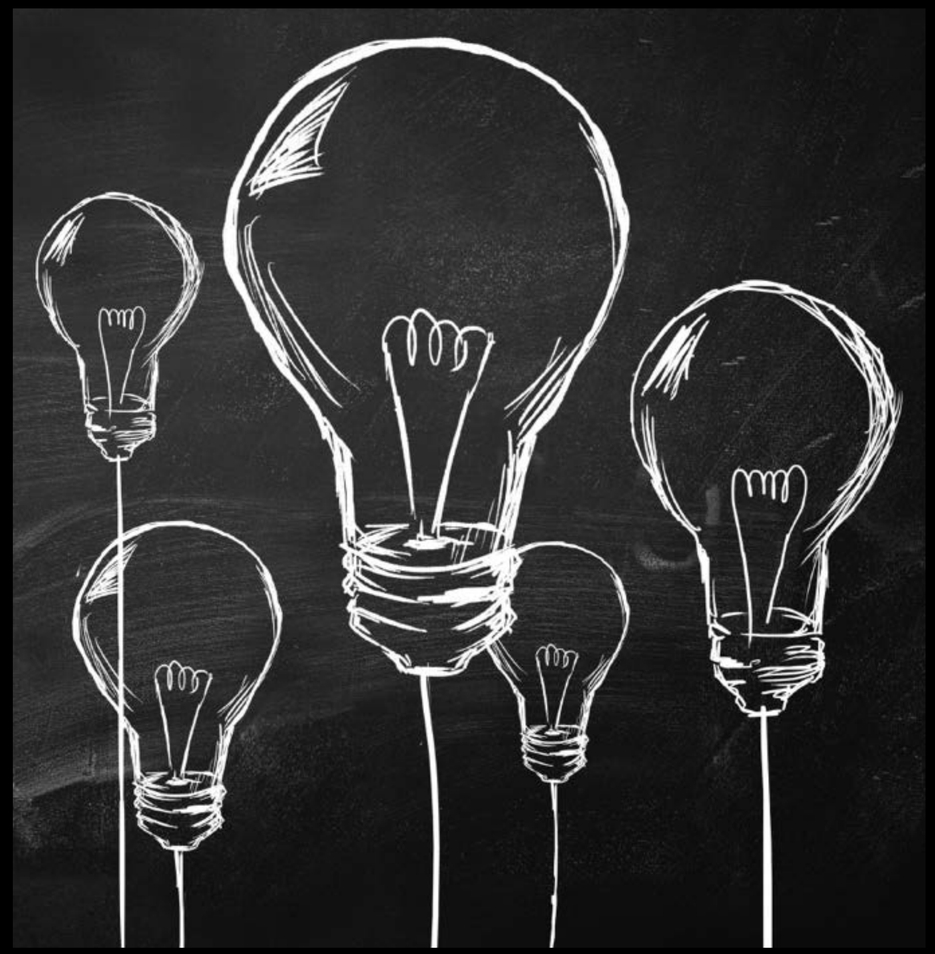
Featured #WWDC15

What's New in Swift

Session 106

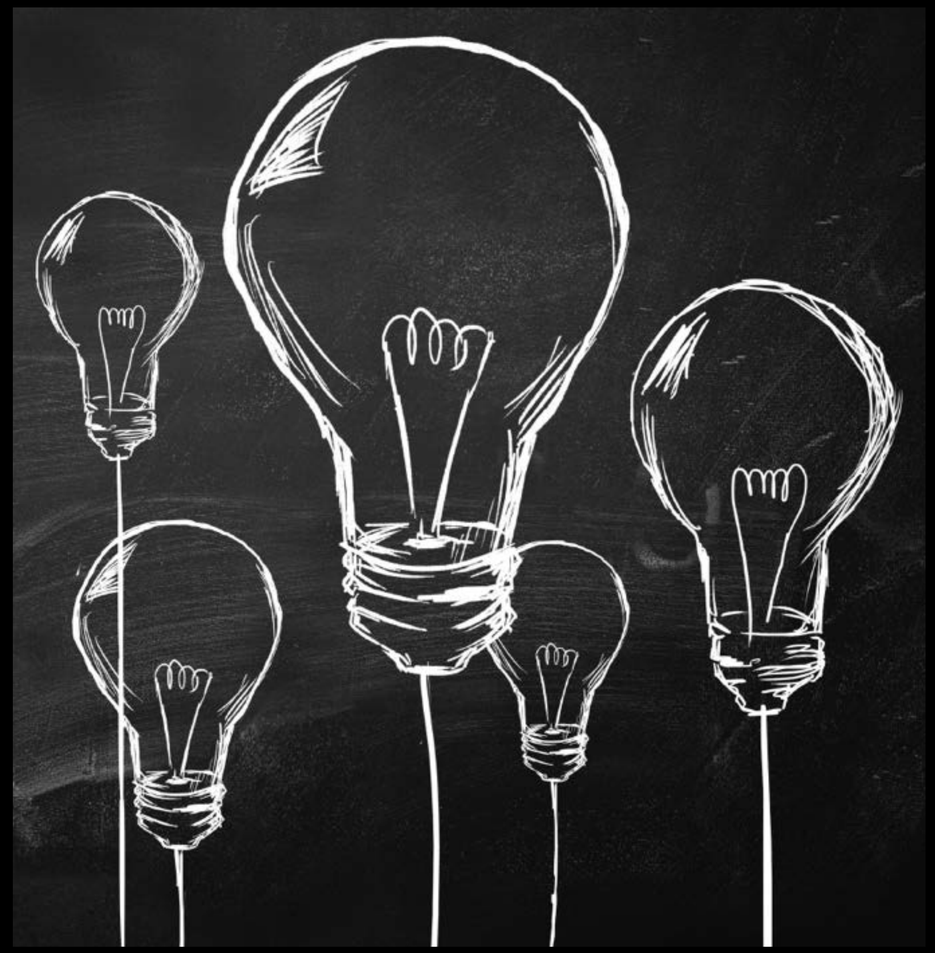
Chris Lattner Sr. Director, Developer Tools
John McCall Compiler Engineer

Goals



Goals

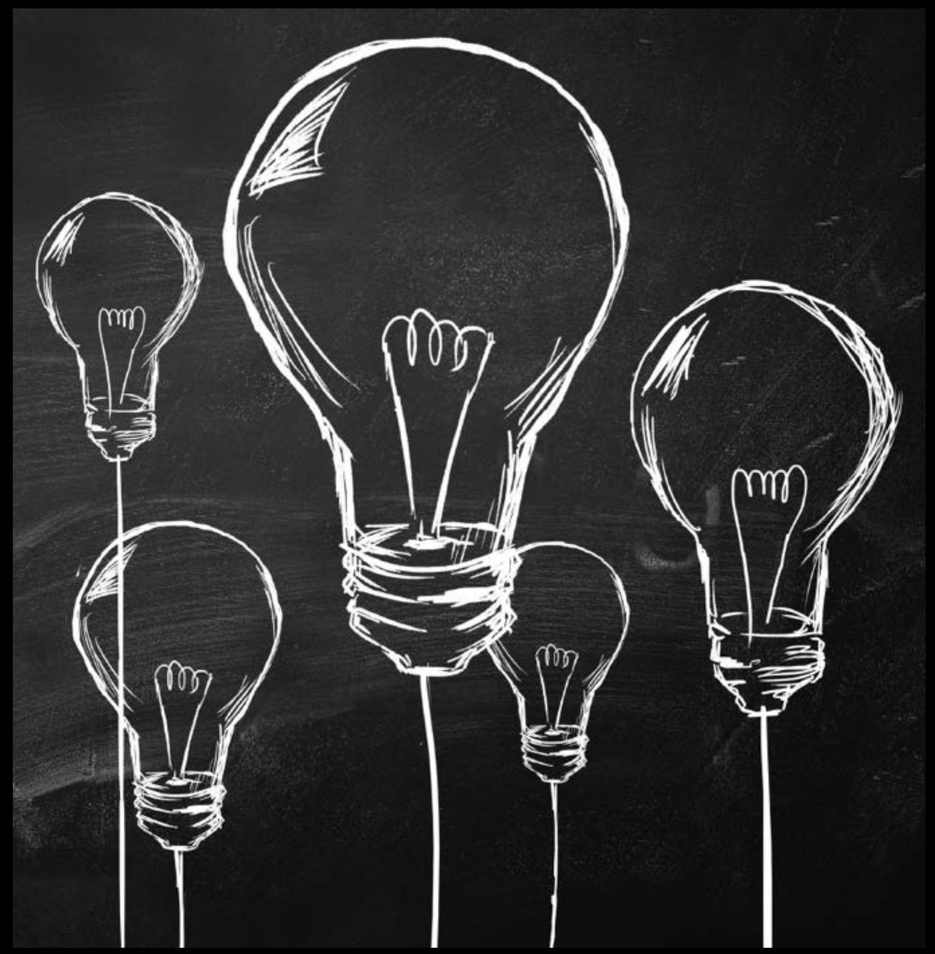
Refine language and tools fundamentals



Goals

Refine language and tools fundamentals

Affordances for writing robust and safe code

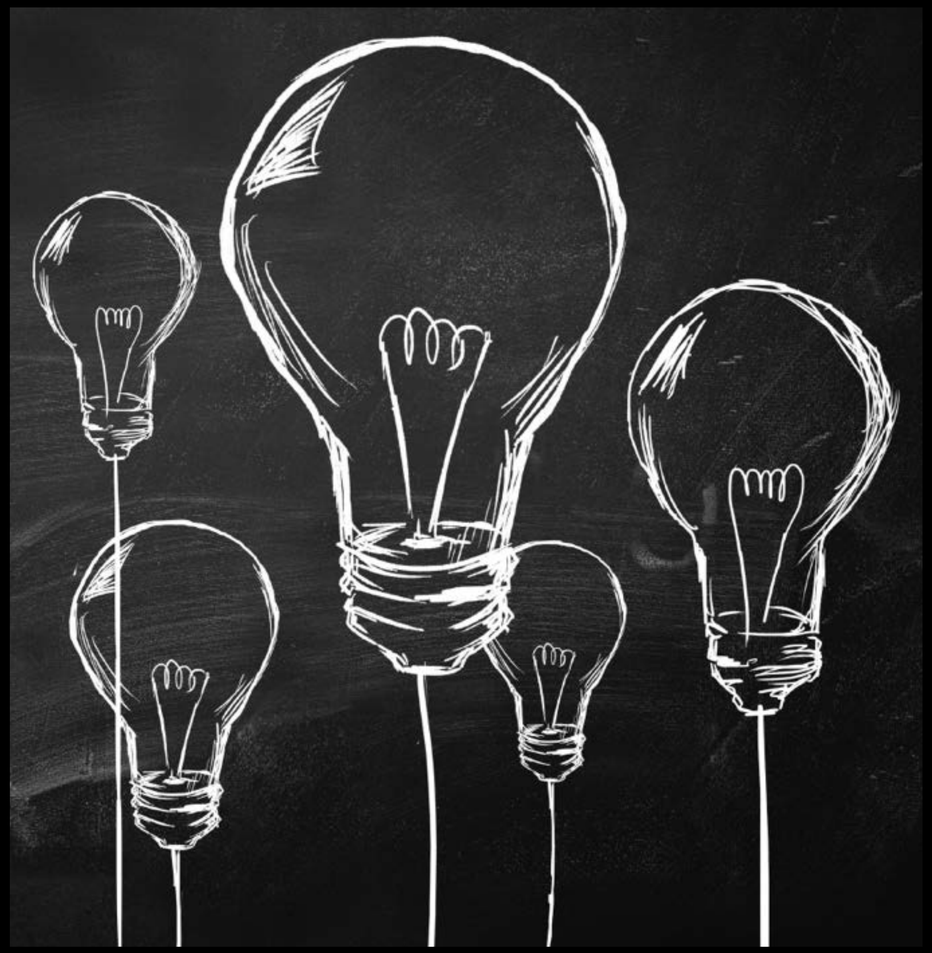


Goals

Refine language and tools fundamentals

Affordances for writing robust and safe code

Enable expressive libraries and APIs



What's New

Swift 2

What's New

Fundamentals

Pattern Matching

Availability Checking

Protocol Extensions

Error Handling

Swift 2

Fundamentals

Enums

```
enum Animals {
  case Dog, Cat, Troll, Dragon
}

let a = Animals.Dragon
```

Enums

```
enum Animals {
  case Dog, Cat, Troll, Dragon
}

let a = Animals.Dragon

print(a)
```

(Enum Value)

Enums

```
enum Animals {
  case Dog, Cat, Troll, Dragon
}

let a = Animals.Dragon

print(a)
```

```
Animals.Dragon
```

Associated Values in Enums

Associated Values in Enums

```
enum Either<T1, T2> {
  case First(T1)
  case Second(T2)
}
```

Associated Values in Enums

```
enum Either<T1, T2> {
  case First(T1)
  case Second(T2)
}
```

```
                Playground Console Output

Playground execution failed: /var/folders/3w/
j_3ky_8966j30xj_2b9v6b0h0000gq/T/./lldb/2582/
playground1.swift:28:6: error: unimplemented ir
generation feature non-fixed multi-payload enum layout
enum Either<T1, T2> {
     ^
```

Associated Values in Enums

```
enum Either<T1, T2> {
  case First(T1)
  case Second(T2)
}
```

Recursive Enums

```
enum Tree<T> {
  case Leaf(T)
   case Node(Tree, Tree)
}
```

Recursive Enums

```
enum Tree<T> {
  case Leaf(T)
   case Node(Tree, Tree)
}
```

```
! Recursive value type 'Tree<T>' is not allowed
```

Recursive Enums

```
enum Tree<T> {
  case Leaf(T)
  indirect case Node(Tree, Tree)
}
```

“do” Statement

```
let a = Animals.Troll
...
```

“do” Statement

```
do {
  let a = Animals.Troll
  ...
}
```

“do” Statement

```
do {
  let a = Animals.Troll
  ...
}
```

```
do {
  //
  //
  //
  //
  // lots of code here
  //
  //
```

“do” Statement

```
do {
  let a = Animals.Troll
  ...
}
```

```
repeat {
  //
  //
  //
  //
  // lots of code here
  //
  //
```

Option Sets

Option Sets

Swift 1:

```
viewAnimationOptions = .Repeat | .CurveEaseIn | .TransitionCurlUp
```

Option Sets

Swift 1:

```
viewAnimationOptions = .Repeat | .CurveEaseIn | .TransitionCurlUp

viewAnimationOptions = nil

if viewAnimationOptions & .TransitionCurlUp != nil {
```

Option Sets

Swift 1:

```
viewAnimationOptions = .Repeat | .CurveEaseIn | .TransitionCurlUp

viewAnimationOptions = nil

if viewAnimationOptions & .TransitionCurlUp != nil {
```

Swift 2:

```
viewAnimationOptions = [.Repeat, .CurveEaseIn, .TransitionCurlUp]
```

Option Sets

Swift 1:

```
viewAnimationOptions = .Repeat | .CurveEaseIn | .TransitionCurlUp

viewAnimationOptions = nil

if viewAnimationOptions & .TransitionCurlUp != nil {
```

Swift 2:

```
viewAnimationOptions = [.Repeat, .CurveEaseIn, .TransitionCurlUp]

viewAnimationOptions = []
```

Option Sets

Swift 1:

```
viewAnimationOptions = .Repeat | .CurveEaseIn | .TransitionCurlUp

viewAnimationOptions = nil

if viewAnimationOptions & .TransitionCurlUp != nil {
```

Swift 2:

```
viewAnimationOptions = [.Repeat, .CurveEaseIn, .TransitionCurlUp]

viewAnimationOptions = []

if viewAnimationOptions.contains(.TransitionCurlUp) {
```

Defining an Option Set

```
struct MyFontStyle : OptionSetType {
```

Defining an Option Set

```
struct MyFontStyle : OptionSetType {
   let rawValue : Int
```

Defining an Option Set

```
struct MyFontStyle : OptionSetType {
   let rawValue : Int
  static let Bold          = MyFontStyle(rawValue: 1)
  static let Italic        = MyFontStyle(rawValue: 2)
  static let Underline     = MyFontStyle(rawValue: 4)
  static let Strikethrough = MyFontStyle(rawValue: 8)
}
```

Defining an Option Set

```
struct MyFontStyle : OptionSetType {
   let rawValue : Int
  static let Bold          = MyFontStyle(rawValue: 1)
  static let Italic        = MyFontStyle(rawValue: 2)
  static let Underline     = MyFontStyle(rawValue: 4)
  static let Strikethrough = MyFontStyle(rawValue: 8)
}
myFont.style = []
myFont.style = [.Underline]
myFont.style = [.Bold, .Italic]

if myFont.style.contains(.StrikeThrough) {
```

Functions and Methods

```
func save(name: String, encrypt: Bool) { … }

class Widget {
  func save(name: String, encrypt: Bool) { … }
```

Functions and Methods

```
func save(name: String, encrypt: Bool) { … }

class Widget {
  func save(name: String, encrypt: Bool) { … }

save("thing", false)

widget.save("thing", encrypt: false)
```

Functions and Methods

```
func save(name: String, encrypt: Bool) { … }

class Widget {
  func save(name: String, encrypt: Bool) { … }

save("thing", false)
widget.save("thing", encrypt: false)
```

Swift 1 followed Objective-C:

- No label for global functions
- Labels for second argument (and later) in methods

Consistent Argument Labels

```
func save(name: String, encrypt: Bool) { … }

class Widget {
  func save(name: String, encrypt: Bool) { … }

save("thing", false)
```

```
widget.save("thing", encrypt: false)
```

Consistent Argument Labels

```
func save(name: String, encrypt: Bool) { … }

class Widget {
  func save(name: String, encrypt: Bool) { … }

save("thing", encrypt: false)
```

```
widget.save("thing", encrypt: false)
```

Consistent Argument Labels

```
func save(name: String, encrypt: Bool) { … }

class Widget {
  func save(name: String, encrypt: Bool) { … }

save("thing", encrypt: false)
widget.save("thing", encrypt: false)
```

All ‘func’ declarations are uniform:

- First argument is implied by the base function name
- Labels are used for subsequent arguments

Consistent Argument Labels

```
func save(name: String, encrypt: Bool) { … }

class Widget {
  func save(name: String, encrypt: Bool) { … }

save("thing", encrypt: false)
widget.save("thing", encrypt: false)
```

All ‘func’ declarations are uniform:

- First argument is implied by the base function name
- Labels are used for subsequent arguments

No change to imported C and Objective-C APIs

Argument Label Controls

```
func save(name: String, encrypt: Bool)
```

```
save("foo", encrypt: true)
```

Argument Label Controls

```
func save(_ name: String, encrypt: Bool)
```

```
save("foo", encrypt: true)
```

Argument Label Controls

```
func save(_ name: String, encrypt encrypt: Bool)
```

```
save("foo", encrypt: true)
```

Argument Label Controls

```
func save(_ name: String, encrypt encrypt: Bool)
```

```
save("foo", encrypt: true)
```

```
func save(name name: String, encrypt: Bool)
```

```
save(name: "foo",
     encrypt: true)
```

Duplicate first name to enable argument label

Argument Label Controls

```
func save(_ name: String, encrypt encrypt: Bool)
```

```
save("foo", encrypt: true)
```

```
func save(name name: String, encrypt: Bool)
```

```
save(name: "foo",
     encrypt: true)
```

Duplicate first name to enable argument label

```
func save(name : String, _ encrypt : Bool)
```

```
save("foo", true)
```

Underscore disables argument labels

Argument Label Controls

```
func save(_ name: String, encrypt encrypt: Bool)
```

```
save("foo", encrypt: true)
```

```
func save(name name: String, encrypt: Bool)
```

```
save(name: "foo",
     encrypt: true)
```

Duplicate first name to enable argument label

```
func save(name : String, _ encrypt : Bool)
```

```
save("foo", true)
```

Underscore disables argument labels

Much simpler model:

- Consistency between functions and methods
- No special rules for defaulted parameters
- # argument syntax has been removed

Diagnostics

Diagnostics

```
struct MyCoordinates {
    var points : [CGPoint]

    func updatePoint() {
        points[42].x = 19


    }
}
```

Diagnostics

```
struct MyCoordinates {
    var points : [CGPoint]

    func updatePoint() {
        points[42].x = 19
    }
}
```

! '@lvalue $T7' is not identical to 'CGFloat'

Diagnostics

```
struct MyCoordinates {
    var points : [CGPoint]

    func updatePoint() {
        points[42].x = 19
    }
}
```

! Cannot assign to the result of this expression

Diagnostics

```
struct MyCoordinates {
    var points : [CGPoint]

    func updatePoint() {
        points[42].x = 19
        ^
    }
}
```

! Cannot assign to ‘x’: ‘self’ is immutable

Diagnostics

```
struct MyCoordinates {
    var points : [CGPoint]

    mutating func updatePoint() {
        points[42].x = 19
    }
}
```

Cannot assign to ‘x’: ‘self’ is immutable

Fix-it Mark method ‘mutating’ to make ‘self’ mutable

Cannot assign to ‘x’: ‘self’ is immutable

New Warnings

```
func processEntries() {
    var entries = ...

    let size = entries.count
    var indices = entries.map { $0.index }
    indices.sort()
```

New Warnings

```
func processEntries() {
    var entries = ...

    let size = entries.count

    var indices = entries.map { $0.index }
    indices.sort()
```

⚠ Variable ‘entries’ was never mutated

New Warnings

```
func processEntries() {
    var entries = ...

    let size = entries.count
```

⚠ Variable ‘entries’ was never mutated

⚠ Immutable value ‘size’ was never used

```
    var indices = entries.map { $0.index }
    indices.sort()
```

New Warnings

```
func processEntries() {
    var entries = ...

    let size = entries.count
```

⚠ Variable ‘entries’ was never mutated

⚠ Immutable value ‘size’ was never used

```
    var indices = entries.map { $0.index }
    indices.sort()
```

⚠ ‘sort’ result is unused; use ‘sortInPlace’ to mutate in-place

SDK Improvements

```
class func requestHeaderFieldsWithCookies(cookies: [AnyObject]!)
    -> [NSObject : AnyObject]!
```

SDK Improvements

```
class func requestHeaderFieldsWithCookies(cookies: [AnyObject] )
   -> [NSObject : AnyObject]
```

SDK Improvements

```
class func requestHeaderFieldsWithCookies(cookies: [NSHTTPCookie])
    -> [String : String]
```

Adoption of new features and best practices:

- Nullability qualifiers
- Objective-C typed collections
- NS_ENUM, NS_OPTIONS, instancetype, @property, etc

Swift and Objective-C Interoperability	Mission	Tuesday 1:30PM
What's New in Cocoa	Presidio	Tuesday 1:30PM

Unit Testing

Unit Testing and Access Control

Unit Testing and Access Control

Your code is now built in a "compile for testing" build mode:

- Unit tests use

```
@testable
import MyApp
```

- `public` and `internal` symbols are now available

Unit Testing and Access Control

Your code is now built in a "compile for testing" build mode:

- Unit tests use

```
@testable
import MyApp
```

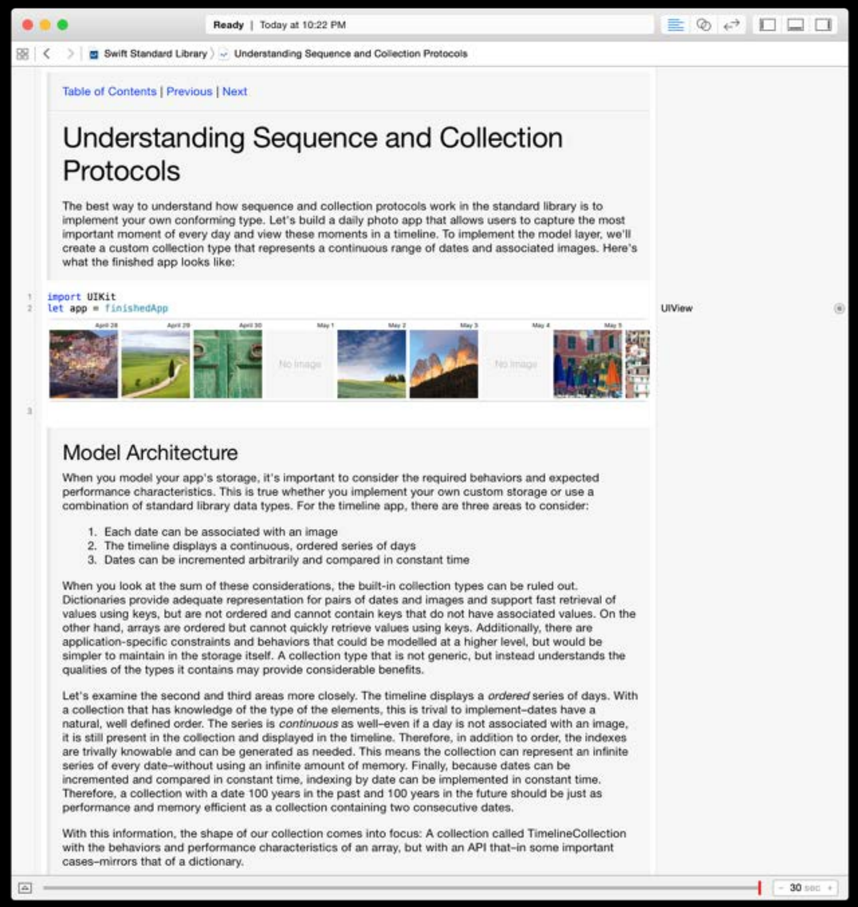
- `public` and `internal` symbols are now available

No behavior change for release builds:

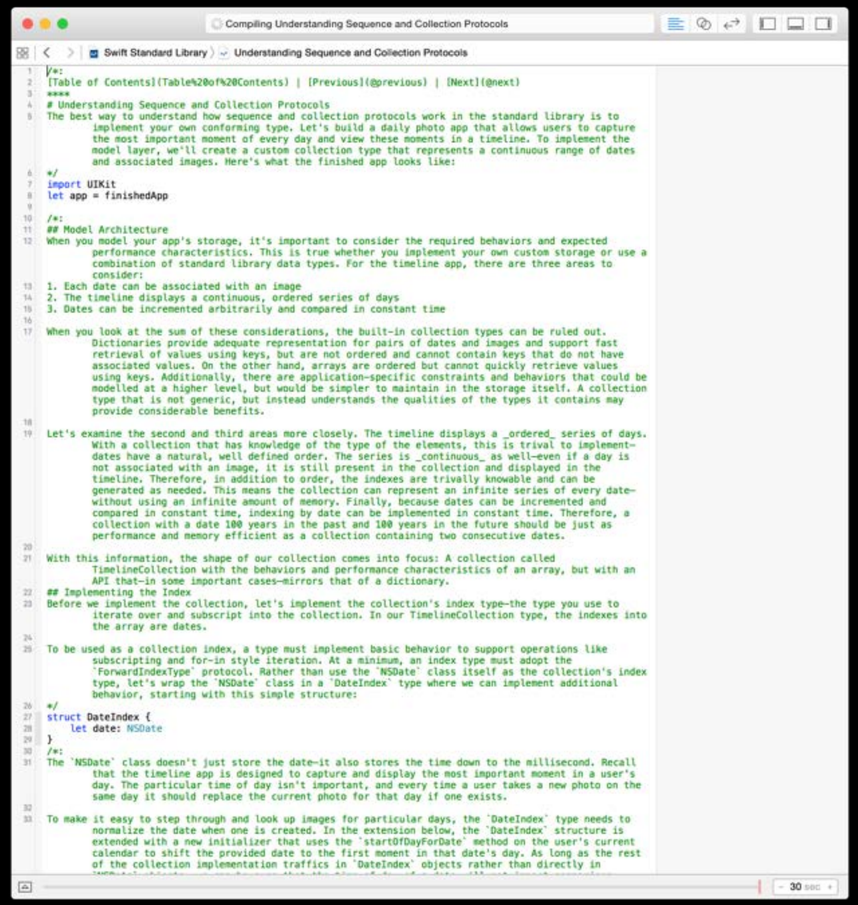
- Same optimizations
- Same insurance against symbol collisions

What's New in Xcode	Presidio	Tuesday 9:00AM
UI Testing in Xcode	Nob Hill	Wednesday 11:00AM

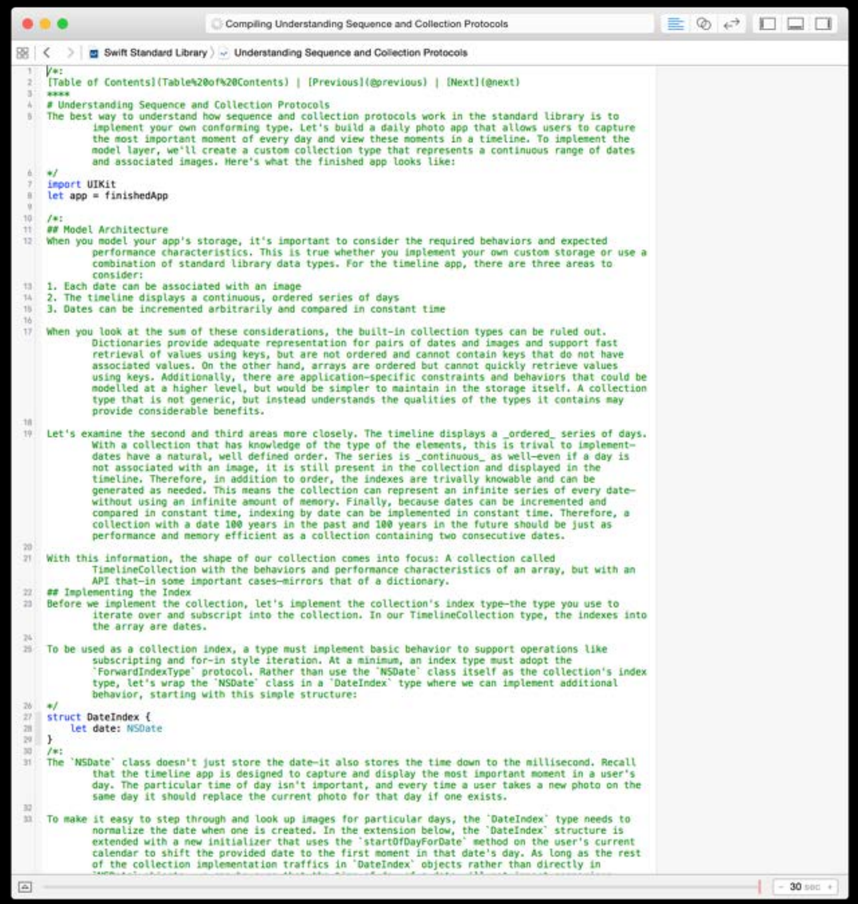
Rich Comments

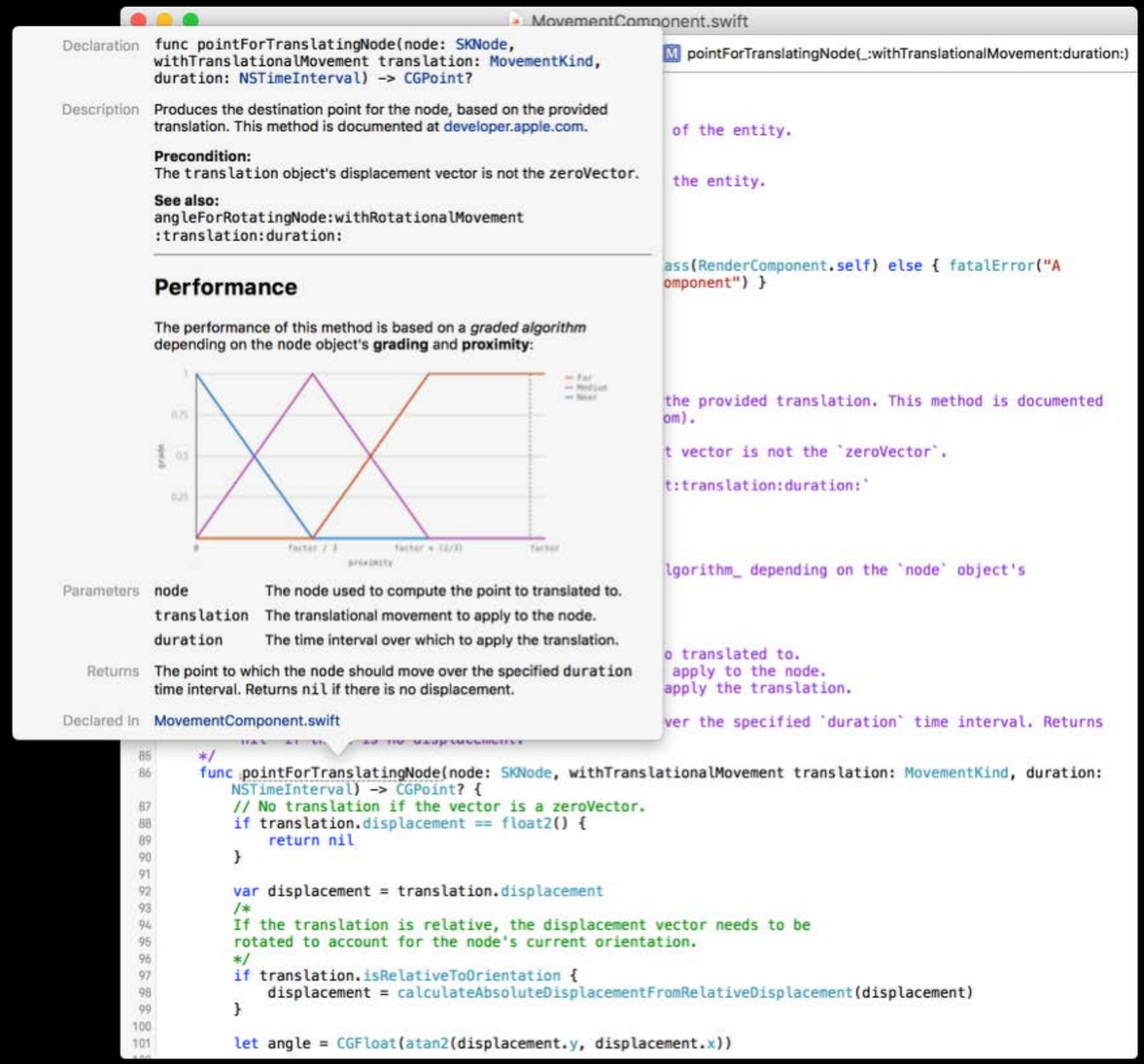


Rich Comments



Rich Comments

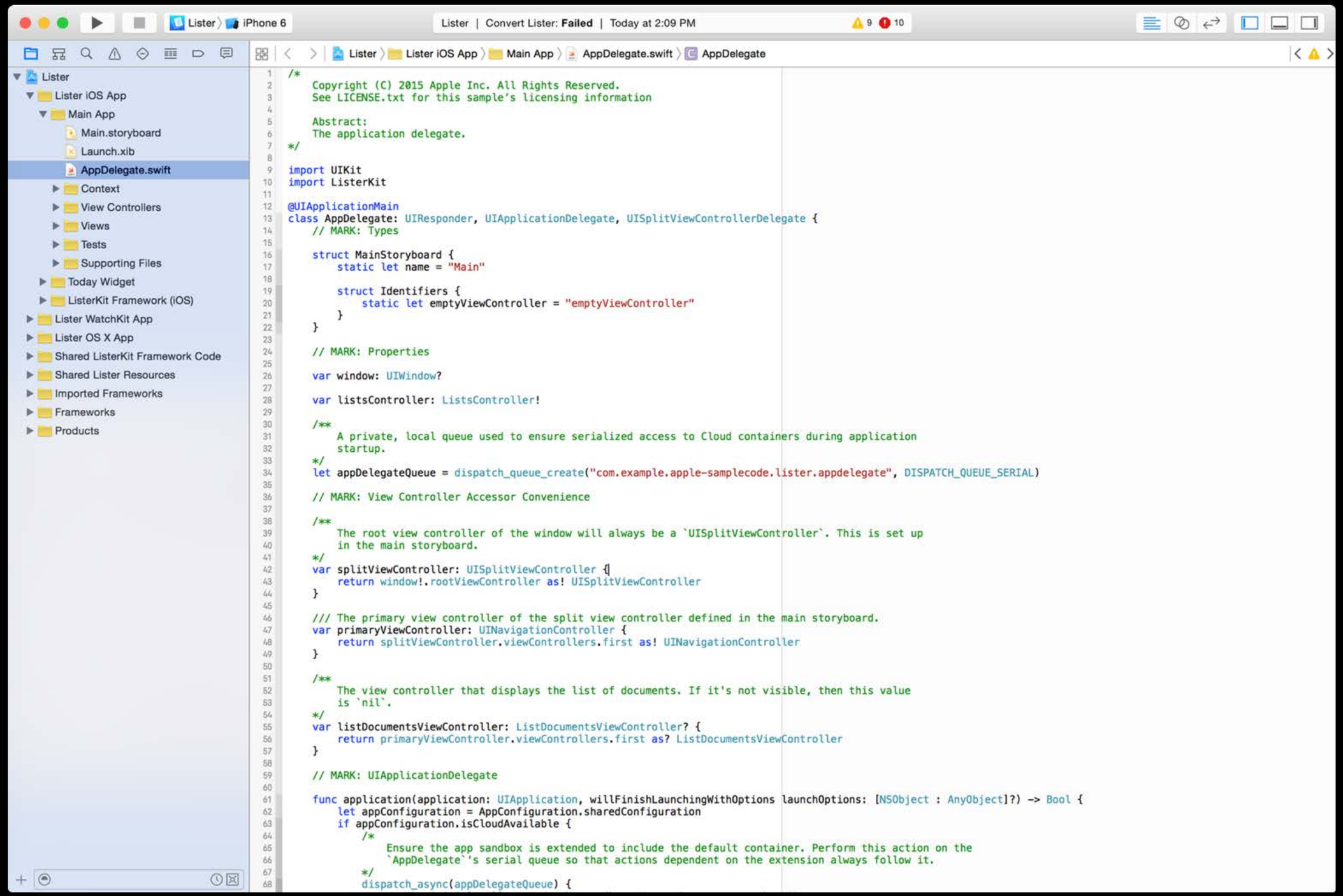


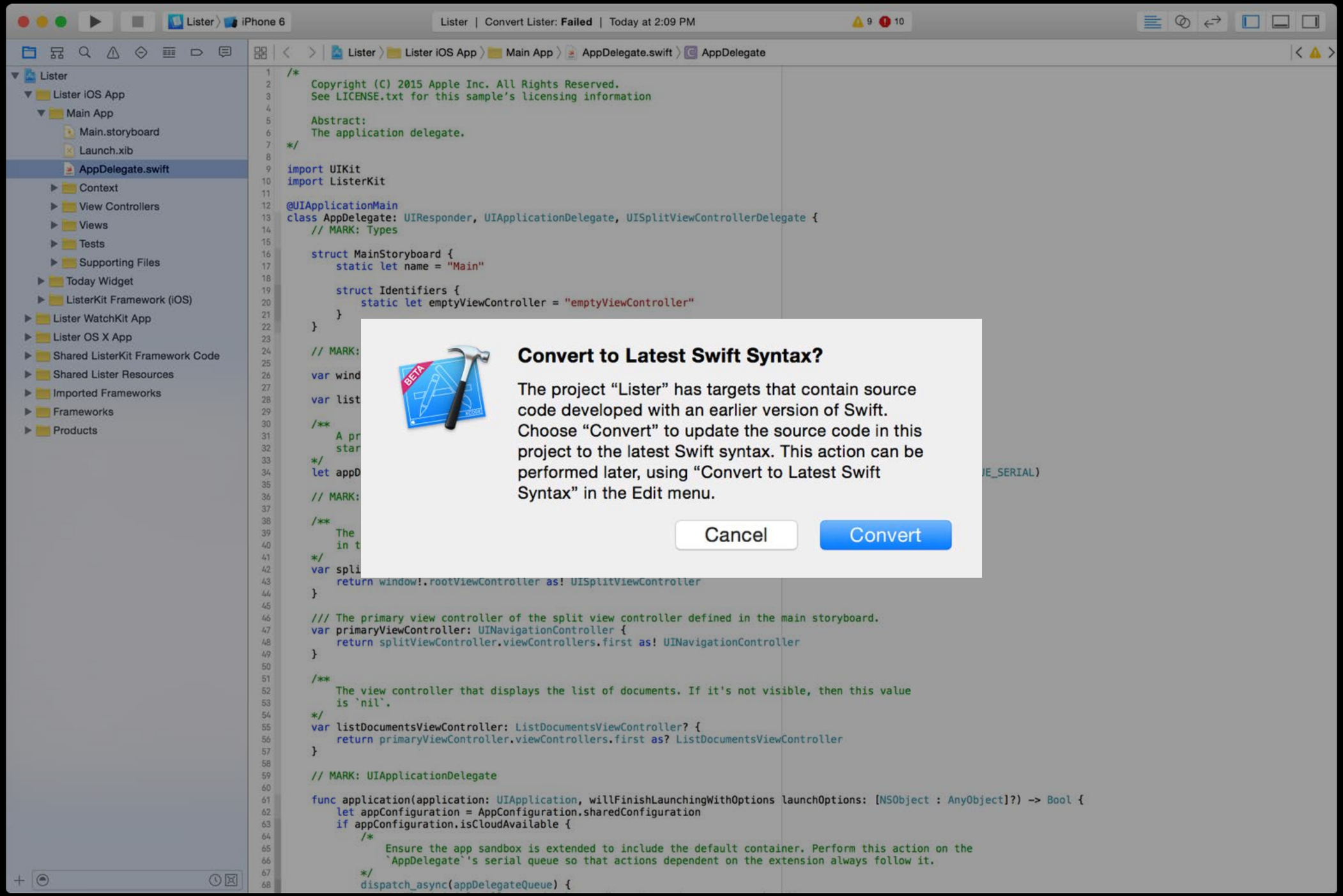


Authoring Rich Playgrounds

Presidio

Wednesday 10:00AM


Lister 〉 iPhone 6
Lister | Convert Lister: Failed | Today at 2:09 PM
9
10
Lister 〉 Lister iOS App 〉 Main App 〉 AppDelegate.swift 〉 AppDelegate
Lister
Lister iOS App
Main App
Main.storyboard
Launch.xib
AppDelegate.swift
Context
View Controllers
Views
Tests
Supporting Files
Today Widget
ListerKit Framework (iOS)
Lister WatchKit App
Lister OS X App
Shared ListerKit Framework Code
Shared Lister Resources
Imported Frameworks
Frameworks
Products
/*
Copyright (C) 2015 Apple Inc. All Rights Reserved.
See LICENSE.txt for this sample's licensing information
Abstract:
The application delegate.
*/
import UIKit
import ListerKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, UISplitViewControllerDelegate {
// MARK: Types
struct MainStoryboard {
static let name = "Main"
struct Identifiers {
static let emptyViewController = "emptyViewController"
}
}
// MARK: Properties
var window: UIWindow?
var listsController: ListsController!
/**
A private, local queue used to ensure serialized access to Cloud containers during application
startup.
*/
let appDelegateQueue = dispatch_queue_create("com.example.apple-samplecode.lister.appdelegate", DISPATCH_QUEUE_SERIAL)
// MARK: View Controller Accessor Convenience
/**
The root view controller of the window will always be a `UISplitViewController`. This is set up
in the main storyboard.
*/
var splitViewController: UISplitViewController {
return window!.rootViewController as! UISplitViewController
}
/// The primary view controller of the split view controller defined in the main storyboard.
var primaryViewController: UINavigationController {
return splitViewController.viewControllers.first as! UINavigationController
}
/**
The view controller that displays the list of documents. If it's not visible, then this value
is `nil`.
*/
var listDocumentsViewController: ListDocumentsViewController? {
return primaryViewController.viewControllers.first as? ListDocumentsViewController
}
// MARK: UIApplicationDelegate
func application(application: UIApplication, willFinishLaunchingWithOptions launchOptions: [NSObject : AnyObject]?) -> Bool {
let appConfiguration = AppConfiguration.sharedConfiguration
if appConfiguration.isCloudAvailable {
/*
Ensure the app sandbox is extended to include the default container. Perform this action on the
`AppDelegate`'s serial queue so that actions dependent on the extension always follow it.
*/
dispatch_async(appDelegateQueue) {


Lister 〉 iPhone 6
Lister | Convert Lister: Failed | Today at 2:09 PM
9 10
Lister 〉 Lister iOS App 〉 Main App 〉 AppDelegate.swift 〉 AppDelegate
Lister
Lister iOS App
Main App
Main.storyboard
Launch.xib
AppDelegate.swift
Context
View Controllers
Views
Tests
Supporting Files
Today Widget
ListerKit Framework (iOS)
Lister WatchKit App
Lister OS X App
Shared ListerKit Framework Code
Shared Lister Resources
Imported Frameworks
Frameworks
Products
Convert to Latest Swift Syntax?
The project “Lister” has targets that contain source code developed with an earlier version of Swift. Choose “Convert” to update the source code in this project to the latest Swift syntax. This action can be performed later, using “Convert to Latest Swift Syntax” in the Edit menu.
Cancel
Convert

Lister 〉 iPhone 6 | Lister | Convert Lister: **Failed** | Today at 2:09 PM | 9 | 10

Lister 〉 Lister iOS App 〉 Main App 〉 AppDelegate.swift 〉 AppDelegate

- Lister
  - Lister iOS App
    - Main App
      - Main.storyboard
      - Launch.xib
      - AppDelegate.swift
      - Context
      - View Controllers
      - Views
      - Tests
      - Supporting Files
    - Today Widget
    - ListerKit Framework (iOS)
  - Lister WatchKit App
  - Lister OS X App
  - Shared ListerKit Framework Code
  - Shared Lister Resources
  - Imported Frameworks
  - Frameworks
  - Products

```
/*
    Copyright (C) 2015 Apple Inc. All Rights Reserved.
    See LICENSE.txt for this sample's licensing information

    Abstract:
    The application delegate.
*/

import UIKit
import ListerKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, UISplitViewControllerDelegate {
    // MARK: Types

    struct MainStoryboard {
        static le

        struct Id
            stati
        }
    }

    // MARK: Prop

    var window: U

    var listsCont

    /**
        A private
        startup.
    */
    let appDelega                                                    _QUEUE_SERIAL)

    // MARK: View

    /**
        The root
        in the ma
    */
    var splitView
        return wi
    }

    /// The primary view controller of the split view controller defined in the main storyboard.
    var primaryViewController: UINavigationController {
        return splitViewController.viewControllers.first as! UINavigationController
    }

    /**
        The view controller that displays the list of documents. If it's not visible, then this value
        is `nil`.
    */
    var listDocumentsViewController: ListDocumentsViewController? {
        return primaryViewController.viewControllers.first as? ListDocumentsViewController
    }

    // MARK: UIApplicationDelegate

    func application(application: UIApplication, willFinishLaunchingWithOptions launchOptions: [NSObject : AnyObject]?) -> Bool {
        let appConfiguration = AppConfiguration.sharedConfiguration
        if appConfiguration.isCloudAvailable {
            /*
                Ensure the app sandbox is extended to include the default container. Perform this action on the
                `AppDelegate`'s serial queue so that actions dependent on the extension always follow it.
            */
            dispatch_async(appDelegateQueue) {
```

Select Targets to Convert:

- [x] Lister Tests.xctest — Lister
- [x] Lister WatchKit App.app — Lister
- [x] Lister WatchKit Extension.appex — Lister
- [x] Lister.app — Lister
- [x] Lister.appex — Lister
- [x] ListerKit.framework — Lister

Cancel | Previous | Next

Lister 〉 iPhone 6 | Lister | Convert Lister: **Failed** | Today at 2:09 PM | 9 | 10

Review Changes:

Lister 〉 Lister iOS App 〉 Main App 〉 View Controllers 〉 NewListDocumentController.swift 〉 No Selection

- AppDelegate.swift
- AppLaunchContext.swift
- ListDocumentsViewCo...
- NewListDocumentCon...
- ListViewController.swift
- AppLaunchContextTes...
- TodayViewController.swift
- ListDocument.swift
- ListsController.swift
- ListUtilities.swift
- ListCoordinator.swift
- LocalListCoordinator.s...
- ListsInterfaceControlle...
- ListInterfaceController...
- GlanceInterfaceContro...
- List.swift
- ListItem.swift
- ListPresenterDelegate...
- ListPresenterType.swift
- AllListItemsPresenter....
- IncompleteListItemsPr...
- ListPresenterAlgorith...
- ListPresenterUtilities.swift
- AppConfiguration.swift

```
62      toolbar.tintColor = selectedColor.colorValue
63  }
64
65  @IBAction func save(sender: AnyObject) {
66      let list = List()
67      list.color = selectedColor
68
69      listsController.createListInfoForList(list, withName: selectedTitle!)
70
71      dismissViewControllerAnimated(true, completion: nil)
72  }
73
74  @IBAction func cancel(sender: AnyObject) {
75      dismissViewControllerAnimated(true, completion: nil)
76  }
77
78  // MARK: Touch Handling
79
80  override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
81      super.touchesBegan(touches, withEvent: event)
82
83      let possibleTouch = touches.first as? UITouch
84
85      if let touch = possibleTouch {
86          // The user has tapped outside the text field, resign first responder,
                if necessary.
87          if nameField.isFirstResponder() && touch.view != nameField {
88              nameField.resignFirstResponder()
89          }
90      }
91  }
92
93  // MARK: UITextFieldDelegate
94
95  func textField(textField: UITextField, shouldChangeCharactersInRange range:
        NSRange, replacementString string: String) -> Bool {
96      let updatedText = (textField.text as NSString).
            stringByReplacingCharactersInRange(range, withString: string)
97      updateForProposedListName(updatedText)
98
99      return true
100 }
101
102 func textFieldDidEndEditing(textField: UITextField) {
103     updateForProposedListName(textField.text)
104 }
105
106 func textFieldShouldReturn(textField: UITextField) -> Bool {
107     textField.resignFirstResponder()
108
109     return true
110 }
```

NewListDocumentController.swift (After Conversion)

1

```
62      toolbar.tintColor = selectedColor.colorValue
63  }
64
65  @IBAction func save(sender: AnyObject) {
66      let list = List()
67      list.color = selectedColor
68
69      listsController.createListInfoForList(list, withName: selectedTitle!)
70
71      dismissViewControllerAnimated(true, completion: nil)
72  }
73
74  @IBAction func cancel(sender: AnyObject) {
75      dismissViewControllerAnimated(true, completion: nil)
76  }
77
78  // MARK: Touch Handling
79
80  override func touchesBegan(touches: Set<NSObject>, withEvent event: UIEvent) {
81      super.touchesBegan(touches, withEvent: event)
82
83      let possibleTouch = touches.first as? UITouch
84
85      if let touch = possibleTouch {
86          // The user has tapped outside the text field, resign first responder,
                if necessary.
87          if nameField.isFirstResponder() && touch.view != nameField {
88              nameField.resignFirstResponder()
89          }
90      }
91  }
92
93  // MARK: UITextFieldDelegate
94
95  func textField(textField: UITextField, shouldChangeCharactersInRange range:
        NSRange, replacementString string: String) -> Bool {
96      let updatedText = (textField.text as NSString).
            stringByReplacingCharactersInRange(range, withString: string)
97      updateForProposedListName(updatedText)
98
99      return true
100 }
101
102 func textFieldDidEndEditing(textField: UITextField) {
103     updateForProposedListName(textField.text)
104 }
105
106 func textFieldShouldReturn(textField: UITextField) -> Bool {
107     textField.resignFirstResponder()
108
109     return true
110 }
```

NewListDocumentController.swift (Before Conversion)

Cancel | Previous | Save

```
66          `AppDelegate`'s serial queue so that actions dependent on the extension always follow it.
67      */
68      dispatch_async(appDelegateQueue) {
```

So Much More…

So Much More…

Default implementations in protocols

C function pointers + closures

Recursive nested functions

if/do labeled breaks

SIMD Vectors

Mirrors API

readLine()

@nonobjc

So Much More…

Default implementations in protocols

C function pointers + closures

Recursive nested functions

if/do labeled breaks

SIMD Vectors

Mirrors API

readLine()

@nonobjc

The Swift Programming Language

Xcode 7 release notes

Pattern Matching

“if let” Statement

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if let dest = segue.destinationViewController as? BlogViewController {
        ...
    }
}
```

“if let” Statement: the “Pyramid of Doom”

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if let dest = segue.destinationViewController as? BlogViewController {
        if let blogIndex = tableView.indexPathForSelectedRow()?.row {
            if segue.identifier == blogSegueIdentifier  {
                dest.blogName = swiftBlogs[blogIndex]
                ...
                ...
            }
        }
    }
}
```

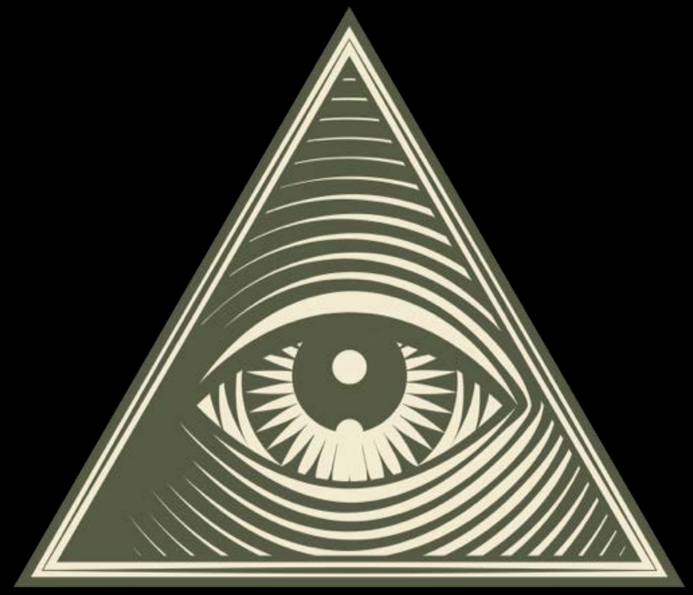
“if let” Statement: the “Pyramid of Doom”

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if let dest = segue.destinationViewController as? BlogViewController {
        if let blogIndex = tableView.indexPathForSelectedRow()?.row {
            if segue.identifier == blogSegueIdentifier  {
                dest.blogName = swiftBlogs[blogIndex]
                ...
                ...

            }
        }
    }
}
```

“if let” Statement: Compound Conditions

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if let dest = segue.destinationViewController as? BlogViewController
       let blogIndex = tableView.indexPathForSelectedRow()?.row
       where segue.identifier == blogSegueIdentifier {
           dest.blogName = swiftBlogs[blogIndex]
           ...
           ...
    }
}
```



“if let” Statement: Compound Conditions

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if let dest = segue.destinationViewController as? BlogViewController
       let blogIndex = tableView.indexPathForSelectedRow()?.row
       where segue.identifier == blogSegueIdentifier {
         dest.blogName = swiftBlogs[blogIndex]
                ...
                ...
    }
}
```

Patterns for Early Exits

```
func process(json: AnyObject) -> Either<Person,String> {
    let name: String? = json["name"] as? String
    if name == nil {
        return .Second(”missing name”)
    }
```

Patterns for Early Exits

```
func process(json: AnyObject) -> Either<Person,String> {
    let name: String? = json["name"] as? String
    if name == nil {
        return .Second(”missing name”)
    }
    let year: Int? = json["year"] as? Int
    if year == nil {
        return .Second("missing year”)
    }
```

Patterns for Early Exits

```
func process(json: AnyObject) -> Either<Person,String> {
    let name: String? = json["name"] as? String
    if name == nil {
        return .Second(”missing name”)
    }
    let year: Int? = json["year"] as? Int
    if year == nil {
        return .Second("missing year”)
    }

    let person = processPerson(name!, year!)
    return .First(person)
}
```

Patterns for Early Exits

```
func process(json: AnyObject) -> Either<Person,String> {
    let name: String? = json["name"] as? String
    if name == nil {
        return .Second(”missing name”)
    }
    let year: Int? = json["year"] as? Int
    if year == nil {
        return .Second("missing year”)
    }

    let person = processPerson(name!, year!)
    return .First(person)
}
```

Patterns for Early Exits

```
func process(json: AnyObject) -> Either<Person,String> {
    let name: String! = json["name"] as? String
    if name == nil {
        return .Second(”missing name”)
    }
    let year: Int! = json["year"] as? Int
    if year == nil {
        return .Second("missing year”)
    }

    let person = processPerson(name , year )
    return .First(person)
}
```

“guard” Statement

```
guard let name = json["name"] as? String else {
    return .Second("missing name")
}
```

“guard” Statement

```
func process(json: AnyObject) -> Either<Person,String> {
    guard let name = json["name"] as? String else {
        return .Second("missing name")
    }
    guard let year = json["year"] as? Int else {
        return .Second("missing year")
    }

    let person = processPerson(name, year)
    return .First(person)
}
```

“guard” Statement: Compound Conditions

```
func process(json: AnyObject) -> Either<Person,String> {
    guard let name = json["name"] as? String,
          let year = json["year"] as? Int else {
        return .Second(“bad input”)
    }
    let person = processPerson(name, year)
    return .First(person)
}
```

Pattern Matching with Switch

```
switch bar() {
case .MyEnumCase(let value) where value != 42:
```

Pattern Matching with Switch

```
switch bar() {
case .MyEnumCase(let value) where value != 42:
  doThing(value)

default: break
}
```

Pattern Matching with “if case”

```
switch bar() {
case .MyEnumCase(let value) where value != 42:
  doThing(value)

default: break
}
```

Pattern Matching with “if case”

```
switch bar() {
case .MyEnumCase(let value) where value != 42:
  doThing(value)

default: break
}
```

```
if case .MyEnumCase(let value) = bar() where value != 42 {
    doThing(value)
}
```

“for … in” Filtering

```
for value in mySequence {
```

“for … in” Filtering

```
for value in mySequence {
    if value != "" {
        doThing(value)
    }
}
```

“for … in” Filtering

```
for value in mySequence {
    if value != "" {
        doThing(value)
    }
}
```

```
for value in mySequence where value != "" {
    doThing(value)
}
```

“for … in” Filtering

```
for value in mySequence {
    if value != "" {
        doThing(value)
    }
}
```

```
for value in mySequence where value != "" {
    doThing(value)
}
```

```
for case .MyEnumCase(let value) in enumValues {
    doThing(value)
}
```

Pattern Matching Summary

Pattern Matching Summary

New “guard” statement for early exits

Pattern Matching Summary

New "guard" statement for early exits

'`case`' uniformly supported in control flow statements

- switch/case, if case, guard case, for-in case, while case

Pattern Matching Summary

New "guard" statement for early exits

'`case`' uniformly supported in control flow statements

- switch/case, if case, guard case, for-in case, while case

Other improvements

- `x?` pattern for optionals
- Improved exhaustiveness checker
- "Unreachable case" warning

API Availability Checking

John McCall
Type Checker

New APIs

```
extension NSButton {
  @available(OSX 10.10.3)
  var springLoaded: Bool
}
```

The Old Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  dropButton.springLoaded = true
}
```

The Old Approach

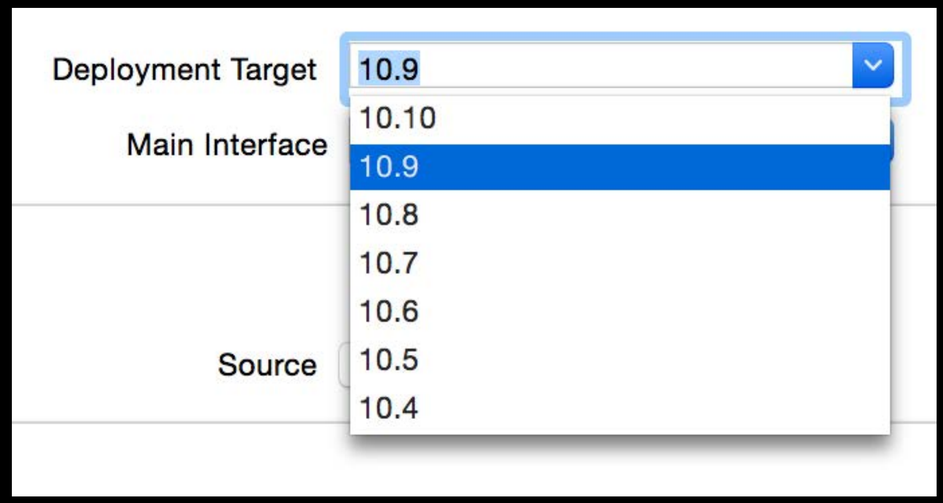
```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  dropButton.springLoaded = true
}
```

```
-[NSButton setSpringLoaded:]: unrecognized
selector sent to instance 0x100010AD0
```

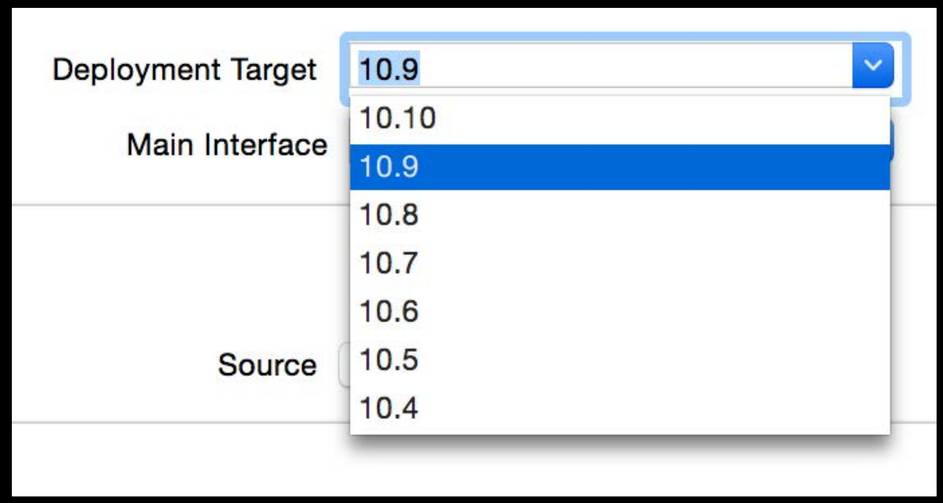
The Old Approach

```
extension NSButton {
  @available(OSX 10.10.3)
  var springLoaded: Bool
}
```



The Old Approach

```
extension NSButton {
  @available(OSX 10.10.3)
  var springLoaded: Bool
}
```



The Old Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  dropButton.springLoaded = true
}
```

The Old Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  if dropButton.respondsToSelector("setSpringLoaded") {
      dropButton.springLoaded = true
    }
}
```

The Old Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  if dropButton.respondsToSelector("setSpringLoaded") {
      dropButton.springLoaded = true
    }
}
```

The Old Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  if dropButton.respondsToSelector("setSpringLoaded:") {
      dropButton.springLoaded = true
    }
}
```

The Old Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  if dropButton.respondsToSelector("setSpringLoaded:") {
      dropButton.springLoaded = true
    }
}
```

The Better Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  dropButton.springLoaded = true
}
```

The Better Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
  dropButton.springLoaded = true
}
```

error: springLoaded is only available
on Mac OS X 10.10.3 or later

The Better Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
    if #available(OSX 10.10.3, *) {
        dropButton.springLoaded = true
    }
}
```

The Better Approach

```
@IBOutlet var dropButton: NSButton!

override func awakeFromNib() {
    if #available(OSX 10.10.3, *) {
        dropButton.springLoaded = true
    }
}
```

Availability Checking

Take advantage of new capabilities without abandoning users

Easy, expressive checks

Statically enforced by the language

Protocol Extensions

Extensions

```
extension Array {
    func countIf(match: Element -> Bool) -> Int {
        var n = 0
        for value in self {
            if match(value) { n++ }
        }
        return n
    }
}
```

Extensions

```
extension Array {
    func countIf(match: Element -> Bool) -> Int {
        var n = 0
        for value in self {
            if match(value) { n++ }
        }
        return n
    }
}
```

Extensions vs. Global Functions

```
func countIf<T : CollectionType>(collection: T,
                                 match: T.Generator.Element -> Bool) -> Int {
    var n = 0
    for value in self {
        if match(value) { n++ }
    }
    return n
}
```

Extensions

```
extension Array {
    func countIf(match: Element -> Bool) -> Int {
        var n = 0
        for value in self {
            if match(value) { n++ }
        }
        return n
    }
}
```

Protocol Extensions

```
extension Array           {
    func countIf(match: Element -> Bool) -> Int {
        var n = 0
        for value in self {
            if match(value) { n++ }
        }
        return n
    }
}
```

Protocol Extensions

```
extension CollectionType {
    func countIf(match: Element -> Bool) -> Int {
        var n = 0
        for value in self {
            if match(value) { n++ }
        }
        return n
    }
}
```

Methodification

Global generic algorithms are becoming methods

Swift 1:

```
let x = filter(map(numbers) { $0 * 3 }) { $0 >= 0 }
```

Methodification

Global generic algorithms are becoming methods

Swift 1:

```
let x = filter(map(numbers) { $0 * 3 }) { $0 >= 0 }
```

Swift 2:

```
let x = numbers.map { $0 * 3 }.filter { $0 >= 0 }
```

Protocol Extensions

More consistent use of methods in the standard library

Powerful new design patterns

Error Handling

Errors, Errors Everywhere

A big opportunity to improve programming in Swift

- Safety
- Readability
- Expressiveness

Errors, Errors Everywhere

Common problem, but the standard solutions aren't good, in different ways

- Too easy to ignore errors
- Repetitive, bug-prone patterns
- Hard to reason about implicit behavior

Kinds of Error

Trivial errors

- `Int(someString)` can fail, but an optional return type is fine

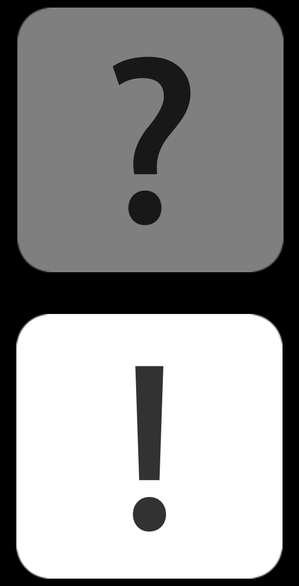
Kinds of Error

Trivial errors

- `Int(someString)`

Logic errors

- Assertions, overflows, NSException, etc.



Kinds of Error

Trivial errors

- `Int(someString)`

Detailed, recoverable errors

- File not found
- Network failure
- User cancellation

Logic errors

- Assertions, overflows, NSException, etc.

Error Handling

```
func preflight() {
    url.checkResourceIsReachable()
    resetState()
}
```

Error Handling

```
func preflight() {
    url.checkResourceIsReachable()
    resetState()
}
```

Error Handling with NSError

```
func preflight(inout error: NSError?) -> Bool {
    if url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Error Handling with NSError

```
func preflight(inout error: NSError?) -> Bool {
    if url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Downsides

```
func preflight(inout error: NSError?) -> Bool {
    if url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Adds a lot of boilerplate

Original code is hiding in there somewhere

Downsides

```
func preflight(inout error: NSError?) -> Bool {
    if url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Adds a lot of boilerplate

Original code is hiding in there somewhere

Manually implementing conventions without help

Downsides

```
func preflight(inout error: NSError?) -> Bool {
    if url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Adds a lot of boilerplate

Original code is hiding in there somewhere

Manually implementing conventions without help

Downsides

```
func preflight(inout error: NSError?) -> Bool {
    if !url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Adds a lot of boilerplate

Original code is hiding in there somewhere

Manually implementing conventions without help

Upsides

```
func preflight(inout error: NSError?) -> Bool {
    if !url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Obvious that `checkResourceIsReachable` can fail

Upsides

```
func preflight(inout error: NSError?) -> Bool {
    if !url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Obvious that `checkResourceIsReachable` can fail

Obvious that `preflight` can fail

Upsides

```
func preflight(inout error: NSError?) -> Bool {
    if !url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Obvious that `checkResourceIsReachable` can fail

Obvious that `preflight` can fail

Upsides

```
func preflight(inout error: NSError?) -> Bool {
    if !url.checkResourceIsReachableAndReturnError(&error) {
        return false
    }
    resetState()
    return true
}
```

Obvious that **`checkResourceIsReachable`** can fail

Obvious that **`preflight`** can fail

All the control flow is explicit

Error Handling in Swift

```
func preflight() {
    url.checkResourceIsReachable()
    resetState()
}
```

Error Handling in Swift

```
func preflight() {
    url.checkResourceIsReachable()
    resetState()
}
```

```
error: call can throw, but it is not marked
with 'try' and the error is not handled
```

try

```
func preflight() {
    try url.checkResourceIsReachable()
    resetState()
}
```

try

```
func preflight() {
    try url.checkResourceIsReachable()
    resetState()
}
```

error: call can throw, but the error is not handled

Propagating Errors

```
func preflight() throws {
    try url.checkResourceIsReachable()
    resetState()
}
```

Handling Errors

```
func preflight() -> Bool {
    do {
        try url.checkResourceIsReachable()
        resetState()
        return true
    } catch {
        return false
    }
}
```

Patterns with “catch”

```
func preflight() -> Bool {
    do {
        try url.checkResourceIsReachable()
        resetState()
        return true
    } catch let error {
        return false
    }
}
```

Patterns with “catch”

```
func preflight() -> Bool {
    do {
        try url.checkResourceIsReachable()
        resetState()
        return true
    } catch NSURLError.FileDoesNotExist {
        return true
    } catch let error {
        return false
    }
}
```

Aside: Impossible Errors

```
func preflight() {
    do {
        try url.checkResourceIsReachable()
        resetState()
    } catch {
        fatalError("I sure didn't expect this error: \(error)")
    }
}
```

Aside: Impossible Errors

```
func preflight() {
    try! url.checkResourceIsReachable()
    resetState()
}
```

Error Types

`ErrorType` is a protocol in the Swift standard library

Any type that conforms to `ErrorType` can be thrown and caught

- `NSError` already conforms to `ErrorType`
- Can make your own types conform as well

Enums as Error Types

`enum` is great for groups of related errors

- Can carry data in a payload
- Compiler handles protocol conformance details automatically

Defining Your Own Error Type

```
enum DataError : ErrorType {
    case MissingName
    case MissingYear
    // add more later
}
```

Defining Your Own Error Type

```
func processPerson(json: AnyObject) -> Either<Person,String> {
  guard let name = json["name"] as? String {
    return .Second("missing name")
  }
  guard let year = json["year"] as? Int {
    return .Second("missing year")
  }

  return .First(Person(name, year))
}
```

Defining Your Own Error Type

```
func processPerson(json: AnyObject) -> Either<Person,String> {
  guard let name = json["name"] as? String {
    throw DataError.MissingName
  }
  guard let year = json["year"] as? Int {
    throw DataError.MissingYear
  }

  return .First(Person(name, year))
}
```

Defining Your Own Error Type

```
func processPerson(json: AnyObject) throws -> Person {
  guard let name = json["name"] as? String {
    throw DataError.MissingName
  }
  guard let year = json["year"] as? Int {
    throw DataError.MissingYear
  }

  return Person(name, year)
}
```

Cleaning Up

```
func processSale(json: AnyObject) throws {
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }

  return Sale(buyer, price)
}
```

Notifying at the Start

```
func processSale(json: AnyObject) throws {
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }

  return Sale(buyer, price)
}
```

Notifying at the Start

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }

  return Sale(buyer, price)
}
```

Notifying at the End

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }

  return Sale(buyer, price)
}
```

Notifying at the End

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }
  delegate?.didEndReadingSale()
  return Sale(buyer, price)
}
```

Notifying at the End

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }
  delegate?.didEndReadingSale()
  return Sale(buyer, price)
}
```

Redundancy

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }
  delegate?.didEndReadingSale()
  return Sale(buyer, price)
}
```

Redundancy

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {

    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {

    throw DataError.MissingPrice
  }

  delegate?.didEndReadingSale()
  return Sale(buyer, price)
}
```

Redundancy

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    delegate?.didEndReadingSale()
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)

  guard let price = json["price"] as? Int {
    delegate?.didEndReadingSale()
    throw DataError.MissingPrice
  }

  delegate?.didEndReadingSale()
  return Sale(buyer, price)
}
```

Redundancy and Awkwardness

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    delegate?.didEndReadingSale()
    throw DataError.MissingBuyer
  }
  let buyer

  guard let price = json["price"] as? Int {
    delegate?.didEndReadingSale()
    throw DataError.MissingPrice
  }

  delegate?.didEndReadingSale()
  return Sale(buyer, price)
}
```

Redundancy and Awkwardness

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    delegate?.didEndReadingSale()
    throw DataError.MissingBuyer
  }
  let buyer : Person
  do {
    buyer = try processPerson(buyerJSON)
  } catch {
    delegate?.didEndReadingSale()
    throw error
  }

  guard let price = json["price"] as? Int {
    delegate?.didEndReadingSale()
    throw DataError.MissingPrice
```

Defer Actions

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  guard let buyerJSON = json["buyer"] as? NSDictionary {

    throw DataError.MissingBuyer
}
let buyer



guard let price = json["price"] as? Int {

  throw DataError.MissingPrice
}
```

Defer Actions

```
func processSale(json: AnyObject) throws {
  delegate?.didBeginReadingSale()
  defer { delegate?.didEndReadingSale() }
  guard let buyerJSON = json["buyer"] as? NSDictionary {
    throw DataError.MissingBuyer
  }
  let buyer = try processPerson(buyerJSON)
  guard let price = json["price"] as? Int {
    throw DataError.MissingPrice
  }

  return Sale(buyer, price)
}
```

A Quick Note About Implementation

A Quick Note About Implementation

"Exception handling" schemes

- Anything can throw
- Performance heavily biased against exceptions

A Quick Note About Implementation

"Exception handling" schemes

- Anything can throw
- Performance heavily biased against exceptions

Swift error handling is more balanced

- Not table-based unwinding
- Similar in performance to an explicit “if” statement

Works Great with Cocoa

Swift 1:

```
extension NSIncrementalStore {
  func loadMetadata(error: NSErrorPointer) -> Bool
}
```

Works Great with Cocoa

Swift 1:

```
extension NSIncrementalStore {
  func loadMetadata(error: NSErrorPointer) -> Bool
}
```

Swift 2:

```
extension NSIncrementalStore {
  func loadMetadata() throws
}
```

Works Great with Cocoa

Swift 1:

```
extension NSData {
  class func bookmarkDataWithContentsOfURL(bookmarkFileURL: NSURL,
                                           error: NSErrorPointer)
    -> NSData?
}
```

Works Great with Cocoa

Swift 1:

```
extension NSData {
  class func bookmarkDataWithContentsOfURL(bookmarkFileURL: NSURL,
                                           error: NSErrorPointer)
    -> NSData?
}
```

Swift 2:

```
extension NSData {
  class func bookmarkDataWithContentsOfURL(bookmarkFileURL: NSURL) throws
    -> NSData
}
```

Error Handling

Expressive way to handle errors

Safe, reliable programming model

Readable and maintainable

What's New in Swift 2

Swift 2

What's New in Swift 2

Fundamentals

Pattern Matching

Availability Checking

Protocol Extensions

Error Handling

Swift 2

More Information

Swift Language Documentation
http://developer.apple.com/swift

Apple Developer Forums
http://developer.apple.com/forums

Stefan Lesser
Swift Evangelist
slesser@apple.com

Related Sessions

Swift and Objective-C Interoperability	Mission	Tuesday 1:30PM
Improving Your Existing Apps with Swift	Pacific Heights	Tuesday 3:30PM
Authoring Rich Playgrounds	Presidio	Wednesday 10:00AM
Protocol-Oriented Programming in Swift	Nob Hill	Wednesday 2:30PM
Optimizing Swift Performance	Presidio	Thursday 9:00AM
Swift in Practice	Presidio	Thursday 2:30PM
Building Better Apps with Value Types in Swift	Mission	Friday 2:30PM

Labs

Swift Lab	Developer Tools Lab A	Tuesday 1:30PM
Swift Lab	Developer Tools Lab A	Wednesday 9:00AM
Swift Lab	Developer Tools Lab A	Wednesday 1:30PM
Swift Lab	Developer Tools Lab A	Thursday 9:00AM
Swift Lab	Developer Tools Lab A	Thursday 1:30PM
Swift Lab	Developer Tools Lab A	Friday 9:00AM
Swift Lab	Developer Tools Lab A	Friday 1:30PM


WWDC15