iOS Application Development

Lecture 5: Auto Layout, Protocols, and Extensions

Prof. Dr. Jan Borchers
Media Computing Group
RWTH Aachen University

WS ’22/’23 • hci.rwth-aachen.de/ios
Recap

• Optionals (Int?)
• Type casting & inspection (as?)
• Guard
• Enumeration
• Segues
• Navigation Controller
View Controllers
View Controller Life-Cycle

- View Controllers can be in one of these states:
  - View not loaded
  - View appearing
  - View appeared
  - View disappearing
  - View disappeared

```
- viewDidLoad()
- viewWillAppear(_:)`

```
- viewDidAppear(_)
- viewWillDisappear(_:)`

- viewDidDisappear(_)
- viewDidLoad()
```

Not Loaded

View Controller Life-Cycle Diagram:

1. Not Loaded
   - viewDidLoad()
   - viewWillAppear(_:)

2. View not loaded
   - viewDidAppear(_:)

3. View appearing
   - viewWillDisappear(_:)

4. View appeared
   - viewWillDisappear(_:)

5. View disappearing
   - viewWillDisappear(_):

6. View disappeared
   - viewDidLoad()
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    return true
}

func application(_ application: UIApplication, configurationForConnecting connectingSceneSession:UISceneSession, options: UIScene.ConnectionOptions) -> UISceneConfiguration {
}

func application(_ application: UIApplication, didDiscardSceneSessions sceneSessions: Set<UISceneSession>) {
}
Scene Life Cycle

Source: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle
func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {
}

func sceneDidDisconnect(_ scene: UIScene) {
}

func sceneDidBecomeActive(_ scene: UIScene) {
}

func sceneWillResignActive(_ scene: UIScene) {
}

func sceneWillEnterForeground(_ scene: UIScene) {
}

func sceneDidEnterBackground(_ scene: UIScene) {
}
Auto Layout Demo
4 Pieces of Information

- We need 4 pieces of information for Auto Layout to position a view
  - X position
  - Y position
  - Width
  - Height
Safe Area

- This is a space on the screen which usually should not contain views
- It is automatically included when adding a view controller in the storyboard
- Constraints will, by default, chose the boundaries of the safe area instead of the whole device
Constraints

- Constraints are rules by which the views are aligned using Auto Layout

- Examples for such constraints are:
  - Centring a view
  - Aligning a view relative to a margin or another view
  - Setting the size of a view relative to another view
Leading and Trailing

- *Leading* and *trailing* constraints are used to specify constraints to the right and left side of a view

- To support different language directions, we usually don’t use *left* and *right* constraints

- *Leading* constraints deal with the starting edge of a view in language direction

- *Trailing* constraints deal with the ending edge of a view in language direction
Misplacement

• If the visual representation does not match the correct layout, Auto Layout provides a warning

• Auto Layout can fix the misplacement
Intrinsic Size

• Some views already know their size e.g., labels and switches

• Such views only need information on their position for Auto Layout to handle them correctly
Hugging and Compressing

• Content Hugging Priority
  • The view does not want to grow (it hugs its content)

• Content Compression Resistance Priority
  • The view does not want to shrink

• Equal priorities might cause layout ambiguities for Auto Layout

• The view with the lower priority has to adapt to fulfil the constraints
Variants

- Variants allow us to define attributes based on the class of the used device
  - E.g., different font sizes or whether the view is visible at all

- We differentiate between the following sizes
  - Compact (e.g., smaller side of an iPhone)
  - Regular (e.g., width and height of an iPad)
  - Any (matches Compact and Regular)
Protocols
Protocols

• A protocol defines a blueprint of methods, properties and other requirements

• If your type adopts a protocol, you promise to implement these requirements

• Swift utilizes many protocols such as:
  • CustomStringConvertible
  • Equatable
  • Comparable
Sample Protocol: CustomStringConvertible

```swift
let string = "Hello, world!"
print(string) // Output: Hello, world!

let number = 42
print(number) // Output: 42

let boolean = false
print(boolean) // Output: false
```
```swift
// Sample Protocol: CustomStringConvertible

class Shoe {
    let color: String
    let size: Int
    let hasLaces: Bool

    init(color: String, size: Int, hasLaces: Bool) {
        self.color = color
        self.size = size
        self.hasLaces = hasLaces
    }
}

let myShoe = Shoe(color: "Black", size: 12, hasLaces: true)
let yourShoe = Shoe(color: "Red", size: 8, hasLaces: false)

print(myShoe)  // Output: __lldb_expr_143.Shoe
print(yourShoe) // Output: __lldb_expr_143.Shoe
```
Sample Protocol: CustomStringConvertible

class Shoe : CustomStringConvertible {
    let color: String
    let size: Int
    let hasLaces: Bool

    init(color: String, size: Int, hasLaces: Bool) {
        self.color = color
        self.size = size
        self.hasLaces = hasLaces
    }

    var description: String {
        return "\(color) shoe in size \(size), hasLaces: \(hasLaces)"
    }
}

let myShoe = Shoe(color: "Black", size: 12, hasLaces: true)
let yourShoe = Shoe(color: "Red", size: 8, hasLaces: false)

print(myShoe)
// Output:
Black shoe in size 12, hasLaces: true

print(yourShoe)
// Output:
Red shoe in size 8, hasLaces: false
Sample Protocol: Equatable

```swift
struct Employee {
    var firstName: String
    var lastName: String
    var jobTitle: String
    var phoneNumber: String
}

let employeeA = Employee(...)

let employeeB = Employee(...)

if employeeB == employeeA { // Do Something
} else { // Do Something else
}
```

error: binary operator '==' cannot be applied to two 'Employee' operands
Sample Protocol: Equatable

```swift
struct Employee : Equatable {
    var firstName: String
    var lastName: String
    var jobTitle: String
    var phoneNumber: String

    static func ==(lhs: Employee, rhs: Employee) -> Bool {
        return lhs.firstName == rhs.firstName && lhs.lastName == rhs.lastName
    }
}

let employeeA = Employee(...)
let employeeB = Employee(...)

if employeeB == employeeA {
    // Do Something
}
else {
    // Do Something else
}
```
Sample Protocol: Comparable

```swift
struct Employee : Equatable {
    var firstName: String
    var lastName: String
    var jobTitle: String
    var phoneNumber: String

    static func ==(lhs: Employee, rhs: Employee) -> Bool {
        return lhs.firstName == rhs.firstName && lhs.lastName == rhs.lastName
    }
}
```

```swift
let employees = [employee1, employee2, employee3, employee4, employee5]
let sortedEmployees = employees.sorted(by: <)
```
struct Employee : Equatable, Comparable {
    var firstName: String
    var lastName: String
    var jobTitle: String
    var phoneNumber: String

    static func ==(lhs: Employee, rhs: Employee) -> Bool {
        return lhs.firstName == rhs.firstName && lhs.lastName == rhs.lastName
    }

    static func <(lhs: Employee, rhs: Employee) -> Bool {
        return lhs.lastName < rhs.lastName
    }
}

let employees = [employee1, employee2, employee3, employee4, employee5]
let sortedEmployees = employees.sorted(by: <)
Creating a Protocol

```swift
protocol FullyNamed {
    var fullName: String { get }

    func sayFullName()
}
```
Creating a Protocol

```swift
protocol FullyNamed {
    var fullName: String { get }

    func sayFullName()
}

struct Person: FullyNamed {
    var firstName: String
    var lastName: String

    var fullName: String {
        return "\(firstName) \(lastName)"
    }

    func sayFullName() {
        print(fullName)
    }
}
```
Delegation

- Delegation is a **design pattern** to hand off (delegate) responsibilities to an instance of another type.

- UIKit and other iOS frameworks use it extensively.

- TableView example:

```swift
class MyViewController: UITableViewDataSource, UITableViewDelegate {
    ...
    myTableView.dataSource = self
    myTableview.delegate = self
}
```
Extensions
Extensions

• Extensions add functionality to types that are already defined.

• You can:
  • add **computed** properties
  • define methods
  • provide new initializers
  • conform an existing type to a protocol

• You cannot add properties!

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

extension UIColor {
    static var favoriteColor: UIColor {
        return UIColor(red: 0.5,
                        green: 0.1,
                        blue: 0.5,
                        alpha: 1.0)
    }
}
```
Extensions

```swift
struct Employee : Equatable {
    var firstName: String
    var lastName: String
    var jobTitle: String
    var phoneNumber: String

    static func ==(lhs: Employee, rhs: Employee) -> Bool {
        return lhs.firstName == rhs.firstName && lhs.lastName == rhs.lastName
    }
}

struct Employee {
    var firstName: String
    var lastName: String
    var jobTitle: String
    var phoneNumber: String
}
extension Employee : Equatable {
    static func ==(lhs: Employee, rhs: Employee) -> Bool {
        return lhs.firstName == rhs.firstName && lhs.lastName == rhs.lastName
    }
}
```
Summary

- App and ViewController Lifecycle
- AutoLayout
- Protocols
- Extensions
- No classes next Monday & Tuesday—catch up on reading
- Next lecture
  - ScrollViews and TableViews