

iOS Application Development

Lecture 4: Navigation and Workflow

11



Dr. Simon Völker & Philipp Wacker
Media Computing Group
RWTH Aachen University

Winter Semester 2017/2018

<http://hci.rwth-aachen.de/ios>

Auto Layout



Swift

- Optionals
- Type casting and inspection
- Guard
- Enumerations

Variables with nil

```
struct Book {  
    var name: String  
    var publicationYear: Int  
}  
  
let firstHarryPotter = Book(name: "Harry Potter and the Sorcerer's Stone",  
publicationYear: 1997)  
let secondHarryPotter = Book(name: "Harry Potter and the Chamber of Secrets",  
publicationYear: 1998)
```

```
let unannouncedBook = Book(name: "Rebels and Lions", publicationYear: 0)
```

- Zero isn't accurate, because that would mean the book is over 2,000 years old.

```
let unannouncedBook = Book(name: "Rebels and Lions", publicationYear: nil)
```

Optionals

- Normal variable in Swift **cannot** be nil

```
var string = nil // error!!
```

- Optionals contain either an instance of the expected type or nothing at all (nil).

```
var string: String? = nil // this works
```

```
var string: String? = "string" // this works as well
```

```
struct Book {  
    var name: String  
    var publicationYear: Int?  
}
```



Working with Optionals

- Optionals can be unwrapped using the **force-unwrap** operator !:

```
let unwrappedYear = publicationYear! //runtime error
```

- Before unwrapping an optional we need to make sure the value is not **nil**:

```
if publicationYear != nil {  
    let actualYear = publicationYear!  
    print(actualYear)  
}
```

- Shorter version:

```
if let actualYear = publicationYear {  
    print(actualYear)  
}  
else { }
```



Working with Optionals

- Unwrapping multiple optionals:

```
if let actualYear = publicationYear {  
    if let bookEdition = edition {  
        print(actualYear,bookEdition)  
    }  
}
```

```
if let actualYear = publicationYear,  
    let bookEdition = edition {  
    print(actualYear,bookEdition)  
}
```

- Optionals in functions:

```
func textFromURL(url: URL?) -> String?  
{  
    return nil  
}
```

- Failable initializers:

```
init?()  
{  
    return nil  
}
```

Optional Chaining

- Unwrapping nested optionals:

```
class Person {
    var age: Int
    var residence: Residence?
}
class Residence {
    var address: Address?
}
class Address {
    var buildingNumber: String
    var streetName: String
    var apartmentNumber: String?
}
```

```
if let theResidence = person.residence {
    if let theAddress = theResidence.address {
        if let theApartmentNumber =
            theAddress.apartmentNumber {
            print("_He/she lives in apartment
                number \ (theApartmentNumber).")
        }
    }
}
```

- Shorter version:

```
if let theApartmentNumber = person.residence?.address?.apartmentNumber {
    print("He/she lives in apartment number \ (theApartmentNumber).")
}
```



Type Casting

```
class Vehicle {}

class Car : Vehicle {}

class Motorcycle : Vehicle {}

func allVehicles() -> [Vehicle] {
    //returns the all vehicles
}

let vehicles = allVehicles()

for vehicle in vehicles {
    if let car = vehicle as? Car {
        //..
    } else if let motorcycle =
        vehicle as? Motorcycle {
        // ..
    }
}
```

- Force cast:

```
let cars = allVehiclesFrom
(manufacturer: "Porsche") as! [Car]
```

- Use **as!** only when you are certain that the specific type is correct.
- If not your app will crash



The Any Type

- The **Any** type can represent an instance of any type: String, Double, func, struct, class ...

```
var items: [Any] = [5, "Tom", 6.7, Car()]
if let firstItem = items[0] as? Int {
    print(firstItem+4) //9
}
```

- The **AnyObject** type can represent any class within Swift, but not a structure.

The Guard Command

```
func singHappyBirthday() {  
    if birthdayIsToday {  
        if invitedGuests > 0 {  
            if cakeCandlesLit {  
                print("Happy Birthday to you!")  
            } else {  
                print("The cake's candles  
                    haven't been lit.")  
            }  
        } else {  
            print("It's just a family party.")  
        }  
    } else {  
        print("No one has a birthday today.")  
    }  
}
```

```
guard condition else {  
    //false: execute some code  
}  
//true: execute some code
```

```
func singHappyBirthday() {  
    guard birthdayIsToday else {  
        print("No one has a birthday today.")  
        return  
    }  
    guard invitedGuests > 0 else {  
        print("It's just a family party.")  
        return  
    }  
    guard cakeCandlesLit else {  
        print("The cake's candles haven't  
            been lit.")  
        return  
    }  
    print("Happy Birthday to you!")  
}
```

Guard

- If statements only allow access to the constant within the braces.

```
if let eggs = goose.eggs {  
    print("The goose laid \ \(eggs.count) eggs.")  
}  
//`eggs` is not accessible here
```

- Guard statements allow access to the constant throughout the rest of the function

```
guard let eggs = goose.eggs else  
{ return }  
//`eggs` is accessible hereafter  
print("The goose laid \ \(eggs.count) eggs.")
```

- Unwrapping multiple optionals:

```
func processBook(title: String?, price: Double?, pages: Int?) {  
    guard let theTitle = title, let thePrice = price, let thePages = pages else { return }  
    print("\ \(theTitle) costs $\ \(price) and has \ \(pages) pages.")  
}
```

Enumerations

- Define a enumeration:

```
enum CompassPoint {  
    case north  
    case east  
    case south  
    case west  
}
```

```
enum CompassPoint {  
    case north, east, south, west  
}
```

- Using enumerations:

```
var compassHeading: CompassPoint = .west  
  
var compassHeading = CompassPoint.west  
  
// The compiler assigns `compassHeading` as a `CompassPoint`  
  
compassHeading = .north
```

Enumerations

- Type safety benefits:

```
struct Movie {  
    var name: String  
    var releaseYear: Int  
    var genre: String  
}
```

```
let movie = Movie(name: "Finding Dory",  
                  releaseYear: 2016,  
                  genre: "Aminated")
```

```
let movie = Movie(name: "Finding Dory",  
                  releaseYear: 2016,  
                  genre: "Tom")
```

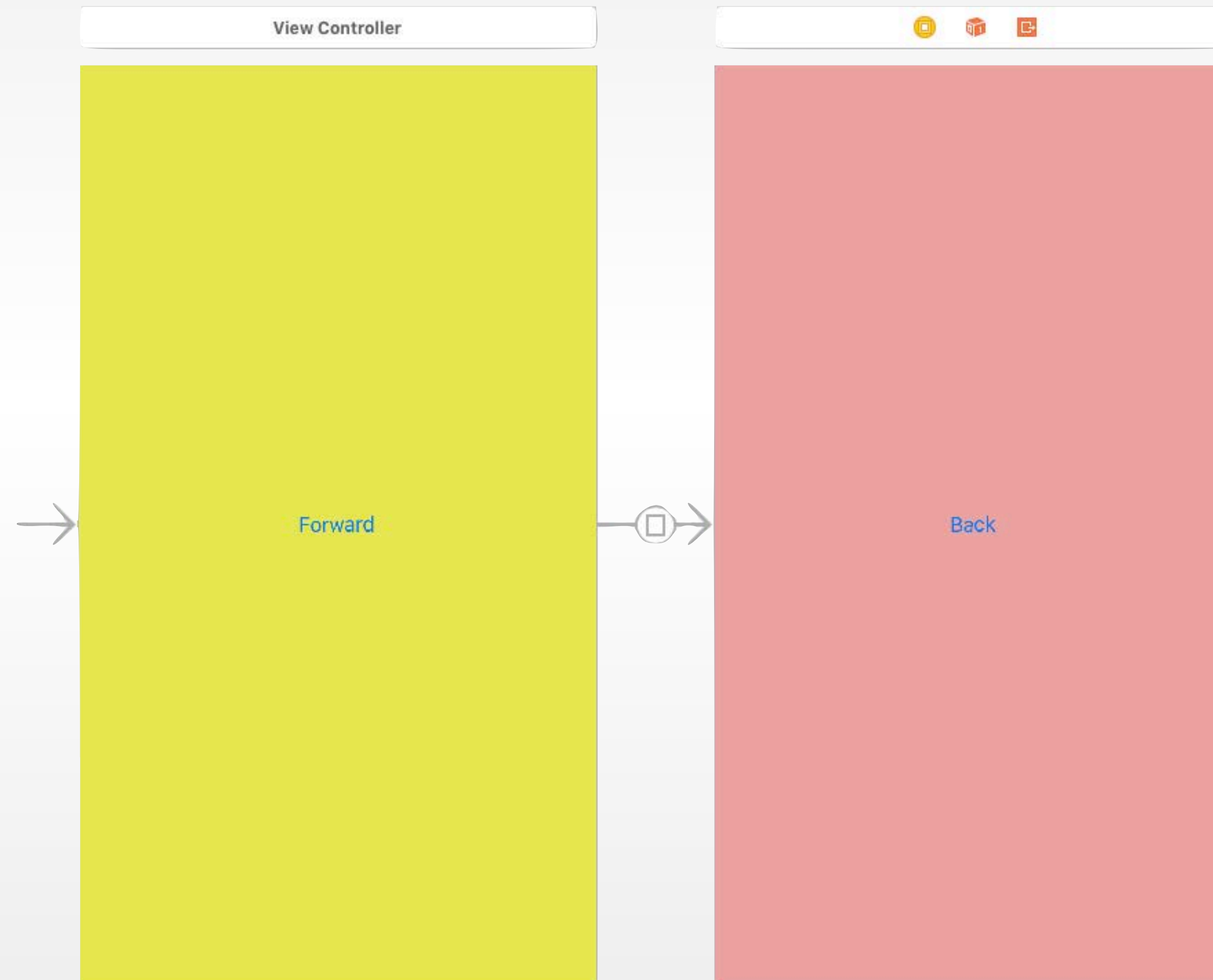
```
enum Genre {  
    case animated, action, romance,  
        documentary, biography, thriller  
}
```

```
struct Movie {  
    var name: String  
    var releaseYear: Int  
    var genre: Genre  
}
```

```
let movie = Movie(name: "Finding Dory",  
                  releaseYear: 2016,  
                  genre: .animated)
```

Segues and Navigation Controllers

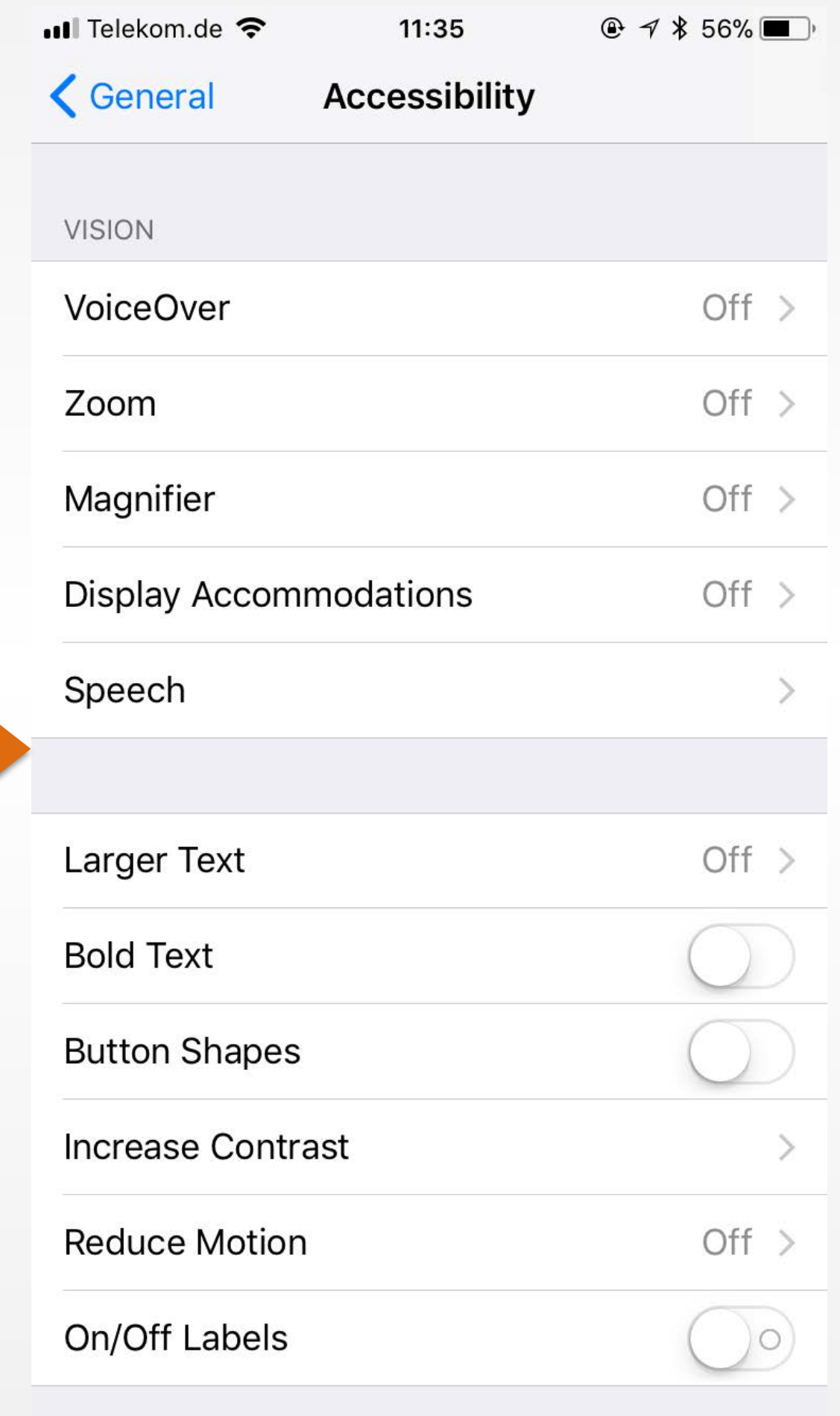
Segues



Segue Demo



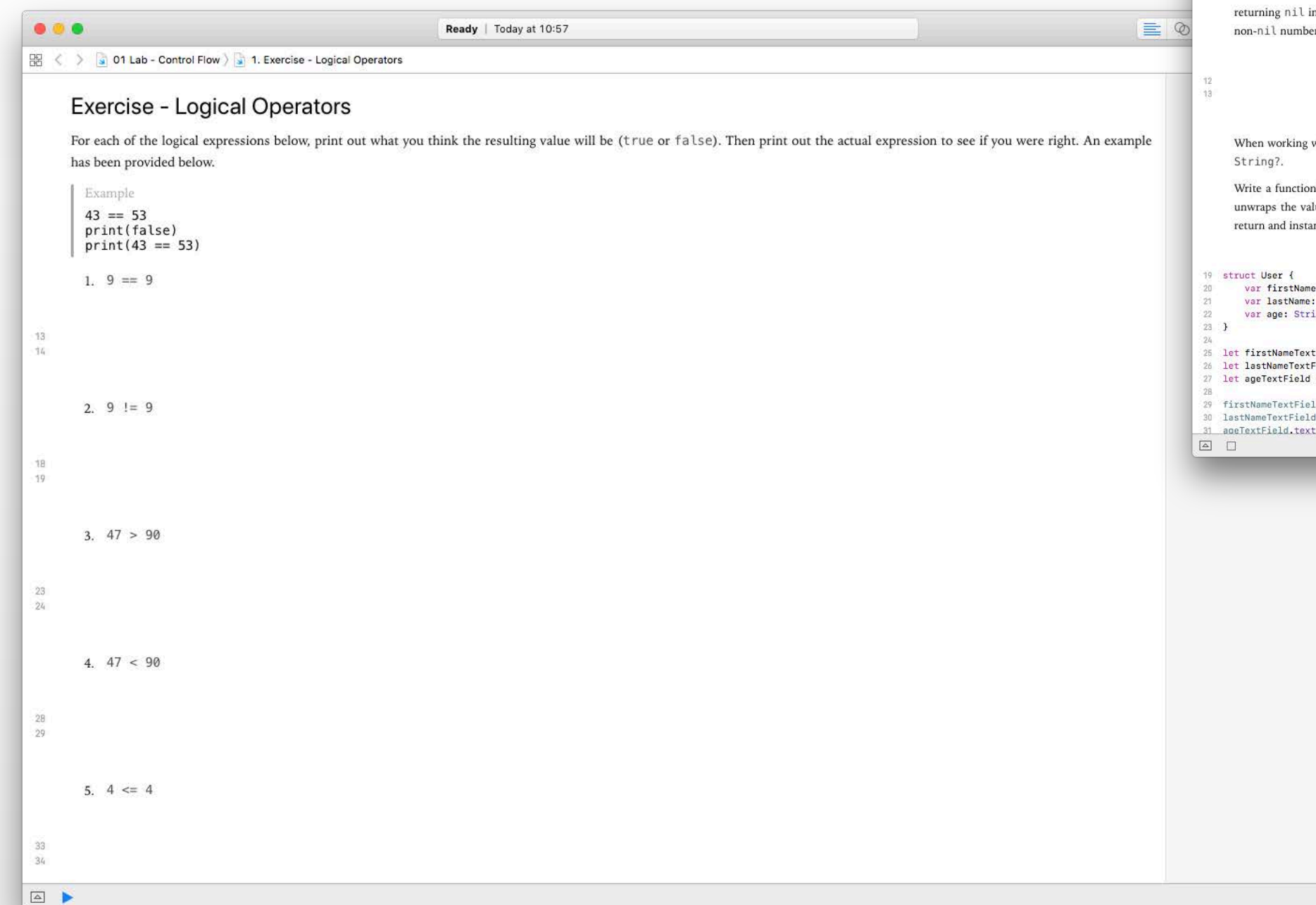
Navigation Controllers



Navigation Controllers Demo

Playgrounds

- Start learning Swift by going through the 10 Playgrounds in L2P
 - Control Flow
 - Strings
 - Functions
 - Structures
 - Classes
 - Loops
 - Optionals
 - Type Casting
 - Guard
 - Enumerations



Ready | Today at 10:57

O1 Lab - Control Flow > 1. Exercise - Logical Operators

Exercise - Logical Operators

For each of the logical expressions below, print out what you think the resulting value will be (true or false). Then print out the actual expression to see if you were right. An example has been provided below.

Example

```
43 == 53
print(false)
print(43 == 53)
```

1. 9 == 9

13
14

2. 9 != 9

18
19

3. 47 > 90

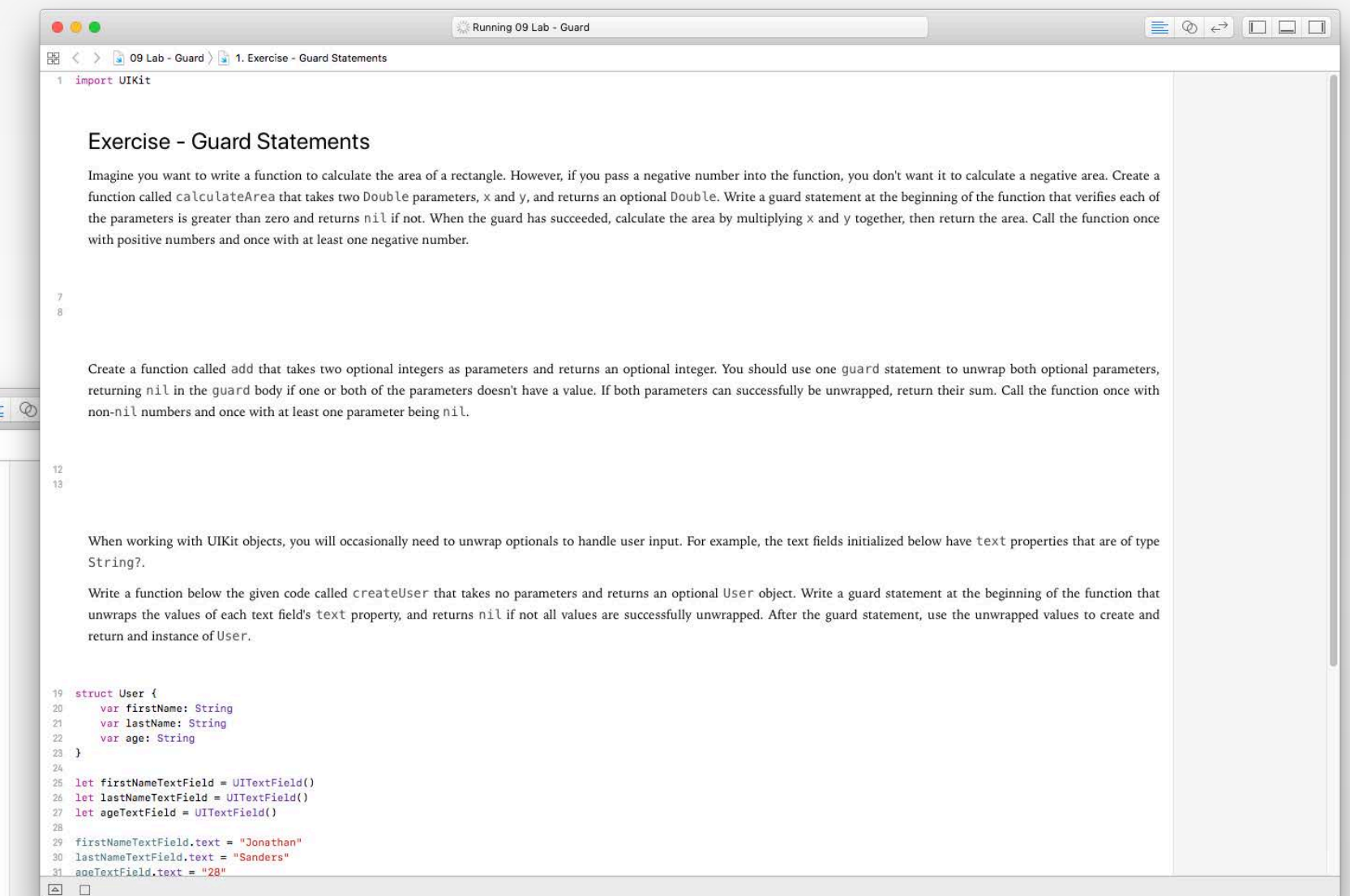
23
24

4. 47 < 90

28
29

5. 4 <= 4

33
34



Running O9 Lab - Guard

O9 Lab - Guard > 1. Exercise - Guard Statements

Exercise - Guard Statements

Imagine you want to write a function to calculate the area of a rectangle. However, if you pass a negative number into the function, you don't want it to calculate a negative area. Create a function called `calculateArea` that takes two `Double` parameters, `x` and `y`, and returns an optional `Double`. Write a guard statement at the beginning of the function that verifies each of the parameters is greater than zero and returns `nil` if not. When the guard has succeeded, calculate the area by multiplying `x` and `y` together, then return the area. Call the function once with positive numbers and once with at least one negative number.

7
8

Create a function called `add` that takes two optional integers as parameters and returns an optional integer. You should use one guard statement to unwrap both optional parameters, returning `nil` in the guard body if one or both of the parameters doesn't have a value. If both parameters can successfully be unwrapped, return their sum. Call the function once with non-`nil` numbers and once with at least one parameter being `nil`.

12
13

When working with UIKit objects, you will occasionally need to unwrap optionals to handle user input. For example, the text fields initialized below have `text` properties that are of type `String?`.

Write a function below the given code called `createUser` that takes no parameters and returns an optional `User` object. Write a guard statement at the beginning of the function that unwraps the values of each text field's `text` property, and returns `nil` if not all values are successfully unwrapped. After the guard statement, use the unwrapped values to create and return an instance of `User`.

```
19 struct User {
20     var firstName: String
21     var lastName: String
22     var age: String
23 }
24
25 let firstNameTextField = UITextField()
26 let lastNameTextField = UITextField()
27 let ageTextField = UITextField()
28
29 firstNameTextField.text = "Jonathan"
30 lastNameTextField.text = "Sanders"
31 ageTextField.text = "28"
```

Access to Macs

- 8 Mac's in the RBI
- Ahornstr. 55, E1 basement
- Open:
 - Mo. - Th. 9:00 - 20:00
 - Fr. 9:00 - 17:00
- Account: guest, no pw
- Logout deletes all of your data

