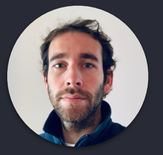
SORBONNE UNIVERSITÉ

Introduction to iOS development with Swift

Lesson 4



Adrien Humilière
Dashlane

adhumi+dant@gmail.com

- Concurrency
- Protocols
- Saving Data

Concurrency

Concurrency

- → Run multiple tasks at the same time
- → Run slow or expensive tasks in the background
- → Free the main thread so it responds to the UI

Synchronous and asynchronous

Synchronous

→ One task completes before another begins

→ Ties up the main thread (main queue)

Asynchronous

→ Multiple tasks run simultaneously on multiple threads (concurrency)

→ Tasks run in the background thread (background queue)

→ Frees up the main thread

Grand Central Dispatch

Grand Central Dispatch

- Allows your app to execute multiple tasks concurrently on multiple threads
- Assigns tasks to "dispatch queues" and assigns priority
- Controls when your code is executed

Grand Central Dispatch

- Main queue
  - Created when an app launches
  - Highest priority
  - Used to update the UI and respond quickly to user input
- Background queues
  - Lower-priority
  - Used to run long-running operations

Dispatch Queue

Use the DispatchQueue type to create and assign tasks to different queues

For example:

→ Assign a UI task to the main dispatch queue

→ Tasks added with main.async(...) run sequentially

```
DispatchQueue.main.async {
    // Code here will be executed on the main queue
}
```

Protocols

Protocols

- Defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality
- Swift standard library defines many protocols, including these:

```
CustomStringConvertible
Equatable
Comparable
Codable
```

- When you adopt a protocol, you must implement all required methods.

CustomStringConvertible

Printing with CustomStringConvertible

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

let number = 42
print(number) // 42

let boolean = false
print(boolean) // false
```

Printing with CustomStringConvertible

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

  init(color: String, size: Int, hasLaces: Bool) {
    …
  }
}

let myShoe = Shoe(color: "Black", size: 12, hasLaces: true)
print(myShoe) // __lldb_expr_1.Shoe
```

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

    init(color: String, size: Int, hasLaces: Bool) {
      …
    }




}
```

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

    init(color: String, size: Int, hasLaces: Bool) {
      …
    }

    var description: String {
      return "Shoe(color: \(color), size: \(size),
hasLaces: \(hasLaces))"
    }
}
```

```
let myShoe = Shoe(color: "Black", size: 12, hasLaces: true)
print(myShoe)
    // Shoe(color: Black, size: 12, hasLaces: true)
```

Equatable

Comparing information with Equatable

```
struct Employee {
  let firstName: String
  let lastName: String
  let jobTitle: String
  let phoneNumber: String
}

struct Company {
  let name: String
  let employees: [Employee]
}
```

Comparing information with Equatable

```
let currentEmployee = Session.currentEmployee
let selectedEmployee = Employee(firstName: "Adrien",
                                lastName: "Humilière",
                                jobTitle: "Engineer",
                                phoneNumber: "0612345678")

if currentEmployee == selectedEmployee {
  // Enable "Edit" button
}
```

Comparing information with Equatable

```
struct Employee: Equatable {
  let firstName: String
  let lastName: String
  let jobTitle: String
  let phoneNumber: String

  static func ==(lhs: Employee, rhs: Employee) -> Bool {
    // Equality logic
  }
}
```

Comparing information with Equatable

```
struct Employee: Equatable {
  let firstName: String
  let lastName: String
  let jobTitle: String
  let phoneNumber: String

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

Comparing information with Equatable

```
let currentEmployee = Employee(firstName: "Adrien",
                               lastName: "Humilière",
                               jobTitle: "Mobile engineer",
                               phoneNumber: "0612345678")
let selectedEmployee = Employee(firstName: "Adrien",
                                lastName: "Humilière",
                                jobTitle: "Support",
                                phoneNumber: "0612345678")

if currentEmployee == selectedEmployee {
  // Enable "Edit" button
}
```

Comparing information with Equatable

```
struct Employee: Equatable {
  let firstName: String
  let lastName: String
  let jobTitle: String
  let phoneNumber: String

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

Comparable

Sorting information with Comparable

```
let employee1 = Employee(firstName: "Ben",
                         lastName: "Atkins")
let employee2 = Employee(firstName: "Vera",
                         lastName: "Carr")
let employee3 = Employee(firstName: "Grant",
                         lastName: "Phelps")
let employee4 = Employee(firstName: "Sang",
                         lastName: "Han")

let employees = [employee1,
                 employee2,
                 employee3,
                 employee4]
```

```
struct Employee: Equatable, Comparable {
  let firstName: String
  let lastName: String
  let jobTitle: String
  let phoneNumber: String

  static func ==(lhs: Employee, rhs: Employee) -> Bool {
      return …
  }

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

```
let employees = [employee1, employee2, employee3,
                 employee4, employee5]

let sortedEmployees = employees.sorted(by: <)

for employee in sortedEmployees {
  print(employee)
}

// Employee(firstName: "Ben", lastName: "Atkins")
// Employee(firstName: "Vera", lastName: "Carr")
// Employee(firstName: "Sang", lastName: "Han")
// Employee(firstName: "Grant", lastName: "Phelps")
```

```
let employees = [employee1, employee2, employee3,
employee4, employee5]

let sortedEmployees = employees.sorted(by:>)

for employee in sortedEmployees {
  print(employee)
}

// Employee(firstName: "Grant", lastName: "Phelps")
// Employee(firstName: "Sang", lastName: "Han")
// Employee(firstName: "Vera", lastName: "Carr")
// Employee(firstName: "Ben", lastName: « Atkins")
```

Codable

Encoding and decoding objects with Codable

```
struct Employee: Equatable, Comparable, Codable {
    var firstName: String
    var lastName: String
    var jobTitle: String
    var phoneNumber: String

    …
}
```

Encoding and decoding objects with Codable

```
let ben = Employee(firstName: "Ben", lastName: "Atkins",
     jobTitle: "Front Desk", phoneNumber: "415-555-7767")

let jsonEncoder = JSONEncoder()
if let jsonData = try? jsonEncoder.encode(ben),
    let jsonString = String(data: jsonData,
encoding: .utf8) {
    print(jsonString)
}
```

```
{"firstName":"Ben","lastName":"Atkins","jobTitle":"Front
Desk","phoneNumber":"415-555-7767"}
```

Protocol creation

Creating a protocol

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

  func sayFullName()
}

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

Creating a protocol

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

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

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

Delegation

Delegation

Enables a class or structure to hand off responsibilities to an instance of another type

```
protocol ButtonDelegate {
  func userTappedButton(_ button: Button)
}

class GameController: ButtonDelegate {
  func userTappedButton(_ button: Button) {
    print("User tapped the \(button.title) button.")
  }
}
```

Delegation

```
class Button {
  let title: String
  var delegate: ButtonDelegate? // Add a delegate property

  init(title: String) {
    self.title = title
  }

  func tapped() {
    self.delegate?.userTappedButton(self)
          // If the delegate exists, call the delegate
          // function `userTappedButton` on the delegate
  }
}
```

Delegation

```
let startButton = Button(title: "Start Game")
let gameController = GameController()
startButton.delegate = gameController

startButton.tapped()
```

Saving data

Encoding and decoding with Codable

```
class Note: Codable {…}
```

→ Use an Encoder object to encode

→ Use a Decoder object to decode

Encoding and decoding with Codable

```
struct Note: Codable {
    let title: String
    let text: String
    let timestamp: Date
}

let newNote = Note(title: "Dry cleaning", text: "Pick up suit
                   from dry cleaners", timestamp: Date())

let propertyListEncoder = PropertyListEncoder()
if let encodedNotes = try? propertyListEncoder.encode(newNote) {
   …
}
```

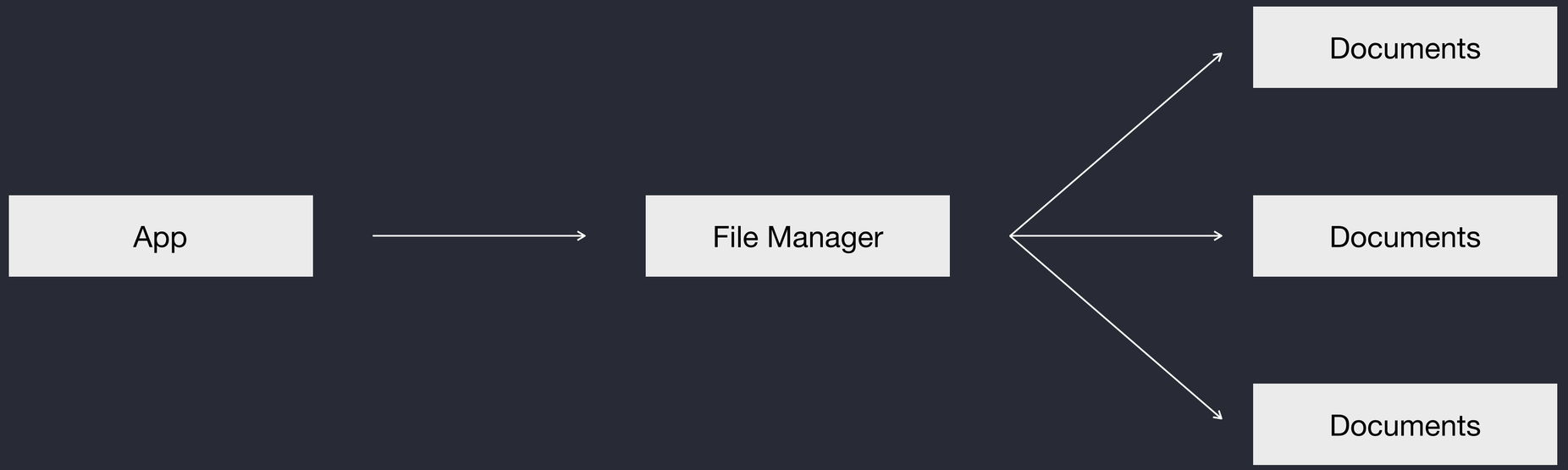
Encoding and decoding with Codable

```
let propertyListDecoder = PropertyListDecoder()
if let decodedNote = try? propertyListDecoder.decode(Note.self,
from: encodedNote) {
    …
}
```

App Sandbox

App Sandbox

Writing data to a file



Encoding and decoding with Codable

```
let documentsDirectory =
FileManager.default.urls(for: .documentDirectory,
                         in: .userDomainMask).first!
let archiveURL =
documentsDirectory.appendingPathComponent("appData")
                  .appendingPathExtension("plist")
```

Encoding and decoding with Codable

```
let propertyListEncoder = PropertyListEncoder()
let encodedData = try? propertyListEncoder.encode(data)

try? encodedData?.write(to: archiveURL,
                        options: .noFileProtection)
```

Encoding and decoding with Codable

```
let propertyListDecoder = PropertyListDecoder()
if let retrievedData = try? Data(contentsOf: archiveURL),
    let decodedNote = try? propertyListDecoder.decode(Note.self,
from: retrievedNoteData) {
  …
}
```

Your model objects should implement the Codable protocol. Reading and writing should happen in the model controller.

Archive in the correct app delegate life-cycle events. For example:

- When the app enters the background
- When the app is terminated