IS 543 Fall 2023

Mobile Platform Development

Project 1 Questions

Multithreading

Networking

Dr. Stephen W. Liddle
782 TNRB, office hours Tu 9:30-10:30am, Th 2-3pm
is543@byu.edu
801-422-8792 (rings on my cell)
Zoom: https://bit.ly/liddlezoom

Tuesday's Devotional

- Sister Kristin M. Yee

  Second Counselor, Relief Society General Presidency
  The Church of Jesus Christ of Latter-day Saints

  Our Covenant Relationship with God: A Wellspring of Relief
  1. There is relief in partnering with God
  2. There is relief in trusting in the Lord
  3. We can find the Savior's relief as we bring His relief to others
  4. We can find the Savior's relief in our personal relationships
  5. Our covenant relationship with God can be a wellspring of relief

Today

- Project 1 questions?

  I have seen several implementations where you track isSelected, isMatched, isMismatched as three different variables; it's cleaner to define an enum of four possible states for each card:

  notSelected, selected, matched, mismatched

- Multithreading

  `DispatchQueue`, `async`, `await`

- Networking

  You always want to call network APIs in the background

Multithreading

- Cardinal rule #1: Don't block my UI!
  - It is never okay for a mobile app's UI to be unresponsive to the user
  - But sometimes you need to do things that might take more than a few milliseconds
  - Most notably, accessing the network
  - Another example: doing some very CPU-intensive analysis like machine learning
  - How do we keep our UI responsive when these long-lived tasks are going on?
  - We run them on a different "thread of execution" than the UI is utilizing

Multithreading

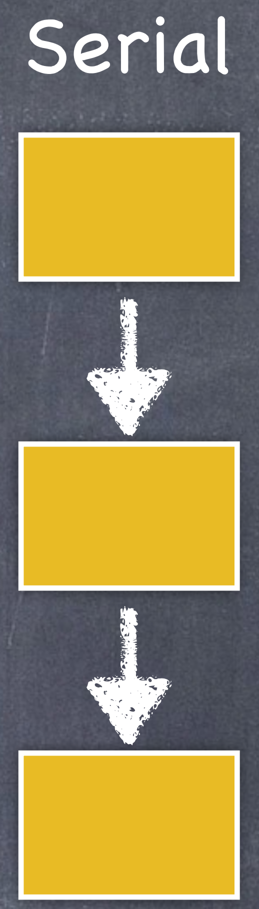
- Threads

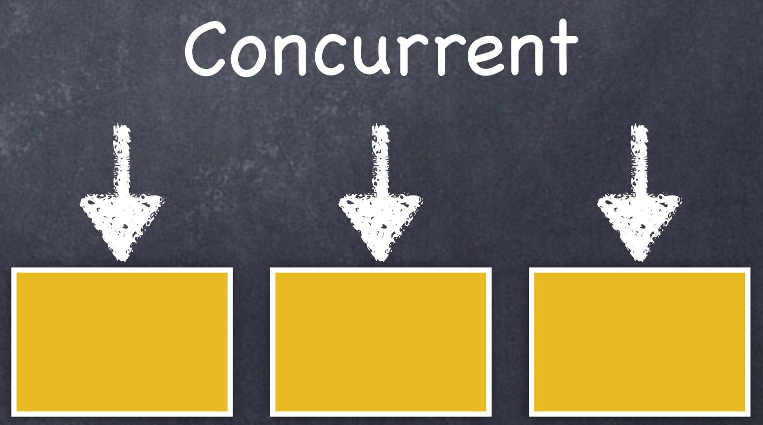
  Most modern operating systems (including iOS) let you specify a "thread of execution" to use
  These threads all appear to be executing their code simultaneously
  And they might be (in a multi-core—like iOS devices—or multi-processor computer)
  But they might just be switching back and forth between them really quickly
  With some of them getting higher priority to run than others
  You as a programmer can't tell the difference (and you don't want to know)

- Enter the queues…

Queue Up

- The iOS model is built on "dispatch queues"
  - A "queue" manages a list of tasks (functions, usually closures)
  - The queue executes its tasks on an associated thread
  - Queues are "serial" (perform one task at a time) or
    - "concurrent" (perform multiple tasks simultaneously)
- Main queue
  - This is a special serial queue that handles UI activity
  - **ALL** UI activity **MUST** occur on this queue
  - You should move any time-intensive non-UI activity OFF this queue
    - (You can have as many other queues/threads as you need)
  - Otherwise your UI may not be responsive





Concurrency in Swift 5.5

- Concurrency got a makeover in Swift 5.5
  - Functions can be marked `async`
  - You can use `await` to invoke an `async` function
  - You can assign asynchronously with `async let` statements, then `await` those assignments later
  - You can create a `Task` or `TaskGroup` to manage hierarchies of asynchronous operations and to control priorities, pause or cancel background operations, etc.
  - When multiple threads need to synchronize to avoid race conditions, use an `actor` type
- There are some common concurrency ideas:
  - Many programs are more effective when written to use multiple CPU threads
  - Multi-threaded applications need to be careful about accessing and modifying shared state
    - Updates to shared state need to be synchronized carefully to avoid "race conditions"
    - Multiple threads competing for the same resources can get into a state of "deadlock"
  - The UI thread (or "main" thread) is special; do NOT EVER block it with a long-running operation

An Example

- Consider the following example

  It's a bit complicated because of the completion handlers: closures passed along to be invoked when each step is completed

```
listPhotos(inGallery: "Summer Vacation") { photoNames in
    let sortedNames = photoNames.sorted()
    let name = sortedNames[0]
    downloadPhoto(named: name) { photo in
        show(photo)
    }
}
```

  We can infer that the definition of `listPhotos()` looks something like this:

```
func listPhotos(inGallery gallery: String,
                completionHandler: ([String]) -> Void) {
    // Do some work, then invoke completionHandler()
    let photoNames: [String] = ...
    completionHandler(photoNames)
}
```

async/await

- A function can be marked async (it runs in the background when called)

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}
```

- Use await to pause while the async function executes

```
let photoNames = await listPhotos(inGallery: "Summer Vacation”)
let sortedNames = photoNames.sorted()
let name = sortedNames[0]
let photo = await downloadPhoto(named: name)
show(photo)
```

- This is very much like JavaScript’s use of async/await

Using await

- Because await can suspend execution of the current thread, it can only be used in certain places:
  Code in the body of an async function, method, or property
  Code in the static main() method of a struct, class, or enum marked with @main
  Code in a "detached child task"

  Again, this is similar to the JavaScript async/await rules

Asynchronous Sequences

- You can write a sequence that conforms to AsyncSequence
- And then you can iterate with await:

```
import Foundation

let handle = FileHandle.standardInput
for try await line in handle.bytes.lines {
    print(line)
}
```

Controlling Concurrency

- What's the potential problem with this code?

```
let firstPhoto = await downloadPhoto(named: photoNames[0])
let secondPhoto = await downloadPhoto(named: photoNames[1])
let thirdPhoto = await downloadPhoto(named: photoNames[2])

let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

The downloads happen serially (in sequence), not in parallel

Parallel Execution

- Here's a potential solution to the previous situation with async let:

```
async let firstPhoto = downloadPhoto(named: photoNames[0])
async let secondPhoto = downloadPhoto(named: photoNames[1])
async let thirdPhoto = downloadPhoto(named: photoNames[2])

// Now the three downloads can happen in parallel
let photos = await [firstPhoto, secondPhoto, thirdPhoto]
// Do not continue on to show(photos) until all three downloads have finished
show(photos)
```

Synchronizing Access to Shared State

- The actor reference type is similar to class

  Only one Task can access an actor's mutable state at a time

```
actor TemperatureLogger {
     let label: String
     var measurements: [Int]
     private(set) var max: Int

     init(label: String, measurement: Int) {
          self.label = label
          self.measurements = [measurement]
          self.max = measurement
     }
}
```

  Use await to access properties and methods of an actor from outside the actor:

```
let logger = TemperatureLogger(label: "Outdoors", measurement: 25)
print(await logger.max) // Prints "25"
```

  Do not use await to access properties and methods of an actor from within the actor

Calling async from Regular Function

- When you need to use await in an ordinary non-async function, wrap it in a Task:

```
private func loadDocument() {
    Task {
        // You can use await here even though loadDocument() is not declared async
    }
}
```

Multithreading (Pre Swift-5.5)

- The primary API for multithreading is `DispatchQueue`
  Choose various attributes and submit a task synchronously or asynchronously
- Example:
  Perform a task on the main (UI) thread after waiting at least 1/10 second

```
DispatchQueue.main.asyncAfter(deadline: .now() + 0.1) {
    // Perform desired action on the main thread,
    // but after at least 0.1s has elapsed

    // ...
}
```

Multithreading (Pre Swift-5.5)

- Several helpful DispatchQueue properties and methods:

```
class var main: DispatchQueue      // Gives access to the main queue

// This gives access to the global non-UI queue
class func global(qos: DispatchQoS.QoSClass = default) -> DispatchQueue
    // Since qos has a default value, you can just call DispatchQueue.global()
    // to retrieve a queue that runs items in the background, off the main queue

sync(execute: DispatchWorkItem)    // Execute a task synchronously

async(execute: DispatchWorkItem)   // Execute a task asynchronously

asyncAfter(deadline: DispatchTime, execute: DispatchWorkItem)
```

There are others, but this set is what you'll usually use

Multithreading (Pre Swift-5.5)

- How to create a DispatchQueue if needed

  Often you can get away with using only DispatchQueue.main and DispatchQueue.global()

```
init(label: String,
     qos: DispatchQoS = default,
     attributes: DispatchQueue.Attributes = default,
     autoreleaseFrequency: DispatchQueue.AutoreleaseFrequency = default,
     target: DispatchQueue? = default)
```

  DispatchQoS represents quality of service (from high to low):

  userInteractive, userInitiated, default, utility, background

  attributes is mostly used just to specify a concurrent queue (serial is the default)

  I generally ignore the last two parameters and go with the defaults
  I doubt you'll need more than just the label and attributes parameters:

```
let serialQueue = DispatchQueue(label: "ImageLoader")
let concurrentQueue = DispatchQueue(label: "Writer", attributes: .concurrent)
```

Multithreading (Pre Swift-5.5)

- Common pattern: execute closure on background thread, update UI on main thread

```
let queue = DispatchQueue(label: "ImageLoader", attributes: .concurrent)

queue.async {
    // Load an image, scale it to the desired size, do other long-running stuff
    // Then when it’s time to update the UI:
    DispatchQueue.main.async {
        // Update the UI, e.g. load the Image data into your UIImageView
    }
}
```

Multithreading (Pre Swift-5.5)

- We're only scratching the surface here

  But that's mostly what you'll need for the kinds of apps you develop

  By the way, when you see the phrase Grand Central Dispatch (or GCD), they're talking about the iOS multithreading mechanisms we've discussed here

  There are also `OperationQueue`s and `Operations`

Multithreading (Pre Swift-5.5)

- iOS APIs that are multithreaded

  Often the iOS API will execute off the main thread

  Sometimes you get to pass a closure that will execute off the main thread too

  Remember to dispatch back to the main queue if you update the UI in that closure!

- Example: networking

```
if let url = URL(string: "https://scriptures.byu.edu/images/sci.png") {
    let task = URLSession.shared.downloadTask(with: url) { (url, response, error) in
        // If in this closure we want to update the UI using the
        // downloaded image, we MUST dispatch back to the main queue
    }
    task.resume()
}
```

Another Example

- Put long-running tasks in the background

  In Gospel Library, how long does it take to assemble/render HTML for a given book/chapter?

  This could take a relatively long time, causing UI to pause

  Things to do:

  - Read database
  - Gather records for each verse
  - Inject CSS, JavaScript, append verse text within an HTML page template

  Only then can we tell the web view to load the given string

```
DispatchQueue.global().async {
    // This code is now running in a background thread
    let html = ScriptureRenderer.shared.htmlForBookId(book.id, chapter: chapter)

    DispatchQueue.main.async {
        // This code is now running on the main thread
        webView.loadHTMLString(html, baseURL: nil)
    }
}
```

Modifying the UI on a background thread can lead to subtle bugs that are hard to detect. And it <u>might</u> work <u>sometimes</u>.

Another Example

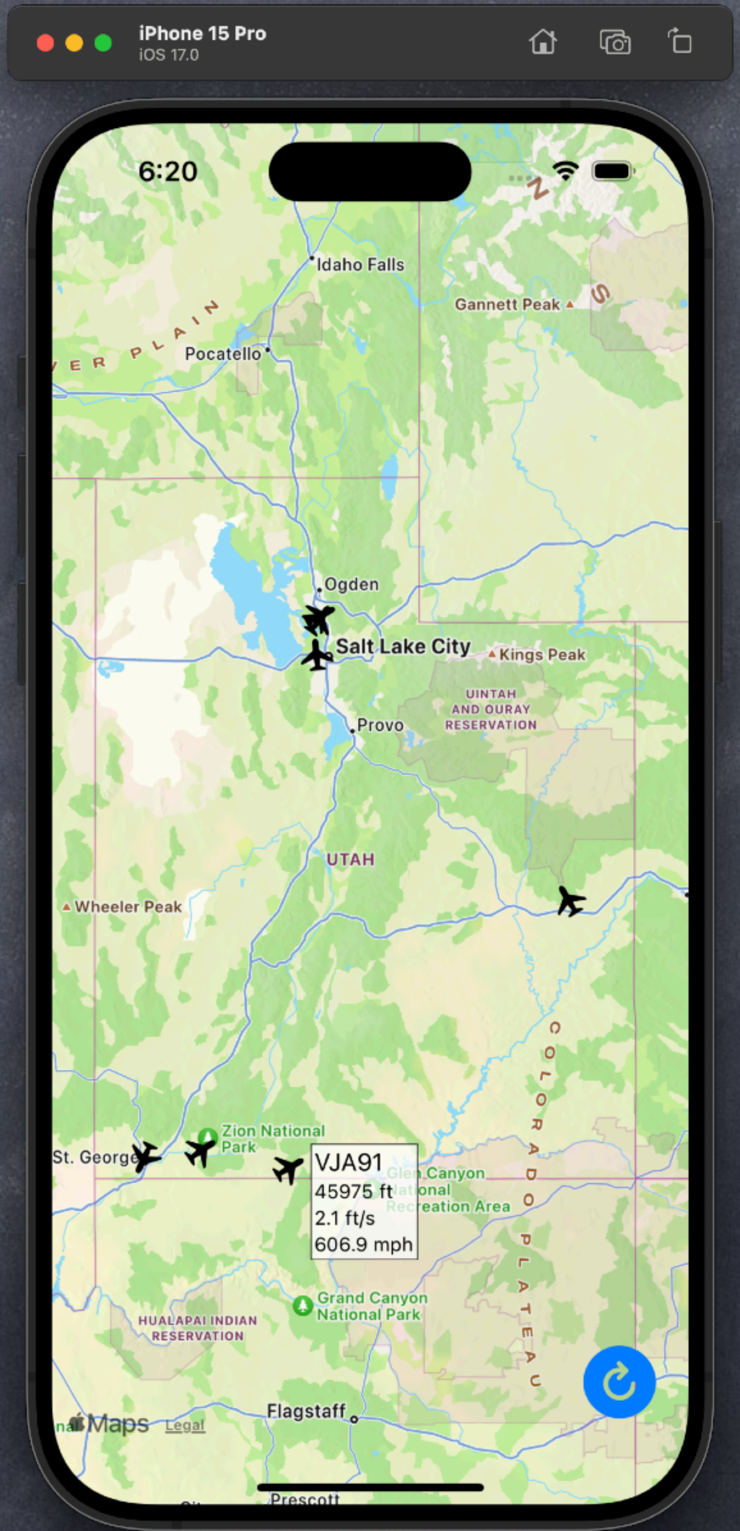
- Load an image from the network
  Display a placeholder immediately so UI operates smoothly
  Request images via network request
  When image comes back from network, if it's still visible on screen, replace placeholder with image

```
DispatchQueue.global().async {
    // Request image data here ...

    DispatchQueue.main.async {
        // On successful return of data, check whether image is still visible
        // and if so, update it here using the data loaded from the network ...
    }
}
```

OpenSky Utah Demo

- A new app
  - Inquire of the OpenSky network API
  - Display information about airplanes in Utah



Coming Up…

- Continue learning SwiftUI techniques and patterns

  We need to learn how to read/write data in the filesystem and DBMS

  There are a number of additional SwiftUI views, techniques, patterns yet to explore

  And we need to learn something about UIKit too