iOS DESIGN PATTERNS

HANDS-ON CHALLENGES

iOS Design Patterns

Joshua Greene

Notice of Rights

Notice of Liability

This challenge and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express of implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Table of Contents: Overview

Table of Contents: Extended

5 MulticastClosureDelegate - Challenge

By Joshua Greene

At this point, you have a working `MulticastClosureDelegate`.

However, is it thread safe? For example, what happens if you add several closure pairs on a concurrent background thread?

You haven't actually implemented thread safety yet, so it's *probably* not thread safe... *cough cough*, it's not... but you'll first need to prove it! ;]

Challenge Setup

Before trying to "fix" a problem, it's a good idea to confirm one actually exists.

Add the following to the top of the playground:

```
import PlaygroundSupport
PlaygroundPage.current.needsIndefiniteExecution = true
```

This causes the playground to continue running even after all top-level code is executed. This is required because you'll be dispatching to a background queue, which will be executed asynchronously.

Next, add the following at the **end of the file**:

```
// MARK: - Multithreading
print("-- Multithreading --")
multicastDelegate.mapTable.removeAllObjects()

let count = 3
for _ in 0 ..< 3 {
  DispatchQueue.global(qos: .background).async {
    multicastDelegate.addClosurePair(for: objectKey, success: {

    }, failure: {
    })
    print("count: \(multicastDelegate.count)")
  }
}
```

If `MulticastClosureDelegate` was thread safe, you should see print statements counting from 1 to 3 **in the console**... however, this always prints 1!

This confirms that `MulticastClosureDelegate` is *not* thread safe.

> **Note**: well technically, since this is dispatched asynchronously, you would expect to see print statements counting from "1 to 3", "all 3s", "1, 3, 3" or even "2, 3, 3."
>
> Any of these would be acceptable, as each would indicate the closure pairs were all added.

Challenge

You *could* make `MulticastClosureDelegate` thread safe by implementing a "one-off" solution, by using as a mutex lock, GCD serial queue, etc. However, there's a better way: create a reusable thread-safe wrapper class!

Open the project navigator, go to **Sources** and open **SynchronizedValue.swift**.

Use this file as a starting point for creating a readers-write lock. If you're not familiar with it, see the Wikipedia article on it readers-write lock:

https://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock

After implementing **SynchronizedValue**, update **MulticastClosureDelegate** using it to add thread safety.

When you're done, add **SynchronizedValue.swift** to the **RWClean** project, and update **MulticastClosureDelegate.swift**.

Warning: this challenge is hard!

If you want a hint, scroll to the next page.

Hint

Here's one solution to implementing a readers-write lock using GCD.

Replace **SynchronizedValue.swift** with the following:

```
import Foundation

public class SynchronizedValue<ValueType: Any> {

  // MARK: - Properties
  private let queue = DispatchQueue(
    label: "SynchronizedValue(\(type(of: ValueType.self))",
    attributes: .concurrent)

  private var backingValue: ValueType

  // MARK: - Object Lifecycle
  public init(_ value: ValueType) {
    self.backingValue = value
  }

  // MARK: - Safe Accessors
  public func get() -> ValueType {
    var value: ValueType!
    queue.sync { value = backingValue }
    return value
  }

  public func set(_ closure: (inout ValueType) -> ()) {
    queue.sync(flags: .barrier) { closure(&backingValue) }
  }

  // MARK: - Unsafe Accessors
  public var unsafeValue: ValueType {
    get { return backingValue }
    set { backingValue = newValue }
  }

  public func setUnsafeValue(_ closure: (inout ValueType) -> ()) {
    closure(&backingValue)
  }
}
```

You can now use this to add thread safety to **MulticastClosureDelegate**.

Challenge Solution

There are multiple solutions to this challenge, but here's mine using GCD.

If you haven't already, read the **Hint** above and replace **SynchronizedValue.swift** as instructed.

Replace the **MulticastClosureDelegate** playground file with the following:

```
import PlaygroundSupport
PlaygroundPage.current.needsIndefiniteExecution = true

import Foundation

// MARK: - MulticastClosureDelegate
public class MulticastClosureDelegate<Success, Failure> {

  // MARK: - Callback
  class Callback {
    let queue: DispatchQueue
    let success: Success
    let failure: Failure
    init(queue: DispatchQueue, success: Success, failure: Failure) {
      self.queue = queue
      self.success = success
      self.failure = failure
    }
  }

  // MARK: - Instance Properties
  internal var mapTable = SynchronizedValue(
    NSMapTable<AnyObject, NSMutableArray>.weakToStrongObjects()
  )

  public var count: Int {
    return getCallbacks(removeAfter: false).count
  }

  // MARK: - Instance Methods
  public func addClosurePair(for objectKey: AnyObject,
                             queue: DispatchQueue = .main,
                             success: Success,
                             failure: Failure) {

    mapTable.set { mapTable in
      let callBack = Callback(queue: queue, success: success, failure:
failure)
      let array = mapTable.object(forKey: objectKey) ?? NSMutableArray()
      array.add(callBack)
      mapTable.setObject(array, forKey: objectKey)
    }
  }

  public func getSuccessTuples(removeAfter: Bool = true) -> [(Success,
DispatchQueue)] {
    return getCallbacks(removeAfter: removeAfter).map {
      return ($0.success, $0.queue)
    }
  }

  public func getFailureTuples(removeAfter: Bool = true) -> [(Failure,
DispatchQueue)] {
    return getCallbacks(removeAfter: removeAfter).map {
      return ($0.failure, $0.queue)
    }
  }

  fileprivate func getCallbacks(removeAfter: Bool = true) -> [Callback] {

    var callBacks: [Callback]!
    mapTable.set { mapTable in
      let objects = mapTable.keyEnumerator().allObjects as [AnyObject]
      callBacks = objects.reduce([]) { (combinedArray, objectKey) in
        let array = mapTable.object(forKey: objectKey)! as! [Callback]
        return combinedArray + array
      }
      guard removeAfter else { return }
      objects.forEach { mapTable.removeObject(forKey: $0) }
    }
    return callBacks
  }
}

// MARK: - Testing
typealias Success = () -> Void
typealias Failure = () -> Void

let multicastDelegate = MulticastClosureDelegate<Success, Failure>()
let delegate = NSObject()

multicastDelegate.addClosurePair(for: delegate, success: {
  print("Success")
}, failure: {
  print("Failure")
})

let callback = multicastDelegate.getCallbacks(removeAfter: false).first!
let success = callback.success
success()

let (successClosure, successQueue) =
multicastDelegate.getSuccessTuples(removeAfter: false).first!
successClosure()

let (failureClosure, failureQueue) =
multicastDelegate.getFailureTuples(removeAfter: false).first!
failureClosure()

print(multicastDelegate.count)


// MARK: - Multithreading
print("-- Multithreading --")
multicastDelegate.mapTable.unsafeValue.removeAllObjects()
```

```
let count = 3
for _ in 0 ..< 3 {
  DispatchQueue.global(qos: .background).async {
    multicastDelegate.addClosurePair(for: delegate, success: {

    }, failure: {

    })
    print("count: \(multicastDelegate.count)")
  }
}
```

Rememeber to also add **SynchronizedValue.swift** and update **MulticastClosureDelegate.swift** in **RWClean.xcodeproj**.