iOS DeCal : Lecture 9

Delegates, Protocols, Advanced Swift (GCD, Closures, Structs/Enums)

April 11, 2017

Announcements - 3/21

Lab this week - project work day

Attend section with your group (either lab room)

Attendance required

Project 2-2 and Lab 6 due tonight (11:59)

Make sure to submit Lab 6 to Gradescope (even if you got checked off)

Announcements - 3/21

What's Left

- 1 more lab assignment
- 2 more lectures
- **Final Presentations on 5/5 at 10am (Friday of Dead Week)**
  - Attendance is mandatory for a Pass

Overview : Today's Lecture

- Protocols
- Delegates
- Advanced Swift
  - Structs and Enums
  - GCD

Protocols

Protocols : Review

Protocol: a generic outline or skeleton
Set of rules that delegates must follow
Can be above a class declaration, or in own .swift file

Classes that follow such rules are said to "*conform to the protocol*"

Classes can conform to any number of protocols

Protocols : Review

```
protocol Vehicle {
    var numWheels: Int { get }
    func getSpeed() -> Double
    mutating func refuel(percentage: Double)
}
```

For a class to conform to this protocol `SomeDelegate`, it must implement the `sendBack` and `updateModel` methods

Protocols : Review

```
class SmartCar: Car, Vehicle {
    var numWheels = 4
    func getSpeed(){
        return self.speed
    }
    func refuel(percentage: Double) {
        self.gas += self.maxGas * percentage
    }
}
```

`SmartCar` is a subclass of `Car` and conforms to the protocol `Vehicle`

Protocols : Why Though?

Protocols are a way to express an API more concisely

Instead of forcing a caller to pass in a specific class, (i.e. Car) an API can let the caller pass in whatever class they want…as long as it fulfills certain desired requirements (is a Vehicle)

```
func getRide(v: Vehicle) {
        return v
    }
```

Protocols don't do any implementation

This means that they have zero storage associated with them, all storage is provided by implementing classes.

Protocols : They are Types!

You may have seen delegates passed in through functions (like Vehicle on the last slide). In general, Protocols can be used just like types. This includes using them as:

* Variable types
* Function parameters
* Function return values

…and pretty much anywhere else you see types used.

Protocols : Declaration

```
protocol SomeProtocol : InheritedProtocol1,
                        InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double,
                 arg2: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Protocols : Decleration

```
protocol SomeProtocol : InheritedProtocol1,
                        InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double,
                 arg2: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

A protocol can inherit from any number of other protocols, so long as it includes anything that those protocols require

Protocols : Decleration

```
protocol SomeProtocol : InheritedProtocol1,
                        InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double,
                 arg2: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Any variable that is required must be specified as either

`{ get }` or `{ get set }`

`{ get }:` The variable only needs to be gettable

`{ get set }:` The variable must be settable as well as gettable

Protocols : Decleration

```
protocol SomeProtocol : InheritedProtocol1,
                        InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double,
                 arg2: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Normal function requirement, must specify the return type if there is one

Protocols : Decleration

```
protocol SomeProtocol : InheritedProtocol1,
                        InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double,
                 arg2: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Adding the `mutating` keyword means that this method must be capable of mutating the class/instance that is conforming

(Note that you do NOT need to put the `mutating` keyword in front of the function when you implement it in

Protocols : Decleration

```
protocol SomeProtocol : InheritedProtocol1,
                        InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double,
                 arg2: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

You can even require that the implementor have a specific initializer
Note that an implementing class needs to mark this initializer as `required`.

Delegates

What is Delegation? : Review

Design Pattern

Allows objects to interact with each other without creating dependencies

via **Protocols**, **Delegates**, and **Data Sources**

Delegation : Main use in MVC

Delegation serves as a blind connection from the view to the controller.

The view assumes that it has some minion who is capable of performing certain actions, and uses this minion despite being blind to who that minion actually is.

Delegation : Process

Delegation like this is accomplished by following the following steps:

1. A view declares a protocol (i.e. what the controller will be doing for it
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self, thus declaring that it is the delegate.
6. That controller actually implements the protocol, so that it can do what it is told to.

Delegation : Process

With that, the View is now hooked up to a Controller!

The fun thing is, the View doesn't actually know anything about the Controller (besides that it is capable of implementing the View's protocol), so the View remains generic/reusable.

Delegation Example

Example : RandomGenerator

```swift
protocol RandomNumberGenerator {
    func random() -> Double
}

class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom =
            ((lastRandom * a + c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
```

Example : RandomGenerator

```
protocol RandomNumberGenerator {
    func random() -> Double
}

class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom =
            ((lastRandom * a + c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
```

Stating that we conform to a protocol

Example : RandomGenerator

```
protocol RandomNumberGenerator {
    func random() -> Double
}

class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom =
            ((lastRandom * a + c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
```

Implementing the required function

Example : Using Protocols as Types

```swift
protocol RandomNumberGenerator {
    func random() -> Double
}

class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

Example : Using Protocols as Types

```
protocol RandomNumberGenerator {
    func random() -> Double
}

class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

Declaring a variable of type `RandomNumberGenerator`, and declaring a function that takes in a `RandomNumberGenerator`

Example : Using Protocols as Types

```
protocol RandomNumberGenerator {
    func random() -> Double
}

class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

Since we know generator is a `RandomNumberGenerator,` we can assume that it has some `random()` function that returns a Double

Example : Delegation

```
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
              didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```swift
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
              didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```swift
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```swift
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```swift
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```
protocol DiceGame {
    var dice: Dice { get }
    var delegate: DiceGameDelegate? { get set }
    func play()
}

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```
protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
              didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}

class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    init(game: DiceGame) {
        var game = game
        game.delegate = self
    }
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```
protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
              didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}

class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    init(game: DiceGame) {
        var game = game
        game.delegate = self
    }
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```swift
protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
              didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}

class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    init(game: DiceGame) {
        var game = game
        game.delegate = self
    }
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. **That controller sets the view's delegate property (from #2) to self,**
6. That controller actually implements the protocol, so that it can do what it is told to.

Example : Delegation

```
protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
              didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}

class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    init(game: DiceGame) {
        var game = game
        game.delegate = self
    }
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

1. A view declares a protocol
2. That view's API has a `delegate` property with the protocol's type
3. The view uses the delegate property to do things that the view can't normally do, assuming that there is some controller actually doing that work
4. Some controller declares that it conforms to the protocol from #1
5. That controller sets the view's delegate property (from #2) to self,
6. That controller actually implements the protocol, so that it can do what it is told to.

5 Minute Break

Structs and Enums

Multithreading

Threads : Review

Thread - a single unit of execution in a process

In applications that support *multithreading* (i.e. iOS applications), you can delegate different sections of code to be handled by different threads

- However, in iOS, you will deal with queues rather than individual threads - more on that later!

Threads : Review

Why use multiple threads? CPU utilization

Example: Say we have an app that makes a network request to read something from a database

If we only have one thread, the CPU will be idle while waiting for the network response.

A better idea is to use multiple threads, so we can do other work (computations, UI updates, etc.) while we wait

Multithreading : Overview

Problem

We want our app to run as fast as possible, but we have lengthy operations (calculations / data traversals / network requests, etc.)

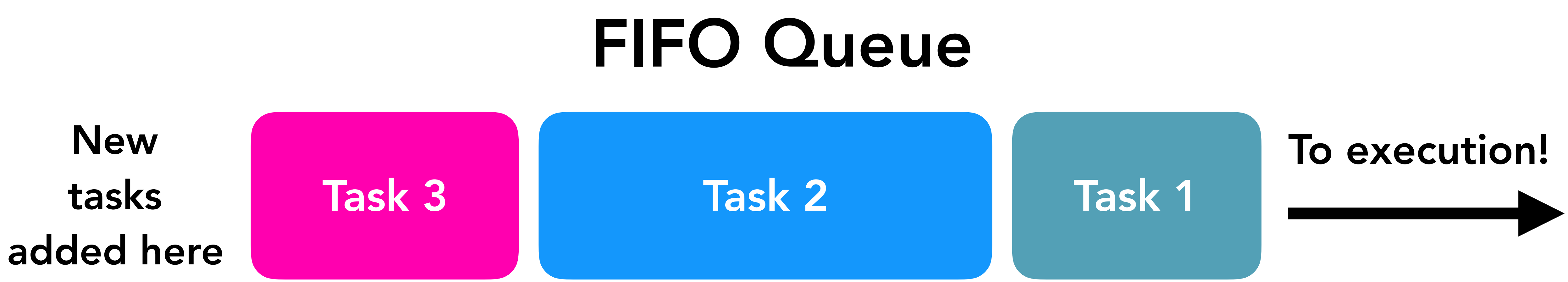
Solution

Put time-consuming computation on lower priority threads

Put short operations that we need done right away on higher priority threads (i.e. UI updates)

Multithreading : Queues

- In iOS, interaction with threads is done via queues
  - **Tasks** (functions / closures / blocks of code) are added to a queue
  - Once the task is popped off, it will be executed by the thread associated with that queue



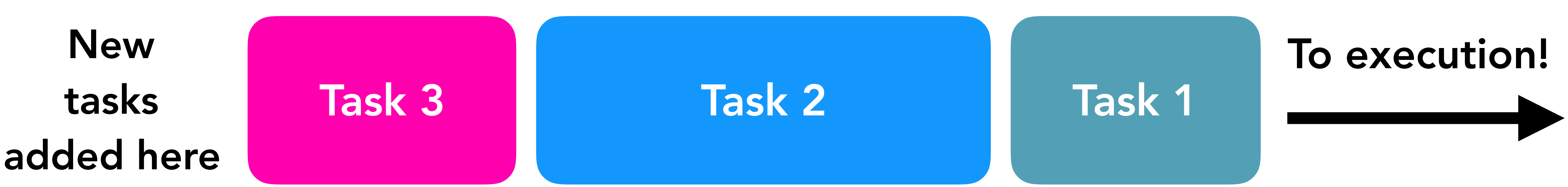
Multithreading : Queues

In iOS, interaction with threads is queues

- **Tasks** (functions / closures / blocks of code) are added to a queue
- Once the task is popped off, it will be executed by the thread associated with that queue

> **Side Note**
> Tasks are sometimes referred to as “work items”

FIFO Queue



Multithreading : Queues

Different Queues will have different priorities

Queues : Two Different Types

Serial Queues

- Executes a single task from the queue at a time
- Tasks are handled in the order that they were inserted (task 2 must wait for task 1 to complete before execution)

Concurrent Queues

- Allows for multiple tasks to be executed in parallel

Queues : Serial

In serial queues, newly added task must wait until their predecessors complete.

Example: task 2 must wait until task 1 is completed before it begins execution

Queues : Concurrent (background)

In concurrent queues, tasks will still begin execution in FIFO order, but do not have to "wait" for other tasks to finish

Queues : Concurrent (background)

In concurrent queues, tasks will still begin execution in FIFO order, but do not have to "wait" for other tasks to finish

Example: task 2 must wait to start until task 1 begins execution but task 2 ends up finishing execution before function 1

Multithreading : Main Queue

- Special Queue for iOS - **the Main Queue**
  - **Serial queue** (only one task from this will be handled at any given time)
  - **Reserved for UI operations**
    - This is done since the UI of your app should always be very responsive (i.e. tapping a button should instantly send feedback to the user).
  - By having a queue reserved for these sorts of operations, we can be sure that UI operations will be responsive

Dispatch Queues : Usage

Grand Central Dispatch - Apple technology to manage queues of tasks in your application

So how do you actually execute code asynchronously in Xcode?

1. Define your task (what you want to be done in another code) by placing it in a function or closure
2. Add your task to one of the default global queues, or a queue created by you

Dispatch Queues (serial) : Creation

To create a queue, create an instance of `DispatchQueue` using a unique label

```
let queue = DispatchQueue(label: “myqueue")
```

To then execute a task on that queue, use the instance methods `sync` and `async`

```
queue.sync {
    print("hello world")
    // your closure code here!
}
```

Dispatch Queues (serial) : Creation

To create a queue, create an instance of `DispatchQueue` using a unique label

```
let queue = DispatchQueue(label: “myqueue")
```

To then execute a task on that queue, use the instance methods `sync` and `async`

```
func sayHello() {
    print("hello world")
}
queue.sync(execute: sayHello)
```

Quality of Service

Remember - often times we want some queues to be executed with a higher priority than other queues.

How do we specify the priority of a queue?

Set it's "Quality of Service" (QoS)

QoS enum cases

(in descending order of priority)

```
userInteractive
userInitiated
default
utility
background
unspecified
```

QoS Class	Type of work and focus of QoS	Duration of work to be performed
User-interactive	Work that is interacting with the user, such as operating on the main thread, refreshing the user interface, or performing animations. If the work doesn't happen quickly, the user interface may appear frozen. Focuses on responsiveness and performance.	Work is virtually instantaneous.
User-initiated	Work that the user has initiated and requires immediate results, such as opening a saved document or performing an action when the user clicks something in the user interface. The work is required in order to continue user interaction. Focuses on responsiveness and performance.	Work is nearly instantaneous, such as a few seconds or less.
Utility	Work that may take some time to complete and doesn't require an immediate result, such as downloading or importing data. Utility tasks typically have a progress bar that is visible to the user. Focuses on providing a balance between responsiveness, performance, and energy efficiency.	Work takes a few seconds to a few minutes.
Background	Work that operates in the background and isn't visible to the user, such as indexing, synchronizing, and backups. Focuses on energy efficiency.	Work takes significant time, such as minutes or hours.

QoS Cases (Apple Developer)

Quality of Service (serial) : Priorities

To create a queue with a QoS, create an instance of `DispatchQueue` using a unique label **and** QoS value

```
let queue = DispatchQueue(label: "myQ1",
                qos: DispatchQoS.userInitiated)

let queue2 = DispatchQueue(label: “myQ2",
                     qos: DispatchQoS.utility)
```

Concurrent Queues

So far we have only been dealing with serial queues (all tasks of a single queue have been executed and completed one after an other)

What if we don't care about the order that the tasks in our queue are run?

Concurrent Queues

So far we have only been dealing with serial queues (all tasks of a single queue have been executed and completed one after an other)

What if we don't care about the order that the tasks in our queue are run?

Create a concurrent queue!

```
let queue = DispatchQueue(label: "myQ",
                          qos: .utility,
                          attributes: .concurrent)
```

Concurrent Queues

So far we have only been dealing with serial queues (all tasks of a single queue have been executed and completed one after an other)

What if we don't care about the order that the tasks in our queue are run?

Simply add the "concurrent" attribute

Create a concurrent queue!

```
let queue = DispatchQueue(label: "myQ",
                          qos: .utility,
                          attributes: .concurrent)
```

Global Queues

Though you can create your own queues, it may not always be necessary to do so

Instead of initializing your own queue with an identifier, you can access predefined **global queues** with specific QoS values

Global Queues

Though you can create your own queues, it may not always be necessary to do so

Instead of initializing your own queue with an identifier, you can access predefined **global queues** with specific QoS values

```
// returns a queue with default QoS
let globalQ1 = DispatchQueue.global()

// returns a queue with QoS = qos
let globalQ2 = DispatchQueue.global(qos: .utility)
```

Main Queues

Often times, we will want to delegate part of our code to the main queue within a function / closure that is placed in a background queue

You can access the main queue as follows:

```
DispatchQueue.main.async {
    // do something on the main queue
}
```

Queues : Real Life Example

Recall this code we went over in our Networking lecture

```
func loadImage() {
  let url = URL(string:"https://instagram.com/img.jpg")
  let session = URLSession.shared
  let task = session.dataTask(with: url!,
              completionHandler: {
                (data, response, error) -> Void in
                if error == nil {
                    let img = UIImage.init(data: data!)
                    self.imageView.image = img
                }
  })
  task.resume()
}
```

Queues : Real Life Example

```
2017-04-11 17:56:13.141 Queue Example[24484:3466140] This
application is modifying the autolayout engine from a background
thread after the engine was accessed from the main thread. This
can lead to engine corruption and weird crashes.
 Stack:(
    0   CoreFoundation                      0x000000010eacfb0b
__exceptionPreprocess + 171
    1   libobjc.A.dylib                     0x000000010bd76141
objc_exception_throw + 48
    2   CoreFoundation                      0x000000010eb38625 +
[NSException raise:format:] + 197
    3   Foundation                          0x000000010ba6f17b
_AssertAutolayoutOnAllowedThreadsOnly + 105
    4   Foundation                          0x000000010ba6ef0f -
[NSISEngine _optimizeWithoutRebuilding] + 61
    5   Foundation                          0x000000010b89e7e6 -
[NSISEngine optimize] + 108
    6   Foundation                          0x000000010ba6cef4 -
[NSISEngine performPendingChangeNotifications] + 84
```

All Output ◇ Filter

Running the code from the previous slide as is will print out the following warning in your console

Queues : Real Life Example

```
2017-04-11 17:56:13.141 Queue Example[24484:3466140] This
application is modifying the autolayout engine from a background
thread after the engine was accessed from the main thread. This
can lead to engine corruption and weird crashes.
 Stack:(
    0   CoreFoundation                      0x000000010eacfb0b
__exceptionPreprocess + 171
    1   libobjc.A.dylib                     0x000000010bd76141
objc_exception_throw + 48
    2   CoreFoundation                      0x000000010eb38625 +
[NSException raise:format:] + 197
    3   Foundation                          0x000000010ba6f17b
_AssertAutolayoutOnAllowedThreadsOnly + 105
    4   Foundation                          0x000000010ba6ef0f -
[NSISEngine _optimizeWithoutRebuilding] + 61
    5   Foundation                          0x000000010b89e7e6 -
[NSISEngine optimize] + 108
    6   Foundation                          0x000000010ba6cef4 -
[NSISEngine performPendingChangeNotifications] + 84
```

All Output ⌃ Filter

To fix the code - we need to make sure we update our UI on the main thread

Queues : Real Life Example

Recall this code we went over in our Networking lecture

```
func loadImage() {
  let url = URL(string:"https://instagram.com/img.jpg")
  let session = URLSession.shared
  let task = session.dataTask(with: url!,
              completionHandler: {
                (data, response, error) -> Void in
                if error == nil {
                    let img = UIImage.init(data: data!)
                    self.imageView.image = img
                }
  })
  task.resume()
}
```

The completion handler of dataTask will be called on a background thread

Queues : Real Life Example

Recall this code we went over in our Networking lecture

```
func loadImage() {
  let url = URL(string:"https://instagram.com/img.jpg")
  let session = URLSession.shared
  let task = session.dataTask(with: url!,
              completionHandler: {
                (data, response, error) -> Void in
                if error == nil {
                    let img = UIImage.init(data: data!)
                    self.imageView.image = img
                }
  })
  task.resume()
}
```

Inside this completion handler, we update our UI

Queues : Real Life Example

Recall this code we went over in our Networking lecture

```
func loadImage() {
  let url = URL(string:"https://instagram.com/img.jpg")
  let session = URLSession.shared
  let task = session.dataTask(with: url!,
              completionHandler: {
                (data, response, error) -> Void in
                if error == nil {
                 let img = UIImage.init(data: data!)
                 DispatchQueue.main.async {
                    self.imageView.image = img
                   }
                }
  })
  task.resume()
```

To fix - update UI on main thread

Project 2 Part 2 and Lab 6

Due **Tonight** at 11:59pm

Next Lab : Project Work Day