lecture 11

Advanced Swift

cs198-001 : spring 2018

Attendances

Will be posting sheet on Piazza this week

Check that we've accounted for all of your attendances!

Make a **private piazza post** if there is an error

Remember - we allow 3 unexcused absences

Lab this week

Project work day: **attendance still mandatory**

You should be *at least* 60% done with your final app

Use this time to work on your project, ask for design advice, work with teammates, etc.

Final Project Submission

Final project due Wednesday of RRR week

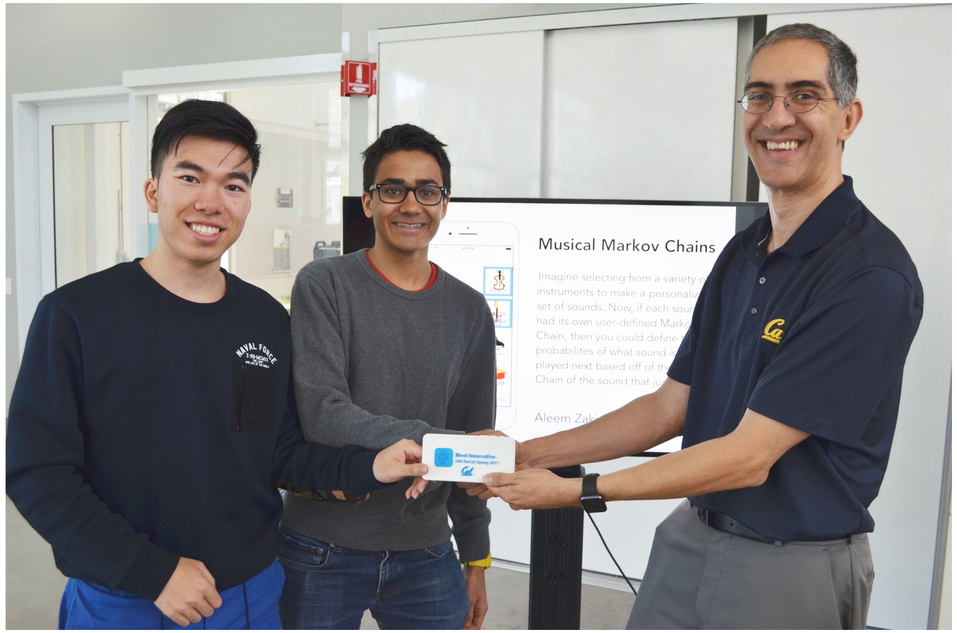
Submit via google form (on website)

- github repo link
- video walkthrough of your app
- app logo
- app description
- app screenshot

Graded on implementation of app (35% of total grade)

If chosen to present, we will notify you on Wednesday (by the end of the day)

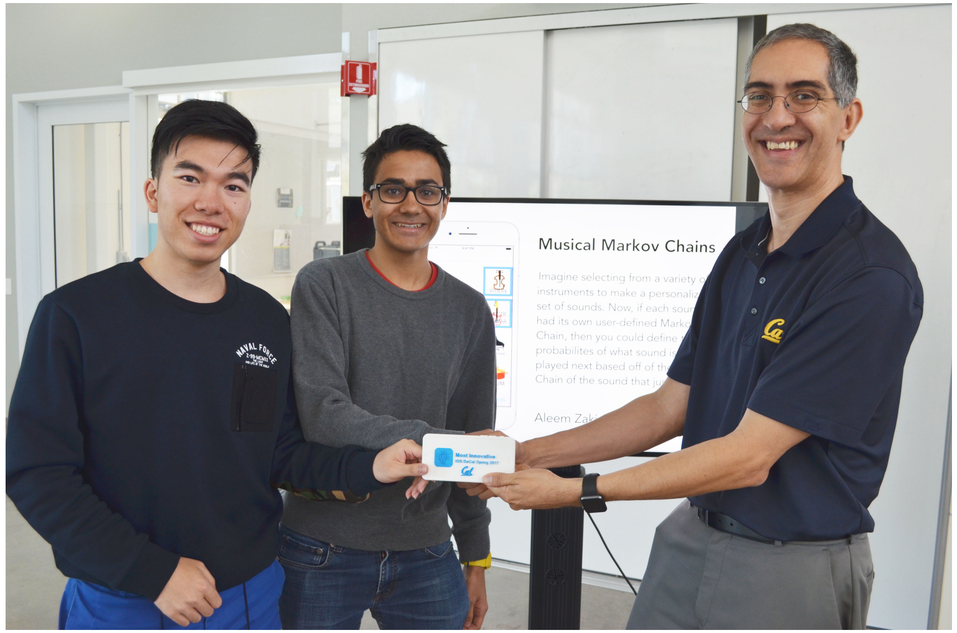
Final Project Showcase



Friday, May 4th at 4-6pm in the HP Auditorium

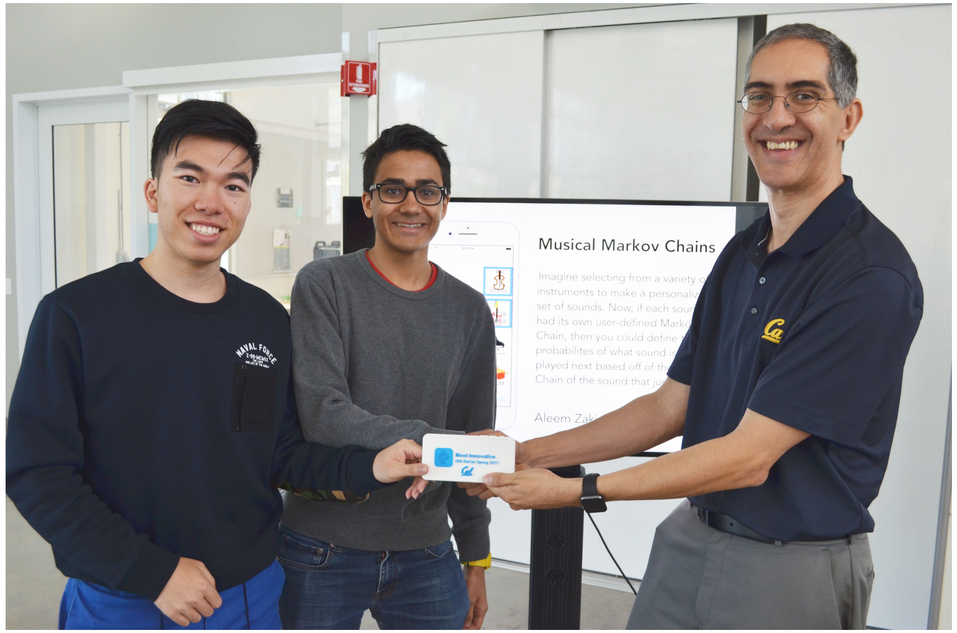
attendance mandatory for Pass

Final Project Showcase



Student app presentations, feedback / Q&A, and awards

Final Project Showcase



Awards

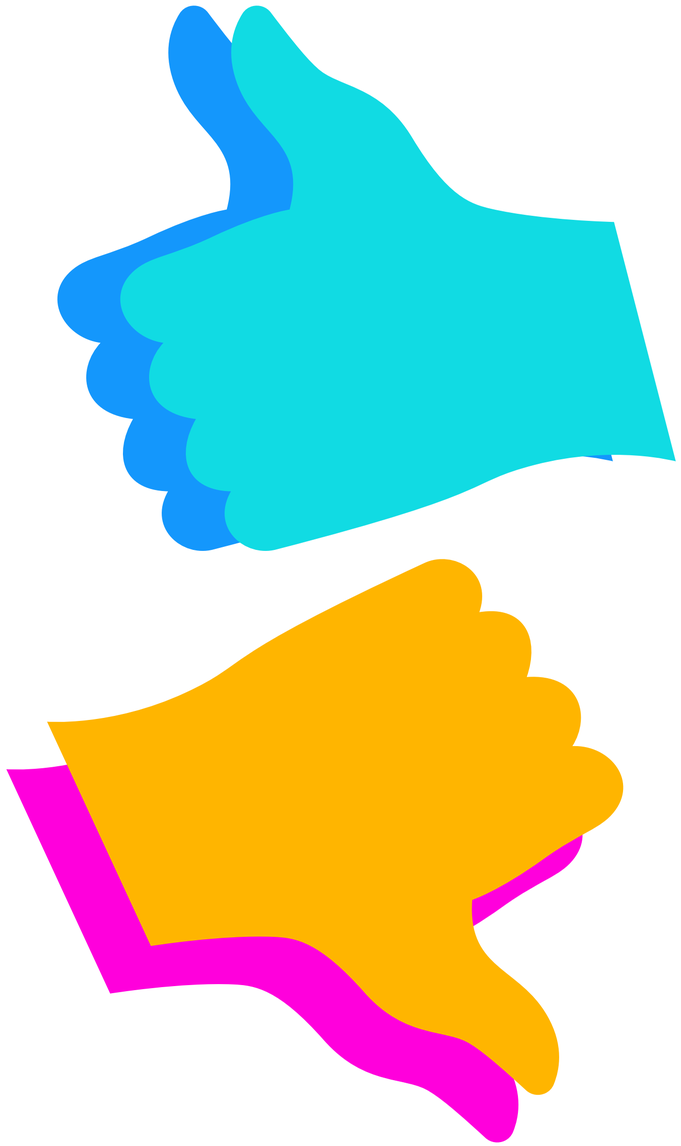
People’s Choice • Most Innovative • Social Impact • Best Design • Most Technical

Apply to TA!

CS 399 - 1 unit

apply by May 9

https://goo.gl/forms/ovrl9dMLQf1MOZQ23



Course Evaluation

Loved the decal? Hated the decal? Let us know!

https://goo.gl/forms/hKMBIEMOzawa7RID2

Overview: Today's Lecture

- Property observers
- Computed properties
- Enumerations
- Concurrency
- Swift/Objective-C Interoperability
- Advice from the old instructors...

Property Observers

Property Observers

- Observe and respond to changes in a stored property's value.

- Called every time a property's value is set.

- Two observers:
  - `willSet`: called before value is stored.
  - `didSet`: called after new value is stored.

Declaration

```
class StepCounter {

    var totalSteps: Int = 0 {
        willSet {
            print("About to set totalSteps to
                  \(newValue)")
        }

        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps -
                      oldValue) steps")
            }
        }
    }
}
```

Named Parameter Declaration

```
class StepCounter {

    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to
                  \(newTotalSteps)")
        }

        didSet(oldTotalSteps) {
            if totalSteps > oldTotalSteps  {
                print("Added \(totalSteps -
                      oldTotalSteps) steps")
            }
        }
    }
}
```

Computed Properties

Computed Properties

- Stored Properties vs. Computed Properties
  - Computed properties don't actually store a value, but instead compute one on access

- They provide a getter (and optional setter) to retrieve/set other properties indirectly.

- Can be defined in classes, structures, and enumerations.

Read-Only Declaration

```
struct Task {
    var totalTime: Double
    var completedTime: Double

    var remainingTime: Double {
        return totalTime - completedTime
    }
}
```

Declaration

```
struct Task {
    var totalTime: Double
    var completedTime: Double

    var remainingTime: Double {
        get {
            return totalTime - completedTime
        }

        set {
            completedTime = totalTime - newValue
        }
    }
}
```

Named Parameter Declaration

```
struct Task {
    var totalTime: Double
    var completedTime: Double

    var remainingTime: Double {
        get {
            return totalTime - completedTime
        }

        set(newRemainingTime) {
            completedTime =
                    totalTime - newRemainingTime
        }
    }
}
```

Enumerations

Enumerations

- Define a common `Type` for a group of related values.
- Enables working with those values in a type-safe way.
- Assign related names to set of values (called *raw* values), which can be of type:
  - String
  - Character
  - Integer
  - Floating-point type
- First-class type: support computed properties, instance methods, protocol conformance.

Declaration

```
// Method 1
enum CompassPoint {
    case north
    case south
    case east
    case west
}

// Method 2
enum CompassPoint {
    case north, south, east, west
}
```

Note: Unlike C and Obj-C, enum cases are not implicitly assigned a default integer value when created.*

Raw Values

- Enum cases can have default values (called *raw* values), which are all of the same type.

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

- Initializing from a raw value:

```
let tabChar = ASCIIControlCharacter(rawValue: "\t")
```

Implicitly Assigned Raw Values

- Integer-backed enums *do* have implicit default values, and count up sequentially expected

```
enum NumbersWithoutThree: Int {
  case zero, one, two, four = 4, five, six
}
```

- `.zero` has an *implicit* raw value of 0
- `.four` has an *explicit* raw value of 4
- `.five` has an *implicit* raw value of 5, and so on.

Implicitly Assigned Raw Values

- String-backed enums also have *implicit* default values, which are the text of that case's name.

```
enum Planet: String {
  case mercury, venus, earth, mars, jupiter, saturn,
    uranus, neptune
}


let earthString = Planet.earth.rawValue


print(earthString) // "earth"
```

Switch Statements

- Must be exhaustive when considering an enum's cases.

```
let directionToHead: CompassPoint = .north

switch directionToHead {
case .north:
    print("Watch out for reindeer")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the sun sets")
}
```

Switch Statements

- But if we only care for one case... default

```
let almaMater: Universities = .berkeley

switch almaMater {
case .berkeley:
    print("Go Bears!")
default:
    print("You suck.")
}
```

check in

Optionals

- The `Optional` type is an enumeration with two cases:
  - `Optional.none`
    - equivalent to the `nil` literal.
  - `Optional.some(Wrapped)`
    - stores a wrapped value.

Optionals

- What does this print?

```
var aString: Optional<String>

switch aString {
case .none:
    print("String is nil")
case .some(let unwrappedValue):
    print("String is '\(unwrappedValue)'")
}
```

Optionals

- What does this print?

```
var aString: Optional<String>

switch aString {
case .none:
    print("String is nil")
case .some(let unwrappedValue):
    print("String is '\(unwrappedValue)'")
}

// "String is nil"
```

Optionals

- What does this print?

```
var aString = Optional.some("not nil!")

switch aString {
case .none:
    print("String is nil")
case .some(let unwrappedValue):
    print("String is '\(unwrappedValue)'")
}
```

Optionals

- What does this print?

```
var aString = Optional.some("not nil!")

switch aString {
case .none:
    print("String is nil")
case .some(let unwrappedValue):
    print("String is '\(unwrappedValue)'")
}

// "String is not nil!"
```

Concurrency

Threads : Review

- **Thread** - a single unit of execution in a process.
- Allow us to delegate different sections of code to be handled by different threads.

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.)

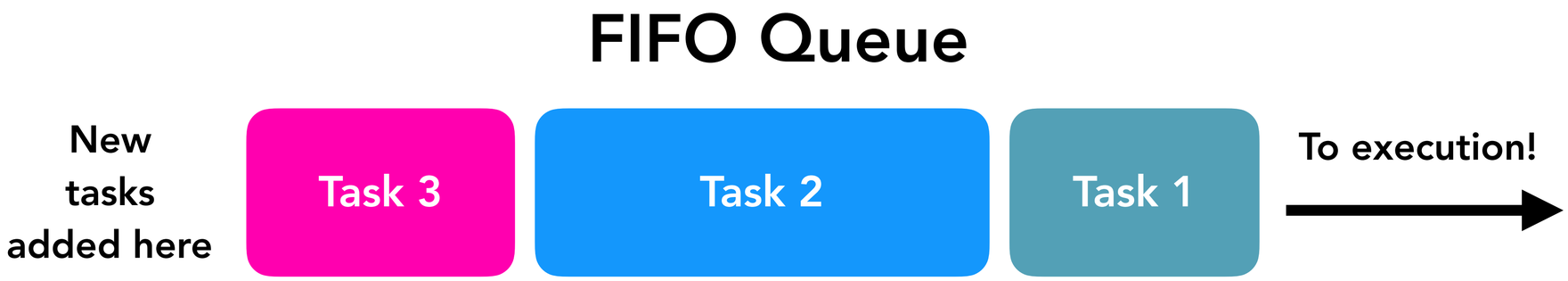
Solution:

- 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



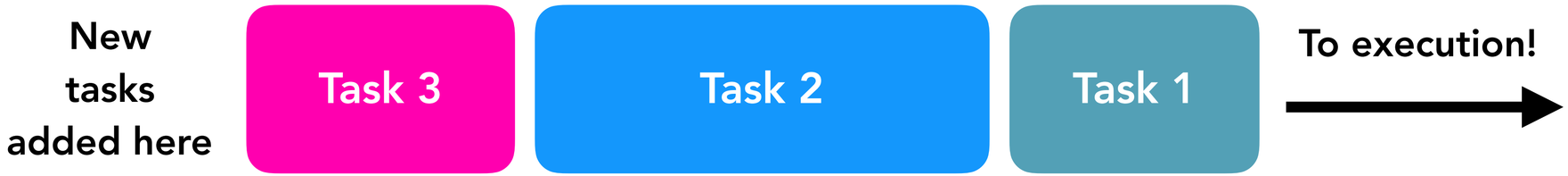
Multithreading : Que

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

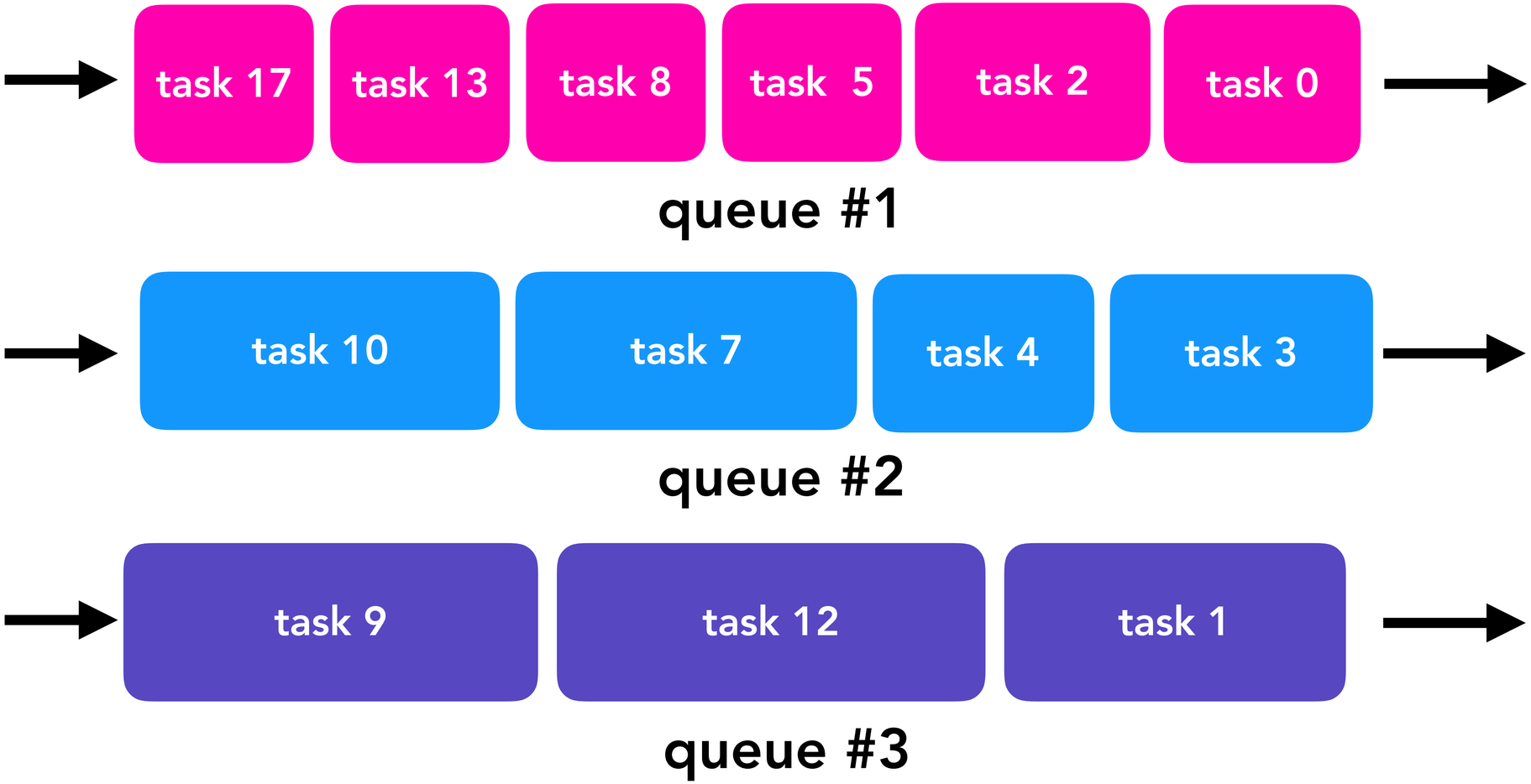
In iOS, interaction with threads is c
queues

- **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

FIFO Queue



Multithreading : Queues



Different Queues will have different priorities

Queues : Two Different Types

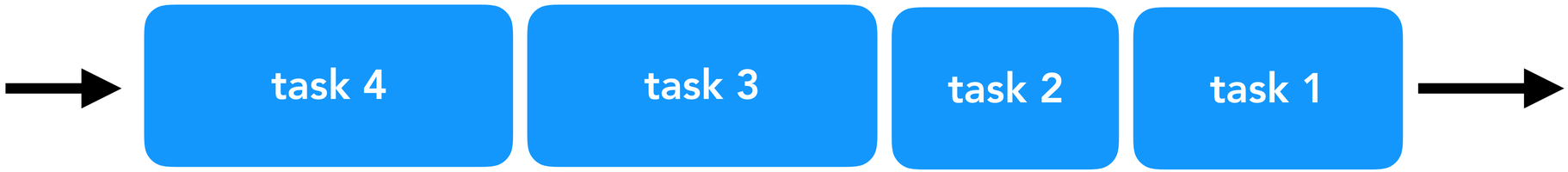
Serial Queues:

- 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

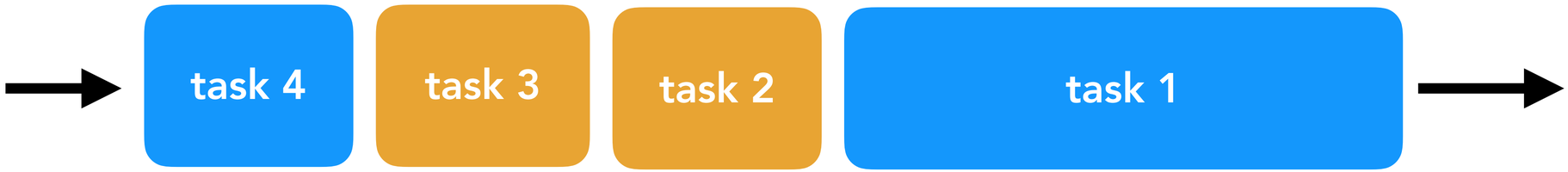
Queues : Serial



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

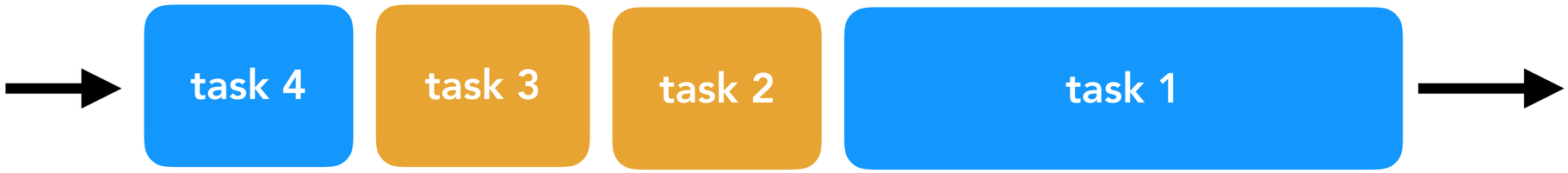
Example: task 2 must wait until task 1 is completed before it begins execution

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 - manages 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 q = DispatchQueue(label: "myQ1",
                      qos: DispatchQoS.userInitiated)

let q2 = 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 q = 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?

Create a concurrent queue!

Simply add the "concurrent" attribute

```
let q = 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 q1 = DispatchQueue.global()

// returns a queue with QoS = qos
let q2 = DispatchQueue.global(qos: .utility)
```

Main Queues

Accessing the main queue:

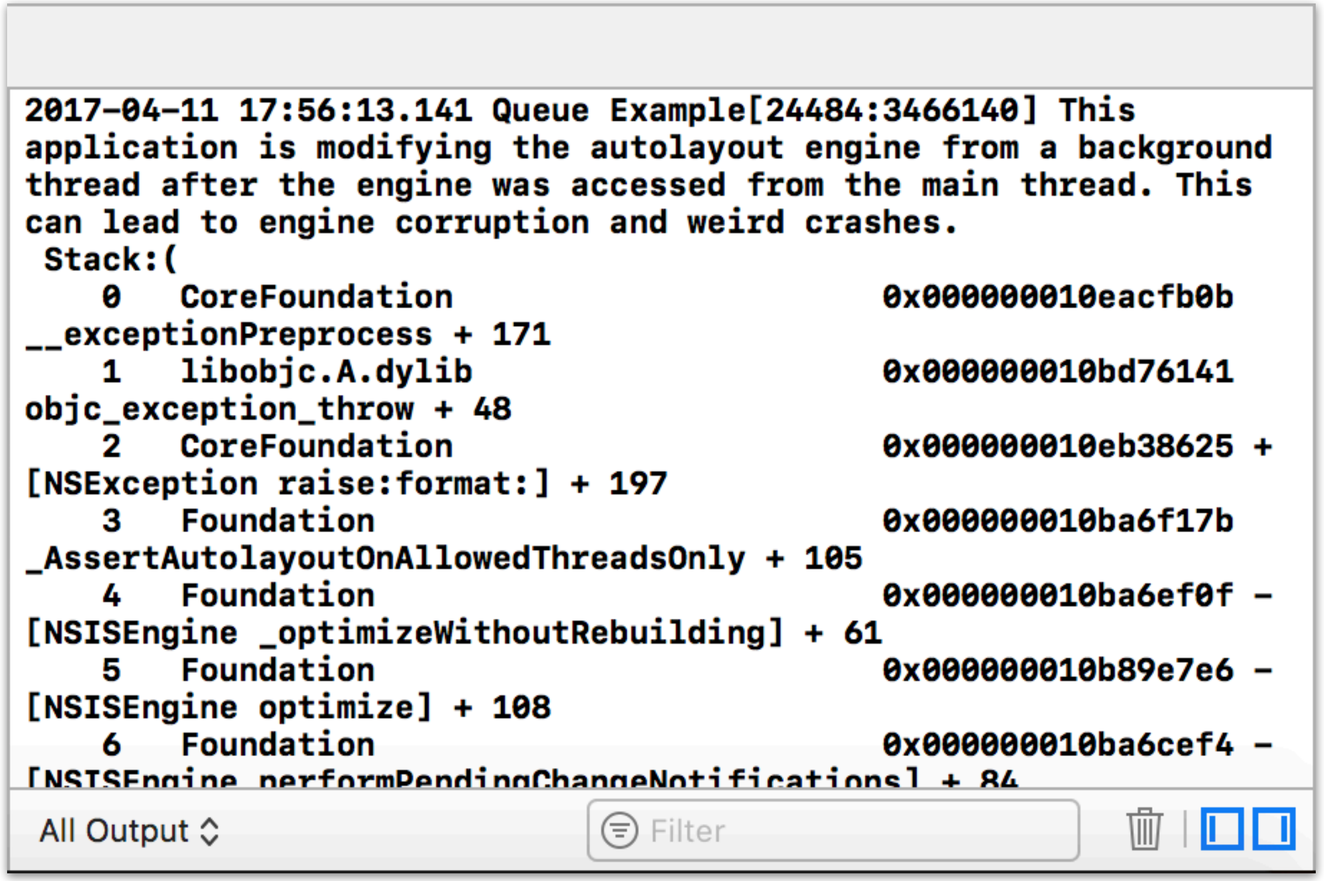
```
DispatchQueue.main.async {
    // do something on the main queue
}
```

Queues : Real Life Example

Recall this code we went over in our Networking lecture

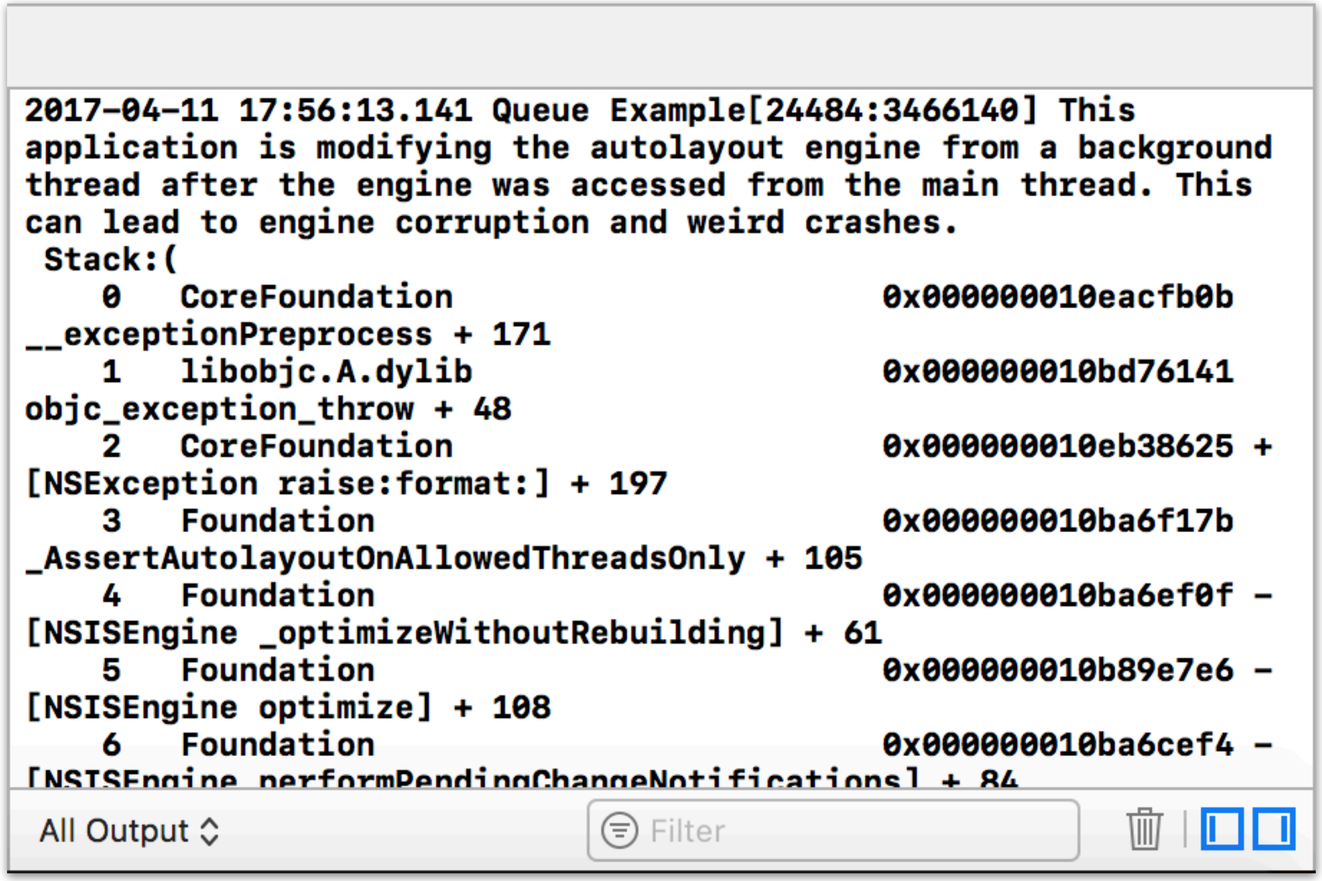
```
func loadImage() {
  let url = URL(string:"https://instagram.com/img.jpg")
  let session = URLSession.shared
  let task = session.dataTask(with: url!,
                completionHandler: {
                  (data, response, error) -> Void in
                  if error == nil {
                   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

Queues : Real Life Example



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

Queues : Real Life Example



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

Swift/Objective-C Interoperability

Getting Swift and Obj-C to play nicely...

	Import into Swift	Import into Objective-C
Swift code	No import statement	#import <ProductName/ ProductModuleName-Swift.h>
Objective-C code	No import statement Use Bridging Header (app) or Umbrella Header (framework)	#import "Header.h"

Getting Swift and Obj-C to play nicely...

	Import into Swift	Import into Objective-C
Swift code	No import statement	#import <ProductName/ ProductModuleName-Swift.h>
Objective-C code	No import statement Use **Bridging Header** (app) or Umbrella Header (framework)	#import "Header.h"

Bridging Header

- Expose Objective-C code to Swift code within the same target

- Create a file like "MyApp-Bridging-Header.h"

```
#import "MyViewController.h"
#import "MyClass.h"
#import "MyCustomCell.h"
```

- Use as you would expect in Swift

```
let vc = MyViewController()
```

Enum Compatibility

- Due to its foundations in C, enums in Objective-C are just integers
- So, to be Objective-C compatible, Swift enums must be Integer-backed

```
@objc public enum CollegeYear: Int {
   case freshman
   case sophomore
   case junior
   case senior
 }
```

Enum Compatibility

- Due to its foundations in C, enums in Objective-C are just integers
- Swift enum's use of computed properties, instance methods, etc. is not supported
- Workaround:

```
@objc public class CollegeYearObjC: NSObject {
   public static func fromRawValue(_ rawValue: Int) ->
      CollegeYear {
      return CollegeYear(rawValue: rawValue)
   }
}
```

- Use:

```
CollegeYear *year = [CollegeYearObjC fromRawValue: 3];
```

Getting Swift and Obj-C to play nicely...

	Import into Swift	Import into Objective-C
Swift code	No import statement	#import <ProductName/ ProductModuleName-Swift.h>
Objective-C code	No import statement Use Bridging Header (app) or **Umbrella Header** (framework)	#import "Header.h"

Nullability Annotations

	Meaning	Swift Translation	Notation
null_unspectified	Not sure	Implicitly Unwrapped Optional	String!
nullable	Can be nil	Optional	String?
nonnull	Won't be nil	Non-Optional	String

Nullability Side Effects

- null_unspecified is the **default** annotation
  - Properties/params in Obj-C will be translated to IUO's in Swift
    - IUO's cause **runtime crashes** if nil
- workaround by either doing the following
  - annotate the nullability correctly
  - use guards