- When code executes the path of execution is singular. (What does this mean?)
- Threads allows the OS and our apps to have multiple separate paths of execution.
- Every line of code runs on *some thread*.
- You will almost never want to deal *directly* with threads in iOS because dealing with threads directly is incredibly complex.
- Instead you will use either the C API *GCD* (Grand Central Dispatch) or NSOperationQueue to interact *indirectly* with threads by using queues.
- Queues are not the same thing as threads.
- Queues are abstractions that employ threads.
- Queues are used to execute *blocks/closures.*
- Threads can be *serial* or *concurrent*.
- A serial queue executes blocks FIFO, and it waits until completion before the next block is executed. Just like an orderly line up at a movie theatre.
- A concurrent queue dequeues blocks in a FIFO manner but does so on different threads often concurrently/simultaneously (depending on “resources”).
- Treat the order of concurrent queue completion as essentially random. It’s probabalistic which makes it extremely difficult to reason about.
- Main Queue:
  - The main queue is the interface queue.
  - The main queue uses the applications main thread.
  - The main queue is where all interface interaction

events are received (like touch events).
- When your app interacts with the user interface it *must* do so on the main queue.
- The main queue is not *thread safe*. (What does “thread safe” mean?)
- Example of tapping a button
  - Touch event arrives on main thread.
  - Button’s action runs on the main thread.
  - No other interface events can be handled while this is happening.
  - Once the code finishes running the main thread is again ready to receive events.
- The main thread is *blocked* while your code runs on it, and this can make the interface unresponsive if your code is a time hog.
- Mostly you should simply execute your code on the main queue and not worry about it.
- But you need to worry about background queues in at least 2 situations
  - 1. Your code does something that might take a long time. (Even if you have a fast network and you’re not doing an expensive request why should you never ever not do this on the main queue?)
  - 2. Some framework/library you’re using calls you back on a background queue. E.g. NSURLSession calls you back on a background queue. So, you must get a reference to the main queue in order to update the view. Note: only come back to the main queue at the moment you need to update the interface, not before. Doing so before may yield unpredictable results.

NSOperationQueue Vs GCD

- Apple offers two separate API's for concurrency: GCD and NSOperationQueue.
- NSOperationQueue is an OO wrapper around GCD.
- For simple concurrency you will most likely use GCD. (It's a pretty clean API).
- Use NSOperationQueue if you need to do more complex concurrency.
- For instance, if you need to create dependencies between concurrent operations. (I will demo this shortly).
- You need your concurrent operations to be objects. (Why might you need to do this?)
- You need to cancel operations, or you need other kinds of control, like scheduling.
- You need to be notified about the state of operations. (NSOperationQueue uses KVO)
- I'm mostly going to focus on GCD, since this will be what you will use most often, but we will also look at some simple examples of NSOperationQueue.

Creating Or Getting Queues in GCD:

- Note: To see everything GCD can do do a search in **Dash** with "*dispatch_*" .
- 2 ways to get a queue:
  - 1) User created.
  - 2) System created.
- You are rarely going to use the first option (can't

control the priority), but for completeness I include it.

- User created background queues look like this:

```
1
2  // C Definition of User Created Q's
3  dispatch_queue_t dispatch_queue_create( const char
   *label, dispatch_queue_attr_t attr);
4
5  // Creation Objc
6  dispatch_queue_t userCreatedBackgroundQ1 =
   dispatch_queue_create("com.lighthouse.threading.1",
   DISPATCH_QUEUE_CONCURRENT);
7
8  dispatch_queue_t userCreatedBackgroundQ2 =
   dispatch_queue_create("com.lighthouse.threading.2",
   DISPATCH_QUEUE_SERIAL);
9
10 // Creation Swift
11
12 let userCreatedBackgroundQ1: dispatch_queue_t  =
   dispatch_queue_create("com.lighthouse.threading.1",
   DISPATCH_QUEUE_CONCURRENT);
13
14 let userCreatedBackgroundQ2: dispatch_queue_t =
   dispatch_queue_create("com.lighthouse.threading.2",
   DISPATCH_QUEUE_SERIAL);
15
16
```

- System created queues (the one you will mostly use) look like this:

```
1
2  // C Definition
3  dispatch_queue_t dispatch_get_global_queue( long
   identifier, unsigned long flags);
4
5  // Objc Creation
6  dispatch_queue_t sysCreatedbackgroundQ1 =
   dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE
   , 0);
7
8  // Swift Creation
9  let sysCreatedbackgroundQ1: dispatch_queue_t =
   dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE
   , 0);
10
11
```

- dispatch_queue_t is the return type (_t is the way C indicates this is a type).
- dispatch_get_global_queue() takes 2 parameters.
- Always pass 0 for the second param (it's reserved for future use)
- The first param is an enum value that specifies the relative priority, called *Quality Of Service (QOS).*
- There are 4 possible options:

```
QOS_CLASS_USER_INTERACTIVE // highest priority
QOS_CLASS_USER_INITIATED // high priority, used when
user initiates an action
QOS_CLASS_UTILITY // used for long running tasks
QOS_CLASS_BACKGROUND // used when user doesn't need
```

result

- You can get ahold of the main queue like this:

```

// Getting Ref To Main Queue Objc
dispatch_queue_t mainQ = dispatch_get_main_queue();

// Getting Ref To Main Queue Swift
let mainQ: dispatch_queue_t =
dispatch_get_main_queue();


```

Adding Tasks To Queues in GCD:

```

// C Definition
void dispatch_async( dispatch_queue_t queue,
dispatch_block_t block);

// Objc Example
dispatch_queue_t backgroundQ1 =
dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE
, 0);

dispatch_async(backgroundQ1, ^{
    printf("1\n");
});
printf("2\n");
```

```swift

// Swift Example

let backgroundQ1: dispatch_queue_t = dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE, 0);

dispatch_async(backgroundQ1){
    print("1")
}
print("2");

```

- `dispatch_async` takes 2 paramaters: the queue and the block to dispatch
- `dispatch_async` returns *immediately*
- What does the above code print and why?
- `dispatch_sync` does not return until the block has run
- You will rarely use `dispatch_sync` . (If you need to handle dependencies use NSOperationQueue instead).

- `dispatch_once` is used to execute a block only once during the lifetime of the app.
- `dispatch_once` is used in the singleton pattern in Objc.

```swift

// C Definition
void dispatch_once( dispatch_once_t *predicate, dispatch_block_t block);

// Objc Creation
```

```
 6 static dispatch_once_t onceToken = 0;
 7
 8 // This waits for completion
 9 dispatch_once(&onceToken, {
10     printf("this will only execute once")
11 });
12
13 dispatch_once(&onceToken, {
14     printf("this will never execute!")
15 })
16
17 // Swift Creation
18
19 var onceToken: dispatch_once_t = 0;
20
21 // This waits for completion
22 dispatch_once(&onceToken, {
23     print("this will only execute once")
24 });
25
26 dispatch_once(&onceToken, {
27     print("this will never execute!")
28 })
29
30 // NB. xCode includes a code snippet with this.
31
```

a

- Open the project **ConcurrencySwift** and look at the test target.