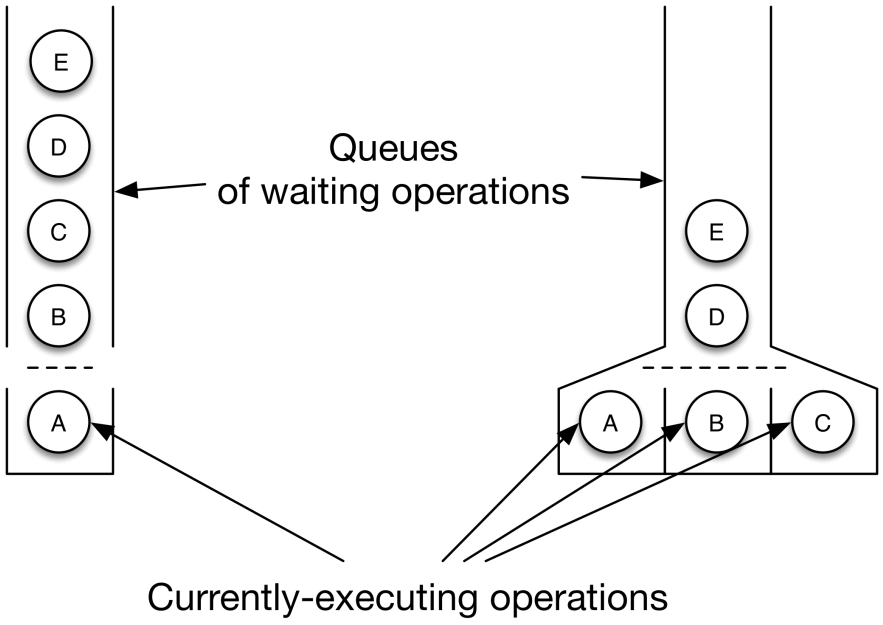
Multithreaded Arch with Operations

Queues, not threads



OperationQueue: Working in the background

```
class ViewController: UIViewController {

    private let bgQueue = OperationQueue()

    func doWork() {
        bgQueue.addOperation {
            // ...do some intensive work
        }
    }
}
```

OperationQueue: UI on the main queue

```
class ViewController: UIViewController {

    private let bgQueue = OperationQueue()

    func doWork() {
        bgQueue.addOperation {
            // ...do some background work
            OperationQueue.main.addOperation {
                // ... update UI on the main queue
                someLabel.text = text
            }
        }
    }
}
```

OperationQueue

```
// Queue Management
func addOperation(_: Operation)
func addOperation(_: () -> Void)

var operations: [Operation] { get } // going away
func cancelAllOperations()

// Suspending
var isSuspended: Bool

// Controlling concurrency
var maxConcurrentOperationCount: Int
```

Operation

```
// Operation Control
func start()
func main()
func cancel()

// Operation State
var isReady: Bool { get }
var isExecuting: Bool { get }
var isFinished: Bool { get }
var isCancelled: Bool { get }
```

Operation Dependencies

```
func addDependency(_: Operation)
func removeDependency(_: Operation)

var dependencies: [Operation] { get }
```

OperationQueue will execute the highest priority operation not dependent on any others

Custom Operation: Complex Lifetime

- set up your "execution environment"
- override -start, -isExecuting, -isFinished, and -isConcurrent
- You are responsible for storing and KVOing execution attributes
- Set your queue's max operation count appropriately
- Set .isConcurrent to true
- just... don't.

Custom Operation: Simple Lifetime

- OperationQueue will make a thread for you
- override main()
- execution attribute updates are automatic

Supporting cancellation of work

Check **isCancelled** from within **main()**

Operation Priority

```
var queuePriority: Operation.QueuePriority { get set }

/* enum Operation.QueuePriority : Int {
    case veryLow
    case low
    case normal
    case high
    case veryHigh
} */
```

Operation QOS

```
var qualityOfService: QualityOfService { get set }

/* public enum QualityOfService : Int {
    case userInteractive
    case userInitiated
    case utility
    case background
    case `default`
} */
```

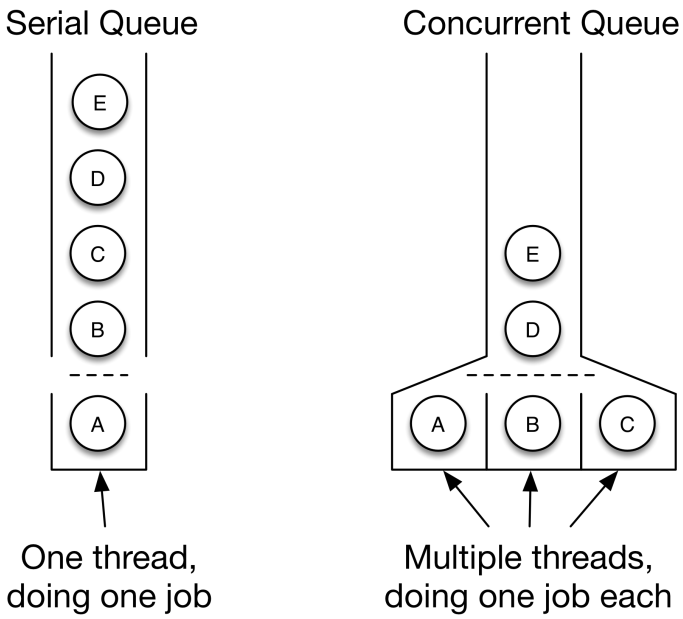
Exercise

- Abstract multithreaded work from Photorama into Operation subclass
- Add cancellation and re-prioritization support

Grand Central Dispatch

Introduction

Queues v. Threads



Enqueuing work

```
DispatchQueue.global(qos: .utility).async {
    // background work
    DispatchQueue.main.async {
        // update UI
    }
}
```

Enqueuing work

```
let myQ = DispatchQueue(label: "com.bnr.BackgroundWorker",
                        qos: .userInitiated,
                        attributes: .concurrent,
                        autoreleaseFrequency: .workItem)

myQ.asyncAfter(deadline: .now() + 0.2)) {
    // do something after a delay of 0.2 seconds
}
myQ.async(flags: .barrier) {
    // do something on its own in a concurrent queue
}
myQ.sync {
    // do something and wait
}
```

Dispatch Groups

```
let group = DispatchGroup()

for i in 0..<10 {
    group.enter()
    DispatchQueue.global().async {
        // ...
        group.leave()
    }
}

group.notify(queue: DispatchQueue.main) {
    print("Finished the group.")
}
```

Thread safety in Swift

- Reference counting
- Initialization of globals in a non-`main` file
- ObjC `atomic` property access

Locking using a semaphore

```
private let semaphore = DispatchSemaphore(value: 1)

private var lockedName: String = "Hello"
var name: String {
    get {
        return lockedName
    }
    set {
        semaphore.wait()
        lockedName = newValue
        semaphore.signal()
    }
}
```