Chapter 32: Concurrency

Section 32.1: Obtaining a Grand Central Dispatch (GCD) queue

Grand Central Dispatch works on the concept of "Dispatch Queues". A dispatch queue executes tasks you designate in the order which they are passed. There are three types of dispatch queues:

- **Serial Dispatch Queues** (aka private dispatch queues) execute one task at a time, in order. They are frequently used to synchronize access to a resource.
- **Concurrent Dispatch Queues** (aka global dispatch queues) execute one or more tasks concurrently.
- The **Main Dispatch Queue** executes tasks on the main thread.

To access the main queue:

Version = 3.0

```
let mainQueue = DispatchQueue.main
```

Version < 3.0

```
let mainQueue = dispatch_get_main_queue()
```

The system provides *concurrent* global dispatch queues (global to your application), with varying priorities. You can access these queues using the `DispatchQueue` class in Swift 3:

Version = 3.0

```
let globalConcurrentQueue = DispatchQueue.global(qos: .default)
```

equivalent to

```
let globalConcurrentQueue = DispatchQueue.global()
```

Version < 3.0

```
let globalConcurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

In iOS 8 or later, the possible quality of service values which may be passed are `.userInteractive`, `.userInitiated`, `.default`, `.utility`, and `.background`. These replace the `DISPATCH_QUEUE_PRIORITY_` constants.

You can also create your own queues with varying priorities:

Version = 3.0

```
let myConcurrentQueue = DispatchQueue(label: "my-concurrent-queue", qos: .userInitiated,
attributes: [.concurrent], autoreleaseFrequency: .workItem, target: nil)
let mySerialQueue = DispatchQueue(label: "my-serial-queue", qos: .background, attributes: [],
autoreleaseFrequency: .workItem, target: nil)
```

Version < 3.0

```
let myConcurrentQueue = dispatch_queue_create("my-concurrent-queue", DISPATCH_QUEUE_CONCURRENT)
let mySerialQueue = dispatch_queue_create("my-serial-queue", DISPATCH_QUEUE_SERIAL)
```

In Swift 3, queues created with this initializer are serial by default, and passing `.workItem` for autorelease frequency ensures an autorelease pool is created and drained for each work item. There is also `.never`, which means you will be managing your own autorelease pools yourself, or `.inherit` which inherits the setting from the environment. In most cases you probably won't use `.never` except in cases of extreme customization.

Section 32.2: Concurrent Loops

GCD provides mechanism for performing a loop, whereby the loops happen concurrently with respect to each other. This is very useful when performing a series of computationally expensive calculations.

Consider this loop:

```
for index in 0 ..< iterations {
    // Do something computationally expensive here
}
```

You can perform those calculations concurrently using `concurrentPerform` (in Swift 3) or `dispatch_apply` (in Swift 2):

`Version = 3.0`

```
DispatchQueue.concurrentPerform(iterations: iterations) { index in
    // Do something computationally expensive here
}
```

`Version < 3.0`

```
dispatch_apply(iterations, queue) { index in
    // Do something computationally expensive here
}
```

The loop closure will be invoked for each `index` from `0` to, but not including, `iterations`. These iterations will be run concurrently with respect to each other, and thus the order that they run is not guaranteed. The actual number of iterations that happen concurrently at any given time is generally dictated by the capabilities of the device in question (e.g. how many cores does the device have).

A couple of special considerations:

- The `concurrentPerform`/`dispatch_apply` may run the loops concurrently with respect to each other, but this all happens synchronously with respect to the thread from which you call it. So, do not call this from the main thread, as this will block that thread until the loop is done.
- Because these loops happen concurrently with respect to each other, you are responsible for ensuring the thread-safety of the results. For example, if updating some dictionary with the results of these computationally expensive calculations, make sure to synchronize those updates yourself.
- Note, there is some overhead associated in running concurrent loops. Thus, if the calculations being performed inside the loop are not sufficiently computationally intensive, you may find that any performance gained by using concurrent loops may be diminished, if not be completely offset, by the overhead associated with the synchronizing all of these concurrent threads.

  So, you are responsible determining the correct amount of work to be performed in each iteration of the loop. If the calculations are too simple, you may employ "striding" to include more work per loop. For example, rather than doing a concurrent loop with 1 million trivial calculations, you may do 100 iterations in your loop, doing 10,000 calculations per loop. That way there is enough work being performed on each thread, so the overhead associated with managing these concurrent loops becomes less significant.

Section 32.3: Running tasks in a Grand Central Dispatch (GCD) queue

`Version = 3.0`

To run tasks on a dispatch queue, use the `sync`, `async`, and `after` methods.

To dispatch a task to a queue asynchronously:

```
let queue = DispatchQueue(label: "myQueueName")

queue.async {
    //do something

    DispatchQueue.main.async {
        //this will be called in main thread
        //any UI updates should be placed here
    }
}
// ... code here will execute immediately, before the task finished
```

To dispatch a task to a queue synchronously:

```
queue.sync {
    // Do some task
}
// ... code here will not execute until the task is finished
```

To dispatch a task to a queue after a certain number of seconds:

```
queue.asyncAfter(deadline: .now() + 3) {
    //this will be executed in a background-thread after 3 seconds
}
// ... code here will execute immediately, before the task finished
```

> **NOTE:** Any updates of the user-interface should be called on the main thread! Make sure, that you put the code for UI updates inside `DispatchQueue.main.async { ... }`

`Version = 2.0`

Types of queue:

```
let mainQueue = dispatch_get_main_queue()
let highQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let backgroundQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0)
```

To dispatch a task to a queue asynchronously:

```
dispatch_async(queue) {
    // Your code run run asynchronously. Code is queued and executed
    // at some point in the future.
}
// Code after the async block will execute immediately
```

To dispatch a task to a queue synchronously:

```
dispatch_sync(queue) {
    // Your sync code
}
// Code after the sync block will wait until the sync task finished
```

To dispatch a task to after a time interval (use `NSEC_PER_SEC` to convert seconds to nanoseconds):

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(2.5 * Double(NSEC_PER_SEC))),
dispatch_get_main_queue()) {
    // Code to be performed in 2.5 seconds here
}
```

To execute a task asynchronously and than update the UI:

```
dispatch_async(queue) {
    // Your time consuming code here
    dispatch_async(dispatch_get_main_queue()) {
        // Update the UI code
    }
}
```

> **NOTE:** Any updates of the user-interface should be called on the main thread! Make sure, that you put the code for UI updates inside `dispatch_async(dispatch_get_main_queue()) { ... }`

Section 32.4: Running Tasks in an OperationQueue

You can think of an `OperationQueue` as a line of tasks waiting to be executed. Unlike dispatch queues in GCD, operation queues are not FIFO (first-in-first-out). Instead, they execute tasks as soon as they are ready to be executed, as long as there are enough system resources to allow for it.

Get the main `OperationQueue`:

Version ≥ 3.0

```
let mainQueue = OperationQueue.main
```

Create a custom `OperationQueue`:

Version ≥ 3.0

```
let queue = OperationQueue()
queue.name = "My Queue"
queue.qualityOfService = .default
```

Quality of Service specifies the importance of the work, or how much the user is likely to be counting on immediate results from the task.

Add an `Operation` to an `OperationQueue`:

Version ≥ 3.0

```
// An instance of some Operation subclass
let operation = BlockOperation {
    // perform task here
}

queue.addOperation(operation)
```

Add a block to an `OperationQueue`:

Version ≥ 3.0

```
myQueue.addOperation {
    // some task
}
```

Add multiple `Operations` to an `OperationQueue`:

Version ≥ 3.0

```
let operations = [Operation]()
// Fill array with Operations

myQueue.addOperation(operations)
```

Adjust how many `Operations` may be run concurrently within the queue:

```
myQueue.maxConcurrentOperationCount = 3 // 3 operations may execute at once

// Sets number of concurrent operations based on current system conditions
myQueue.maxConcurrentOperationCount = NSOperationQueueDefaultMaxConcurrentOperationCount
```

Suspending a queue will prevent it from starting the execution of any existing, unstarted operations or of any new operations added to the queue. The way to resume this queue is to set the `isSuspended` back to `false`:

Version ≥ 3.0

```
myQueue.isSuspended = true

// Re-enable execution
myQueue.isSuspended = false
```

Suspending an `OperationQueue` does not stop or cancel operations that are already executing. One should only attempt suspending a queue that you created, not global queues or the main queue.

Section 32.5: Creating High-Level Operations

The Foundation framework provides the `Operation` type, which represents a high-level object that encapsulates a portion of work that may be executed on a queue. Not only does the queue coordinate the performance of those operations, but you can also establish dependencies between operations, create cancelable operations, constrain the degree of concurrency employed by the operation queue, etc.

`Operations` become ready to execute when all of its dependencies are finished executing. The `isReady` property then changes to `true`.

Create a simple non-concurrent `Operation` subclass:

Version ≥ 3.0

```
class MyOperation: Operation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }

}
```

Version ≤ 2.3

```
class MyOperation: NSOperation {

    init(<parameters>) {
        // Do any setup work here
    }
    override func main() {
        // Perform the task
    }

}
```

Add an operation to an `OperationQueue`:

Version ≥ 1.0

```
myQueue.addOperation(operation)
```

This will execute the operation concurrently on the queue.

Manage dependencies on an `Operation`.

Dependencies define other `Operation`s that must execute on a queue before that `Operation` is considered ready to execute.

Version ≥ 1.0

```
operation2.addDependency(operation1)

operation2.removeDependency(operation1)
```

Run an `Operation` without a queue:

Version ≥ 1.0

```
    operation.start()
```

Dependencies will be ignored. If this is a concurrent operation, the task may still be executed concurrently if its `start` method offloads work to background queues.

Concurrent Operations.

If the task that an `Operation` is to perform is, itself, asynchronous, (e.g. a `URLSession` data task), you should implement the `Operation` as a concurrent operation. In this case, your `isAsynchronous` implementation should return `true`, you'd generally have `start` method that performs some setup, then calls its `main` method which actually executes the task.

When implementing an asynchronous `Operation` begins you must implement `isExecuting`, `isFinished` methods and KVO. So, when execution starts, `isExecuting` property changes to `true`. When an `Operation` finishes its task, `isExecuting` is set to `false`, and `isFinished` is set to `true`. If the operation it is cancelled both `isCancelled` and `isFinished` change to `true`. All of these properties are key-value observable.

Cancel an `Operation`.

Calling `cancel` simply changes the `isCancelled` property to `true`. To respond to cancellation from within your own `Operation` subclass, you should check the value of `isCancelled` at least periodically within `main` and respond appropriately.

Version ≥ 1.0

```
operation.cancel()
```