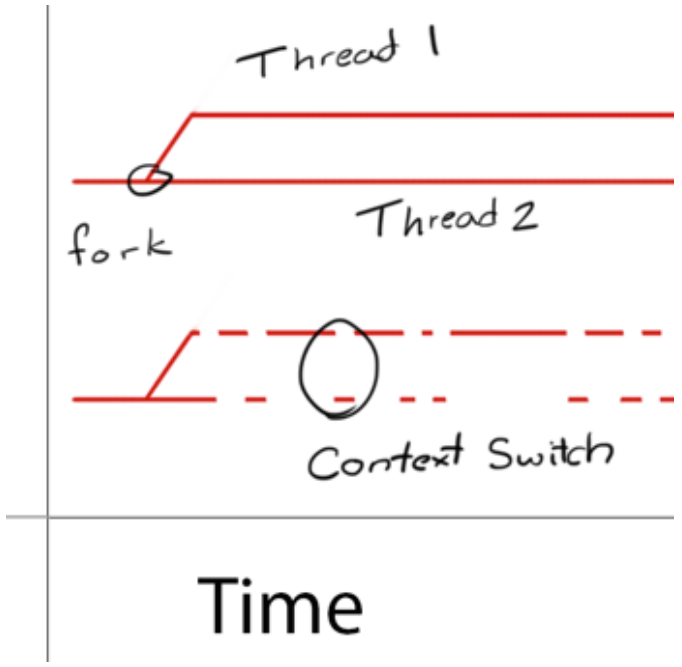
Grand Central Dispatch Tutorial for Swift: Part 1/2

Bjørn Ruud on January 7, 2015

Update note: This tutorial was updated for iOS 8 and Swift by Bjørn Ruud. Original post by Tutorial Team member Derek Selander.

Although **Grand Central Dispatch** (or GCD for short) has been around for a while, not everyone knows how to get the most out of it. This is understandable; concurrency is tricky, and GCD's C-based API can seem like a set of pointy corners poking into the smooth world of Swift.

In this two-part series, you'll learn the ins and outs of GCD. This first part will explain what GCD does and showcase several of the more basic GCD functions. In the second part, you'll learn several of the more advanced functions GCD has to offer.



Learn about concurrency in this Grand Central Dispatch in-depth tutorial series.

Getting Started

GCD is the marketing name for **libdispatch**, Apple's library that provides support for concurrent code execution on multicore hardware on iOS and OS X. It offers the following benefits:

- GCD can improve your app's responsiveness by helping you defer computationally expensive tasks and run them in the background.
- GCD provides an easier concurrency model than locks and threads and helps to avoid concurrency bugs.

To understand GCD, you need to be comfortable with several concepts related to threading and concurrency. These can be both vague and subtle, so take a moment to review them briefly in the context of GCD.

Serial vs. Concurrent

These terms describe *when* tasks are executed with respect to each other. Tasks executed *serially* are always executed one at a time. Tasks executed *concurrently* might be executed at the same time.

Tasks

For the purposes of this tutorial you can consider a task to be a closure. In fact, you can also use GCD with function pointers, but in most cases this is substantially more tricky to use. Closures are just easier!

Don't know what a closure is in Swift? Closures are self-contained, callable blocks of code that can be stored and passed around. When called, they behave like functions and can have parameters and return values. In addition a closure "captures" variables it uses from outside its own scope — that is, it sees the variables from the enclosing scope and remembers their value.

Swift closures are similar to Objective-C blocks and they are almost entirely interchangeable. Their only limitation is just that you cannot, from Objective-C, interact with Swift closures that expose Swift-only language features, like tuples. But interacting with Objective-C from Swift is unhindered, so whenever you read documentation that refers to an Objective-C block then you can safely substitute a Swift closure.

Synchronous vs. Asynchronous

These terms describe when a function will return control to the caller, and how much work will have been done by that point.

A *synchronous* function returns only after the completion of a task that it orders.

An *asynchronous* function, on the other hand, returns immediately, ordering the task to be done but not waiting for it. Thus, an asynchronous function does not block the current thread of execution from proceeding on to the next function.

Be careful — when you read that a synchronous function "blocks" the current thread, or that the function is a "blocking" function or blocking operation, don't get confused! The verb *blocks* describes how a function affects its own thread and has no connection to the noun *block*, which describes an anonymous function literal in Objective-C. Also keep in mind that whenever the GCD documentation refers to Objective-C blocks it is interchangeable with Swift closures.

Critical Section

This is a piece of code that must *not* be executed concurrently, that is, from two threads at once. This is usually because the code manipulates a shared resource such as a variable that can become corrupt if it's accessed by concurrent processes.

Race Condition

This is a situation where the behavior of a software system depends on a specific sequence or timing of events that execute in an uncontrolled manner, such as the exact order of execution of the program's concurrent tasks. Race conditions can produce unpredictable behavior that aren't immediately evident through code inspection.

Deadlock

Two (or sometimes more) items — in most cases, threads — are said to be deadlocked if they all get stuck waiting for each other to complete or perform another action. The first can't finish because it's waiting for the second to finish. But the second can't finish because it's waiting for the first to finish.

Thread Safe

Thread safe code can be safely called from multiple threads or concurrent tasks without causing any problems (data corruption, crashing, etc). Code that is not thread safe must only be run in one context at a time. An example of thread safe code is **let a = ["thread-safe"]**. This array is read-only and you can use it from multiple threads at the same time without issue. On the other hand, an array declared with **var a = ["thread-unsafe"]** is mutable and can be modified. That means it's not thread-safe since several threads can access and modify the array at the same time with unpredictable results. Variables and data structures that are mutable and not inherently thread-safe should only be accessed from one thread at a time.

Context Switch

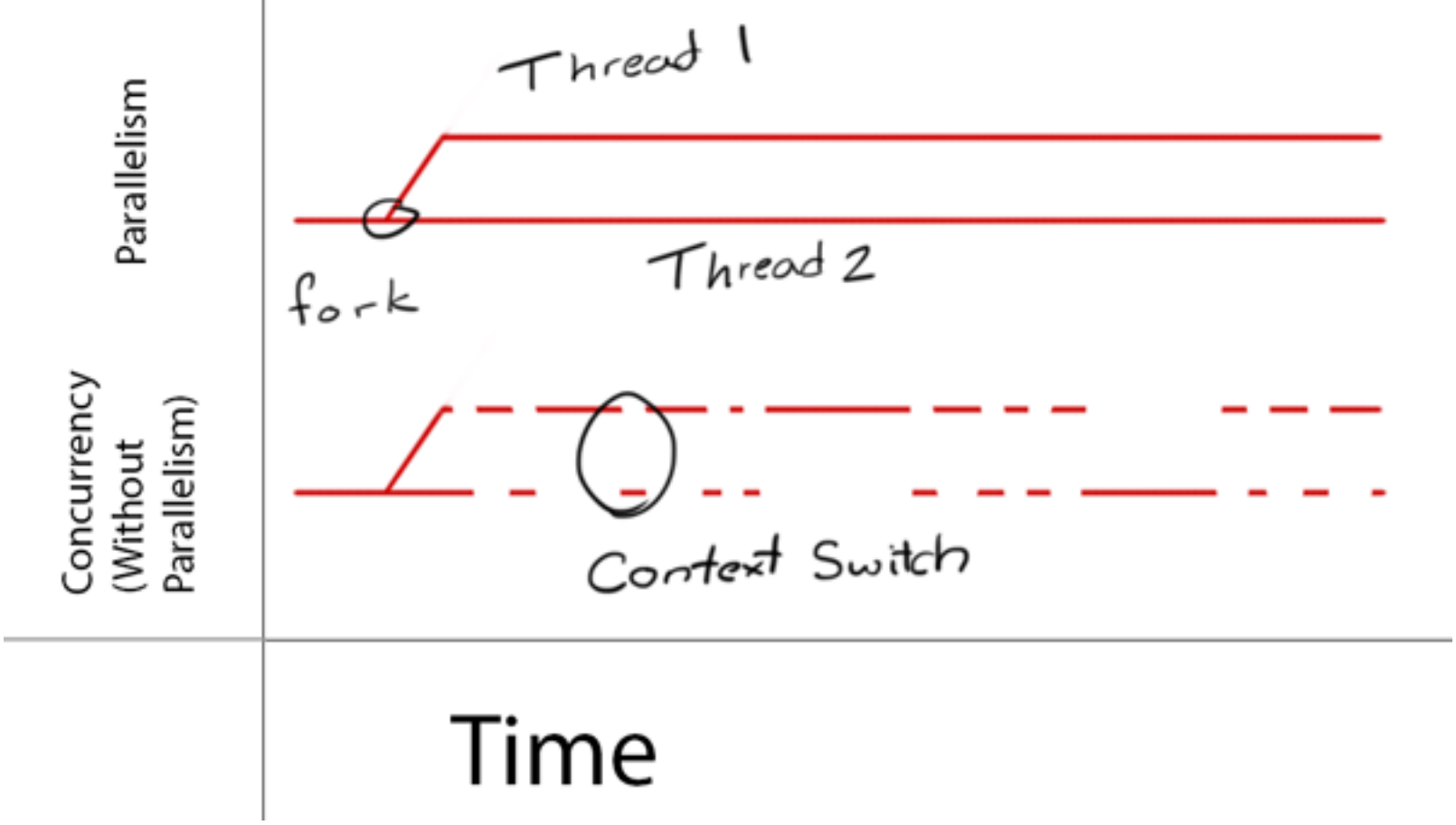
A context switch is the process of storing and restoring execution state when you switch between executing different threads on a single process. This process is quite common when writing multitasking apps, but comes at a cost of some additional overhead.

Concurrency vs Parallelism

Concurrency and parallelism are often mentioned together, so it's worth a short explanation to distinguish them from each other.

Separate parts of **concurrent** code can be executed "simultaneously". However, it's up to the system to decide how this happens — or if it happens at all.

Multi-core devices execute multiple threads at the same time via parallelism; however, in order for single-cored devices to achieve this, they must run a thread, perform a context switch, then run another thread or process. This usually happens quickly enough as to give the illusion of parallel execution as shown by the diagram below:



Although you may write your code to use concurrent execution under GCD, it's up to GCD to decide how much parallelism is required. Parallelism *requires* concurrency, but concurrency does not *guarantee* parallelism.

The deeper point here is that concurrency is actually about *structure*. When you code with GCD in mind, you structure your code to expose the pieces of work that can run simultaneously, as well as the ones that must not be run simulataneously. If you want to delve more deeply into this subject, check out this excellent talk by Rob Pike.

Queues

GCD provides **dispatch queues** to handle submitted tasks; these queues manage the tasks you provide to GCD and execute those tasks in FIFO order. This guarantees that the first task added to the queue is the first task started in the queue, the second task added will be the second to start, and so on down the line.

All dispatch queues are themselves thread-safe in that you can access them from multiple threads simultaneously. The benefits of GCD are apparent when you understand how dispatch queues provide thread-safety to parts of your own code. The key to this is to choose the right *kind* of dispatch queue and the right *dispatching function* to submit your work to the queue.

Serial Queues

Tasks in **serial queues** execute one at a time, each task starting only after the preceding task has finished. As well, you won't know the amount of time between one task ending and the next one beginning, as shown in the diagram below:

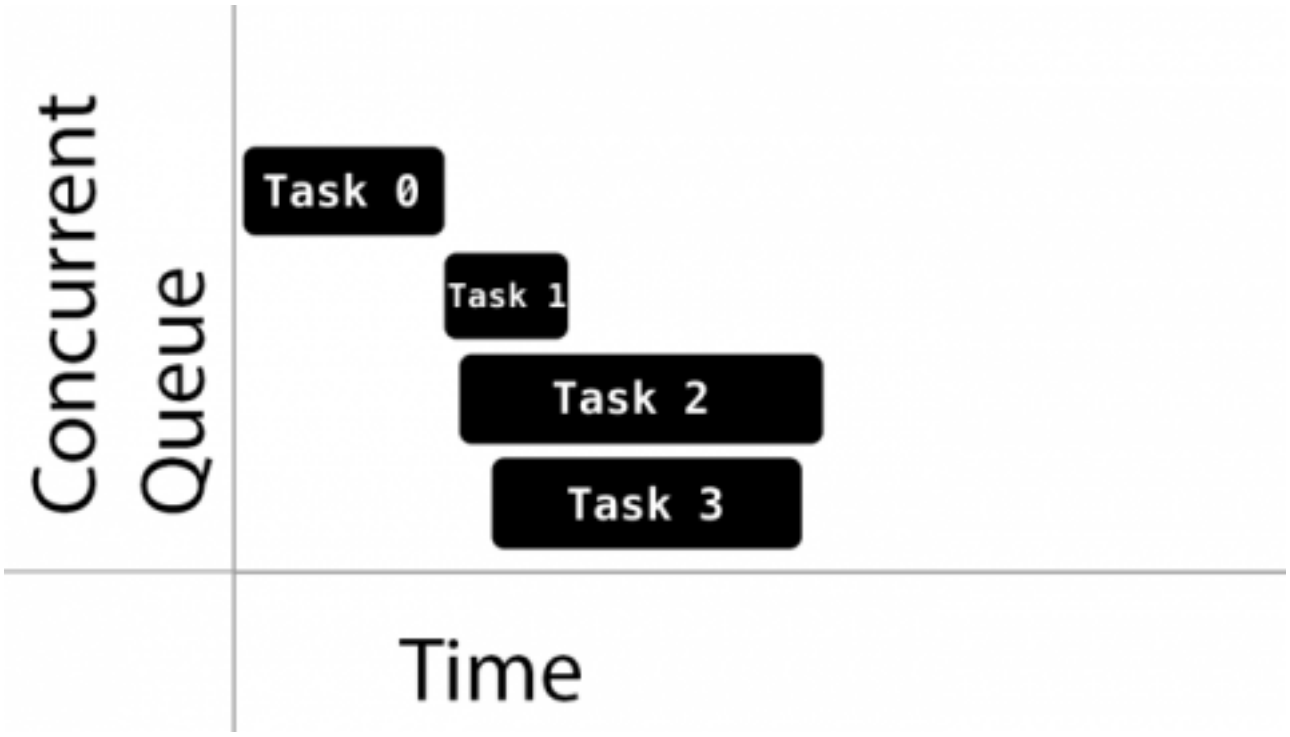
The execution timing of these tasks is under the control of GCD; the only thing you're guaranteed to know is that GCD executes only one task at a time and that it executes the tasks in the order they were added to the queue.

Since no two tasks in a serial queue can ever run concurrently, there is no risk they might access the same critical section at the same time; that protects the critical section from race conditions *with respect to those tasks* only. So if the only way to access that critical section is via a task submitted to that dispatch queue, then you can be sure that the critical section is safe.

Concurrent Queues

Tasks in **concurrent queues** are guaranteed to start in the order they were added… and that's about all you're guaranteed! Items can finish in any order and you have no knowledge of the time it will take for the next task to start, nor the number of tasks that are running at any given time. Again, this is entirely up to GCD.

The diagram below shows a sample task execution plan of four concurrent tasks under GCD:



Notice how Task 1, 2, and 3 all ran quickly, one after another, while it took a while for Task 1 to start after Task 0 started. Also, Task 3 started after Task 2 but finished first.

The decision of when to start a task is entirely up to GCD. If the execution time of one task overlaps with another, it's up to GCD to determine if it should run on a different core, if one is available, or instead to perform a context switch to a different task.

Just to make things interesting, GCD provides you with at least *five* particular queues to choose from within each queue type.

Queue Types

First, the system provides you with a special serial queue known as the **main queue**. Like any serial queue, tasks in this queue execute one at a time. However, it's guaranteed that all tasks will execute on the main thread, which is the only thread allowed to update your UI. This queue is the one to use for sending messages to **UIView** objects or posting notifications.

The system also provides you with several concurrent queues. These queues are linked with their own Quality of Service (QoS) class. The QoS classes are meant to express the intent of the submitted task so that GCD can determine how to best prioritize it:

- **QOS_CLASS_USER_INTERACTIVE**: The **user interactive** class represents tasks that need to be done immediately in order to provide a nice user experience. Use it for UI updates, event handling and small workloads that require low latency. The total amount of work done in this class during the execution of your app should be small.
- **QOS_CLASS_USER_INITIATED**: The **user initiated** class represents tasks that are initiated from the UI and can be performed asynchronously. It should be used when the user is waiting for immediate results, and for tasks required to continue user interaction.
- **QOS_CLASS_UTILITY**: The **utility** class represents long-running tasks, typically with a user-visible progress indicator. Use it for computations, I/O, networking, continous data feeds and similar tasks. This class is designed to be energy efficient.
- **QOS_CLASS_BACKGROUND**: The **background** class represents tasks that the user is not directly aware of. Use it for prefetching, maintenance, and other tasks that don't require user interaction and aren't time-sensitive.

Be aware that Apple's APIs also uses the global dispatch queues, so any tasks you add won't be the only ones on these queues.

Finally, you can also create your own custom serial or concurrent queues. That means you have at least *five* queues at your disposal: the main queue, four global dispatch queues, plus any custom queues that you add to the mix!

And that's the big picture of dispatch queues!

The "art" of GCD comes down to choosing the right queue dispatching function to submit your work to the queue. The best way to experience this is to work through the examples below, where we've provided some general recommendations along the way.

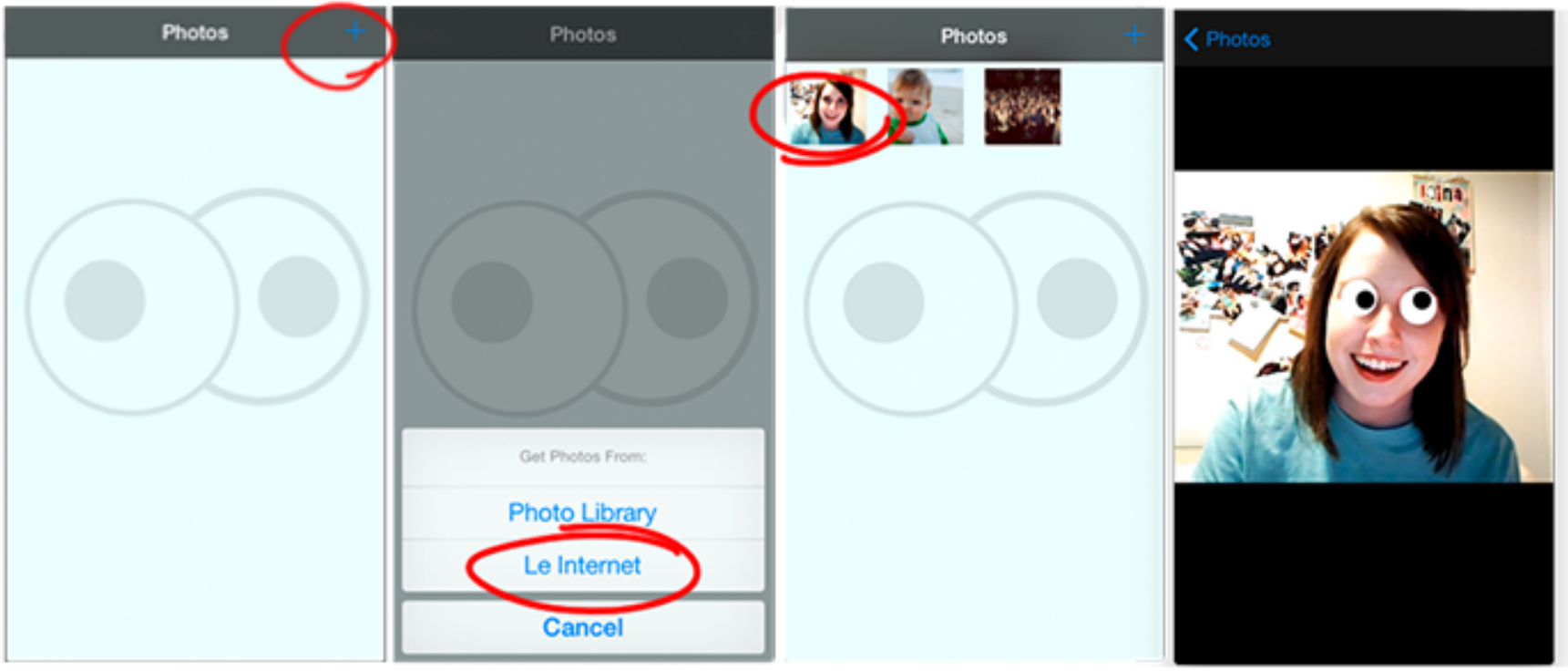
Sample Project

Since the goal of this tutorial is to optimize as well as safely call code from different threads using GCD, you'll start with an almost-finished project named **GooglyPuff**.

GooglyPuff is a non-optimized, threading-unsafe app that overlays googly eyes on detected faces using Core Image's face detection API. For the base image you can select any from the Photo Library or from a set of predefined URL images downloaded from the internet.

GooglyPuff_Swift_Start_1

Once you've downloaded the project, extract it to a convenient location, then open it up in Xcode and build and run. The app will look like the following:



Notice when you choose the **Le Internet** option to download pictures, a **UIAlertController** alert view pops up prematurely. You'll fix this in the second part of this series.

There are four classes of interest in this project:

- **PhotoCollectionViewController**: This is the first view controller that starts the app. It showcases all the selected photos through their thumbnails.
- **PhotoDetailViewController**: This performs the logic to add googly eyes to the image and to display the resulting image in a **UIScrollView**.
- **Photo**: This is a protocol describing the properties of a photo. It provides an image, thumbnail and a status. Two classes are provided which implement the protocol: **DownloadPhoto** which instantiates photos from an instance of **NSURL**, and **AssetPhoto** which instantiates a photo from an instance of **ALAsset**.
- **PhotoManager**: This manages all the **Photo** objects.

Handling Background Tasks with dispatch_sync

Head back to the app and add some photos from your Photo Library or use the **Le Internet** option to download a few.

Notice how long it takes for a new **PhotoDetailViewController** to instantiate after tapping on a **UICollectionViewCell** in the **PhotoCollectionViewController**; there's a noticeable lag, especially when viewing large images on slower devices.

It's easy to overload **UIViewController's viewDidLoad** with too much clutter; this often results in longer waits before the view controller appears. If possible, it's best to offload some work to be done in the background if it's not absolutely essential at load time.

This sounds like a job for **dispatch_async**!

Open **PhotoDetailViewController** and replace **viewDidLoad** with the following implementation:

```
override func viewDidLoad() {
  super.viewDidLoad()
  assert(image != nil, "Image not set; required to use view controller")
  photoImageView.image = image

  // Resize if neccessary to ensure it's not pixelated
  if image.size.height <= photoImageView.bounds.size.height &&
     image.size.width <= photoImageView.bounds.size.width {
    photoImageView.contentMode = .Center
  }

  dispatch_async(dispatch_get_global_queue(Int(QOS_CLASS_USER_INITIATED.value), 0)) { // 1
    let overlayImage = self.faceOverlayImageFromImage(self.image)
```

```
    dispatch_async(dispatch_get_main_queue()) { // 2
      self.fadeInNewImage(overlayImage) // 3
    }
  }
}
```

Here's what's going on in the modified code above:

1. You first move the work off of the main thread and onto a global queue. Because this is a **dispatch_async** call, the closure is submitted asynchronously meaning that execution of the calling thread continues. This lets **viewDidLoad** finish earlier on the main thread and makes the loading feel more snappy. Meanwhile, the face detection processing is started and will finish at some later time.

2. At this point, the face detection processing is complete and you've generated a new image. Since you want to use this new image to update your **UIImageView**, you add a new closure to the main queue. Remember – you must always access **UIKit** classes on the main thread!

3. Finally, you update the UI with **fadeInNewImage** which performs a fade-in transition of the new googly eyes image.

Note that you're using Swift's *trailing closure syntax*, passing closures to **dispatch_async** by writing them *after* the parentheses that contain the earlier parameter specifying the dispatch queue. That syntax can be a little cleaner looking since the closure isn't nested inside the function's parentheses.

Build and run your app; select an image and you'll notice that the view controller loads up noticeably faster and adds the googly eyes after a short delay. This lends a nice effect to the app as you show the before and after photo for maximum impact. As well, if you tried to load an insanely huge image, the app wouldn't hang in the process of loading the view controller, which allows the app to scale well.

As mentioned above, **dispatch_async** appends a task in the form of a closure onto a queue and returns immediately. The task will then be executed at some later time as decided by GCD. Use **dispatch_async** when you need to perform a network-based or CPU intensive task in the background and not block the current thread.

Here's a quick guide of how and when to use the various queue types with **dispatch_async**:

- **Custom Serial Queue**: A good choice when you want to perform background work serially and track it. This eliminates resource contention since you know only one task at a time is executing. Note that if you need the data from a method, you must inline another closure to retrieve it or consider using **dispatch_sync**.
- **Main Queue (Serial)**: This is a common choice to update the UI after completing work in a task on a concurrent queue. To do this, you'll code one closure inside another. As well, if you're in the main queue and call **dispatch_async** targeting the main queue, you can guarantee that this new task will execute sometime after the current method finishes.
- **Concurrent Queue**: This is a common choice to perform non-UI work in the background.

Helper Variables for Getting Global Queues

You may have noticed that the QoS class parameter for **dispatch_get_global_queue** is a bit cumbersome to write. This is due to **qos_class_t** being defined as a struct with a **value** property of type **UInt32**, which must be typecast to **Int**. Add some global computed helper variables to **Utils.swift**, below the URL variables, to make getting a global queue a bit easier:

```
var GlobalMainQueue: dispatch_queue_t {
  return dispatch_get_main_queue()
}

var GlobalUserInteractiveQueue: dispatch_queue_t {
  return dispatch_get_global_queue(Int(QOS_CLASS_USER_INTERACTIVE.value), 0)
}

var GlobalUserInitiatedQueue: dispatch_queue_t {
  return dispatch_get_global_queue(Int(QOS_CLASS_USER_INITIATED.value), 0)
}

var GlobalUtilityQueue: dispatch_queue_t {
```

```
  return dispatch_get_global_queue(Int(QOS_CLASS_UTILITY.value), 0)
}

var GlobalBackgroundQueue: dispatch_queue_t {
  return dispatch_get_global_queue(Int(QOS_CLASS_BACKGROUND.value), 0)
}
```

Go back to **viewDidLoad** in **PhotoDetailViewController** and replace **dispatch_get_global_queue** and **dispatch_get_-main_queue** with the helper variables:

```
dispatch_async(GlobalUserInitiatedQueue) {
  let overlayImage = self.faceOverlayImageFromImage(self.image)
  dispatch_async(GlobalMainQueue) {
    self.fadeInNewImage(overlayImage)
  }
}
```

This makes the dispatch calls much more readable and easy to see which queues are in use.

Delaying Work with dispatch_after

Consider the UX of your app for a moment. It's possible that users might be confused about what to do when they open the app for the first time — were you? :]

It would be a good idea to display a prompt to the user if there aren't any photos in the **PhotoManager** class. However, you also need to think about how the user's eyes will navigate the home screen: if you display a prompt too quickly, they might miss it as their eyes lingered on other parts of the view.

A one-second delay before displaying the prompt should be enough to catch the user's attention as they get their first look at the app.

Add the following code to the stubbed-out implementation of **showOrHideNavPrompt** in **PhotoCollectionViewCon-troller.swift**, near the bottom of the file:

```
func showOrHideNavPrompt() {
  let delayInSeconds = 1.0
  let popTime = dispatch_time(DISPATCH_TIME_NOW,
                              Int64(delayInSeconds * Double(NSEC_PER_SEC))) // 1
  dispatch_after(popTime, GlobalMainQueue) { // 2
    let count = PhotoManager.sharedManager.photos.count
    if count > 0 {
      self.navigationItem.prompt = nil
    } else {
      self.navigationItem.prompt = "Add photos with faces to Googlyify them!"
    }
  }
}
```

showOrHideNavPrompt executes in **viewDidLoad** and anytime your **UICollectionView** is reloaded. Taking each num-bered comment in turn:

1. You declare the variable that specifies the amount of time to delay.
2. You then wait for the amount of time given in the **delayInSeconds** variable and then asynchronously add the closure to the main queue.

Build and run the app. There should be a slight delay, which will hopefully grab the user's attention and show them what to do.

dispatch_after works just like a delayed **dispatch_async**. You still have no control over the actual time of execution nor can you cancel this once **dispatch_after** returns.

Wondering when it's appropriate to use **dispatch_after**?

- **Custom Serial Queue**: Use caution when using **dispatch_after** on a custom serial queue. You're better off sticking to

the main queue.

- **Main Queue (Serial)**: This is a good choice for **dispatch_after**; Xcode has a nice autocomplete template for this.
- **Concurrent Queue**: Use caution when using **dispatch_after** on custom concurrent queues; it's rare that you'll do this. Stick to the main queue for these operations.

Singletons and Thread Safety

Singletons. Love them or hate them, they're as popular in iOS as cats are on the web. :]

One frequent concern with singletons is that often they're not thread safe. This concern is well-justified given their use: singletons are often used from multiple controllers accessing the singleton instance at the same time. The **PhotoManager** class is a singleton, so you will need to consider this issue.

There are two cases to consider, thread-safety during initialization of the singleton instance and during reads and writes to the instance.

Let us consider initialization first. This turns out to be the easy case, because of how Swift initializes variables at global scope. In Swift, global variables are initialized when they are first accessed, and they are guaranteed to be initialized in an atomic fashion. That is, the code performing initialization is treated as a critical section and is guaranteed to complete before any other thread gets access to the global variable. How does Swift do this for us? Behind the scenes, Swift itself is using GCD, by using the function **dispatch_once**, as discussed in this Swift Blog post.

dispatch_once executes a closure once and only once in a thread safe manner. If one thread is in the middle of executing the critical section — the task passed to **dispatch_once** — then other threads will just block until it completes. And once it does it complete, then those threads and other threads will not execute the section at all. And by defining the singleton as a global constant with **let**, we can further guarantee the variable will never be changed after initialization. In a sense, all Swift global constants are naturally born as singletons, with thread-safe initialization.

But we still need to consider reads and writes. While Swift uses **dispatch_once** to ensure that we are initializing the singleton in a thread-safe manner, it does not make the data type it represents thread safe. For example if the global variable is an instance of a class, you could still have critical sections within the class that manipulate internal data. Those would need to be made thread safe in other ways, such as by synchronizing access to the data, as you'll see in the following sections.

Handling the Readers and Writers Problem

Thread-safe instantiation is not the only issue when dealing with singletons. If a singleton property represents a mutable object, like the **photos** array in **PhotoManager**, then you need to consider whether that object is itself thread-safe.

In Swift any variable declared with the **let** keyword is considered a constant and is read-only and thread-safe. Declare the variable with the **var** keyword however, and it becomes mutable and not thread-safe unless the data type is designed to be so. The Swift collection types like **Array** and **Dictionary** are not thread-safe when declared mutable. What about Foundation containers like **NSArray**? Are they thread safe? The answer is — "probably not"! Apple maintains a helpful list of the numerous Foundation classes which are not thread-safe.

Although many threads can read a mutable instance of **Array** simultaneously without issue, it's not safe to let one thread modify the array while another is reading it. Your singleton doesn't prevent this condition from happening in its current state.

To see the problem, have a look at **addPhoto** in **PhotoManager.swift**, which has been reproduced below:

```
func addPhoto(photo: Photo) {
  _photos.append(photo)
  dispatch_async(dispatch_get_main_queue()) {
    self.postContentAddedNotification()
  }
}
```

This is a **write** method as it modifies a mutable array object.

Now take a look at the **photos** property, reproduced below:

```
private var _photos: [Photo] = []
var photos: [Photo] {
```

```
  return _photos
}
```

The getter for this property is termed a **read** method as it's reading the mutable array. The caller gets a copy of the array and is protected against mutating the original array inappropriately, but none of this provides any protection against one thread calling the write method **addPhoto** while simultaneously another thread calls the getter for the **photos** property.

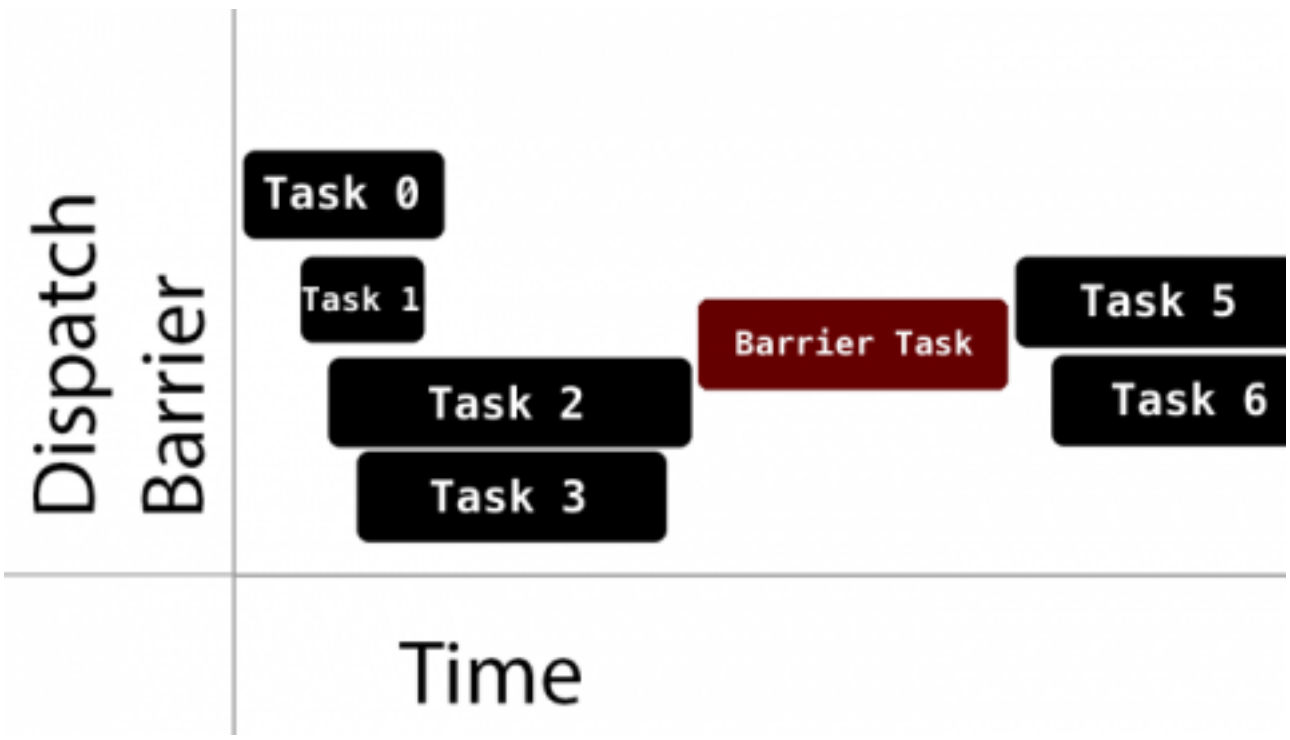
> **Note:** In the code above, why does the caller get a copy of the **photos** array? In Swift parameters and return types of functions are either passed by reference or by value. Passing by reference is the same as passing a pointer in Objective-C, which means you get access to the original object and any changes to it will be seen by any other parts of the code holding a reference to the same object. Passing by value results in a copy of the object, and changes to the copy will not affect the original. By default in Swift class instances are passed by reference and structs by value.
>
> Swift's built-in data types, like **Array** and **Dictionary**, are implemented as structs, and as a result there will seemingly be a lot of copying in your code when passing collections back and forth. Don't worry about the memory usage implications of this. The Swift collection types are optimized to only make copies when necessary, for instance when an array passed by value is modified for the first time after being passed.

This is the classic software development Readers-Writers Problem. GCD provides an elegant solution of creating a read/write lock using **dispatch barriers**.

Dispatch barriers are a group of functions acting as a serial-style bottleneck when working with concurrent queues. Using GCD's barrier API ensures that the submitted closure is the only item executed on the specified queue for that particular time. This means that all items submitted to the queue prior to the dispatch barrier must complete before the closure will execute.

When the closure's turn arrives, the barrier executes the closure and ensures that the queue does not execute any other closures during that time. Once finished, the queue returns to its default implementation. GCD provides both synchronous and asynchronous barrier functions.

The diagram below illustrates the effect of barrier functions on various asynchronous tasks:



Notice how in normal operation the queue acts just like a normal concurrent queue. But when the barrier is executing, it essentially acts like a serial queue. That is, the barrier is the only thing executing. After the barrier finishes, the queue goes back to being a normal concurrent queue.

Here's when you would — and wouldn't — use barrier functions:

- **Custom Serial Queue**: A bad choice here; barriers won't do anything helpful since a serial queue executes one oper-

ation at a time anyway.

- **Global Concurrent Queue:** Use caution here; this probably isn't the best idea since other systems might be using the queues and you don't want to monopolize them for your own purposes.
- **Custom Concurrent Queue:** This is a great choice for atomic or critical areas of code. Anything you're setting or instantiating that needs to be thread safe is a great candidate for a barrier.

Since the only decent choice above is the custom concurrent queue, you'll need to create one of your own to handle your barrier function and separate the read and write functions. The concurrent queue will allow multiple read operations simultaneously.

Open **PhotoManager.swift** and add the following private property to the class, below the **photos** property:

```
private let concurrentPhotoQueue = dispatch_queue_create(
  "com.raywenderlich.GooglyPuff.photoQueue", DISPATCH_QUEUE_CONCURRENT)
```

This initializes **concurrentPhotoQueue** as a concurrent queue using **dispatch_queue_create**. The first parameter is a reversed DNS style naming convention; make sure it's descriptive since this can be helpful when debugging. The second parameter specifies whether you want your queue to be serial or concurrent.

> **Note:** When searching for examples on the web, you'll often see people pass **0** or **NULL** as the second parameter of **dispatch_queue_create**. This is a dated way of creating a serial dispatch queue; it's always better to be specific with your parameters.

Find **addPhoto** and replace it with the following implementation:

```
func addPhoto(photo: Photo) {
  dispatch_barrier_async(concurrentPhotoQueue) { // 1
    self._photos.append(photo) // 2
    dispatch_async(GlobalMainQueue) { // 3
      self.postContentAddedNotification()
    }
  }
}
```

Here's how your new write function works:

1. Add the write operation using your custom queue. When the critical section executes at a later time this will be the only item in your queue to execute.
2. This is the actual code which adds the object to the array. Since it's a barrier closure, this closure will never run simultaneously with any other closure in **concurrentPhotoQueue**.
3. Finally you post a notification that you've added the image. This notification should be posted from the main thread because it will do UI work, so here you dispatch another task asynchronously to the main queue for the notification.

This takes care of the write, but you also need to implement the **photos** read method.

To ensure thread safety with the writer side of matters, you need to perform the read on the **concurrentPhotoQueue** queue. You need to return from the function though, so you can't dispatch asynchronously to the queue because that wouldn't necessarily run before the reader function returns.

In this case, **dispatch_sync** would be an excellent candidate.

dispatch_sync synchronously submits work and waits for it to be completed before returning. Use **dispatch_sync** to keep track of your work with dispatch barriers, or when you need to wait for the operation to finish before you can use the data processed by the closure.

You need to be careful though. Imagine if you call **dispatch_sync** and target the current queue you're already running on. This will result in a deadlock because the call will wait to until the closure finishes, but the closure can't finish (it can't even

start!) until the currently executing closure is finished, which can't! This should force you to be conscious of which queue you're calling from — as well as which queue you're passing in.

Here's a quick overview of when and where to use **dispatch_sync**:

- **Custom Serial Queue:** Be VERY careful in this situation; if you're running in a queue and call **dispatch_sync** targeting the same queue, you will definitely create a deadlock.
- **Main Queue (Serial):** Be VERY careful for the same reasons as above; this situation also has potential for a deadlock condition.
- **Concurrent Queue:** This is a good candidate to sync work through dispatch barriers or when waiting for a task to complete so you can perform further processing.

Still working in **PhotoManager.swift**, replace the **photos** property with the following implementation:

```
var photos: [Photo] {
  var photosCopy: [Photo]!
  dispatch_sync(concurrentPhotoQueue) { // 1
    photosCopy = self._photos // 2
  }
  return photosCopy
}
```

Taking each numbered comment in turn, you'll find the following:

1. Dispatch synchronously onto the **concurrentPhotoQueue** to perform the read.
2. Store a copy of the photo array in **photosCopy** and return it.

Congratulations — your **PhotoManager** singleton is now thread safe. No matter where or how you read or write photos, you can be confident that it will be done in a safe manner with no surprises.

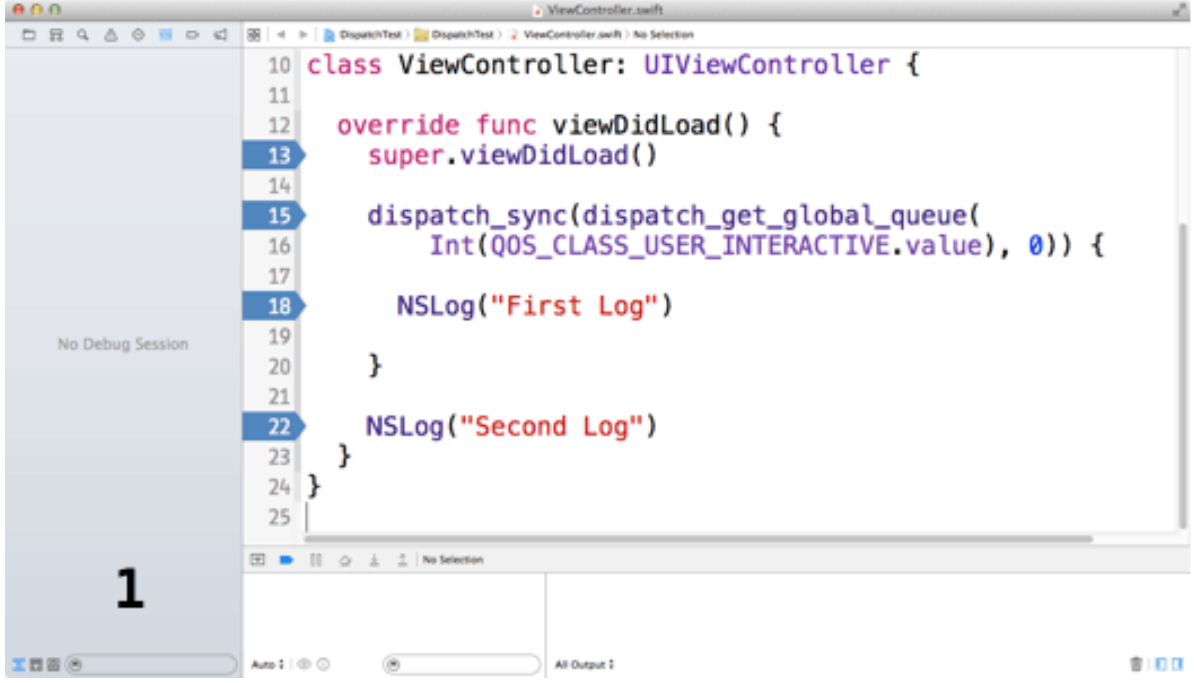
A Visual Review of Queueing

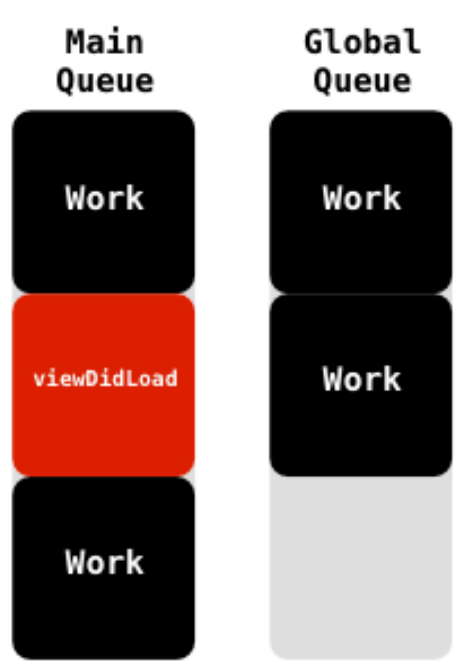
Still not 100% sure on the essentials of GCD? Make sure you're comfortable with the basics by creating simple examples yourself using GCD functions using breakpoints and **NSLog** statements to make sure you understand what is happening.

I've provided two animated GIFs below to help cement your understanding of **dispatch_async** and **dispatch_sync**. The code is included above each GIF as a visual aid; pay attention to each step of the GIF showing the breakpoint in the code on the left and the related queue state on the right.

dispatch_sync Revisited

```
override func viewDidLoad() {
  super.viewDidLoad()

  dispatch_sync(dispatch_get_global_queue(
      Int(QOS_CLASS_USER_INTERACTIVE.value), 0)) {

    NSLog("First Log")

  }

  NSLog("Second Log")
}
```





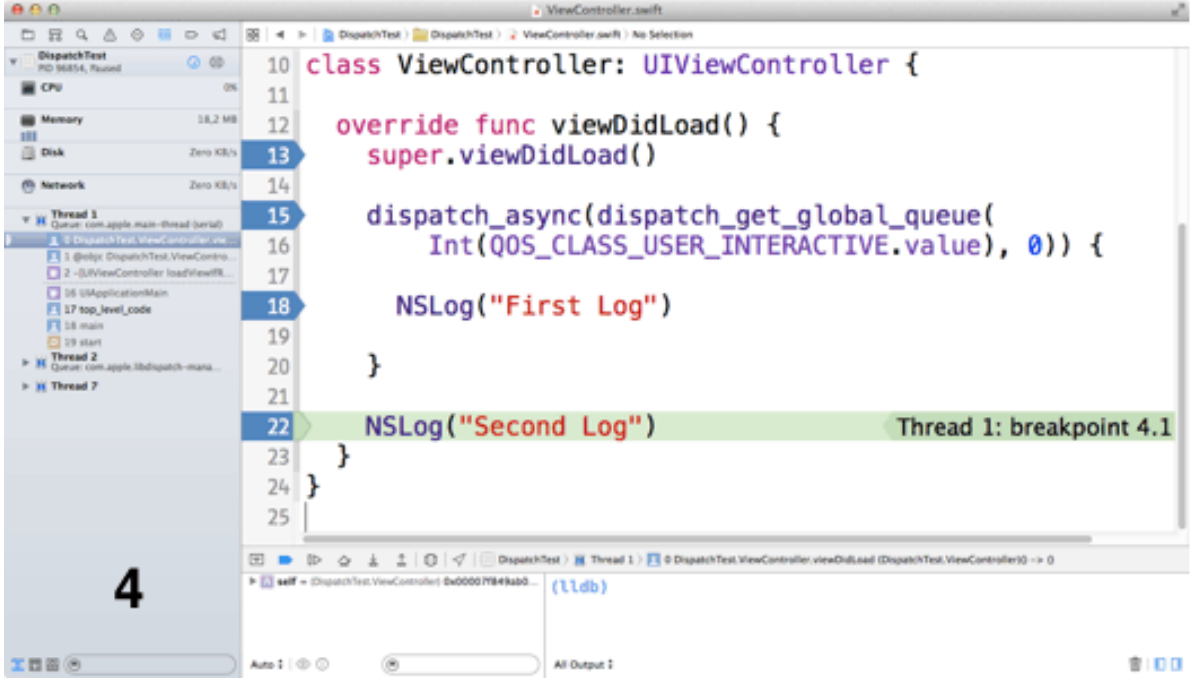
Here's your guide to the various states of the diagram:

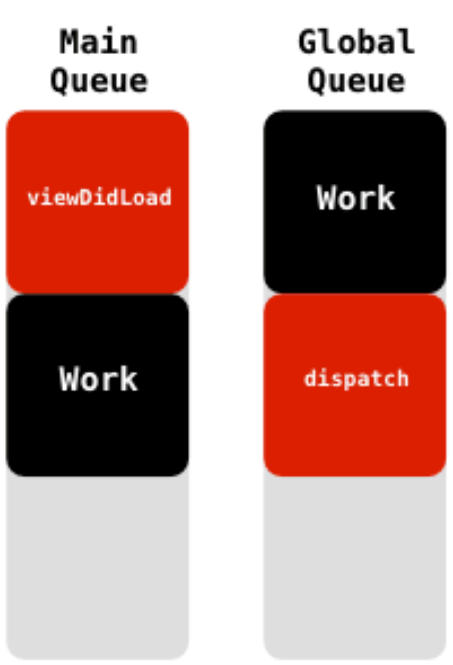
1. The main queue chugs along executing tasks in order — up next is a task to instantiate **UIViewController** which includes **viewDidLoad**.
2. **viewDidLoad** executes on the main thread.
3. The **dispatch_sync** closure is added to a global queue and will execute at a later time. Processes are halted on the main thread until the closure completes. Meanwhile, the global queue is concurrently processing tasks; recall that closures will be dequeued in FIFO order on a global queue but can be executed concurrently. The global queue processes the tasks that were already present on the queue before the **dispatch_sync** closure was added.
4. Finally, the **dispatch_sync** closure has its turn.
5. The closure is done so the tasks on the main thread can resume.
6. **viewDidLoad** method is done, and the main queue carries on processing other tasks.

dispatch_sync adds a task to a queue and waits until that task completes. **dispatch_async** does the exact same thing, but the only exception is that it doesn't wait for the task to complete before proceeding onwards from the calling thread.

dispatch_async Revisited

```
override func viewDidLoad() {
  super.viewDidLoad()

  dispatch_async(dispatch_get_global_queue(
      Int(QOS_CLASS_USER_INTERACTIVE.value), 0)) {

    NSLog("First Log")

  }

  NSLog("Second Log")
}
```





1. The main queue chugs along executing tasks in order — up next is a task to instantiate **UIViewController** which includes **viewDidLoad**.
2. **viewDidLoad** executes on the main thread.
3. The **dispatch_async** closure is added to a global queue and will execute at a later time.
4. **viewDidLoad** continues to move on after adding **dispatch_async** to the global queue and the main thread turns its attention to the remaining tasks. Meanwhile, the global queue is concurrently processing its outstanding tasks. Remember that closures will be dequeued in a FIFO order on a global queue but can be executed concurrently.
5. The closure added by **dispatch_async** is now executing.
6. The **dispatch_async** closure is done and both **NSLog** statements have placed their output on the console.

In this particular instance, the second **NSLog** statement executes, followed by the first **NSLog** statement. This isn't always the case — it's dependent on what the hardware is doing at that given time, and you have no control nor knowledge as to which statement will execute first. The "first" **NSLog** could be the first log to execute in some invocations.

Where to Go From Here?

In this tutorial, you learned how to make your code thread safe and how to maintain the responsiveness of the main thread while performing CPU intensive tasks.

You can download the GooglyPuff Project which contains all the improvements made in this tutorial so far. In the second part of this tutorial you'll continue to improve upon this project.

If you plan on optimizing your own apps, you really should be profiling your work with the **Time Profile** template in **Instruments**. Using this utility is outside the scope of this tutorial, so check out How to Use Instruments for a excellent overview.

Also make sure that you profile with an actual device, since testing on the Simulator can give very different results that are different from what your users will experience.

In the next part of this tutorial you'll dive even deeper into GCD's API to do even more cool stuff.

If you have any questions or comments, feel free to join the discussion below!

Bjørn Olav Ruud

Bjørn is an iOS programmer from Norway. During the day he creates control interfaces for smart home automation solutions. At night he experiments with Swift and games.

In his spare time Bjørn enjoys games (computer/card/board) for his mental health, and

strength training, parkour and other activities that are likely to cause injury for his physical health.

Blog | *GitHub* | *LinkedIn*