## Item 43: Know When to Use GCD and When to Use Operation

GCD is a fantastic technology, but it is sometimes better to use other tools that come as part of the standard system libraries. Knowing when to use each tool is important; using the wrong tool can lead to code that's difficult to maintain.

The synchronization mechanisms of GCD (see Item 41) can hardly be rivaled. The sam goes for single-time code execution through the use of dispatchonce (see Item 45). Howeve using GCD is not always the best approach for executing tasks in the background. separate but related technology, NSOperationQueue, allows you to queue operation (subclasses of NSOperation) that can optionally run concurrently. The similarity to GCD' dispatch queues is not a coincidence. Operation queues came before GCD, but there is n doubt that GCD is based on the principles made popular by operation queues. In fact, fro iOS 4 and Mac OS X 10.6 onward, operation queues use GCD under the hood.

The first difference to note is that GCD is a pure C API, whereas operation queues ar Objective-C objects. In GCD, the task that is queued is a block, which is a fairly lightweig data structure (see Item 37). Operations, on the other hand, are Objective-C objects and ar therefore more heavyweight. That said, GCD is not always the approach of choic Sometimes, this overhead is minimal, and the benefits of using full objects far outweigh th downsides.

Through the use of the NSBlockOperation orNSOperationQueue's addOperationWithBlock method, the syntax of operation queues can look very similar to plain GCD. Here are some c the benefits of NSOperation and NSOperationQueue:

♦ Cancelling operations

With operation queues, this is simple. When run, the cancel method on NSOperation sets internal flags within the operation to tell it not to run, although it cannot cancel an operatio that has already started. On the other hand, GCD queues have no way of cancelling a bloc once it is scheduled. The architecture is very much "fire and forget." Implementing cancellin at the application level, however, would be possible but would require writing a lot of code that has already been written in the form of operations.

♦ Operation dependencies

An operation can have dependencies on as many other operations as it wishes. This enables you to create a hierarchy of operations dictating that certain operations can execut only after another operation has completed successfully. For example, you may have operations to download and process files from a server that requires a manifest file to be downloaded first before others can be processed. The operation to download the manifest file first could be a dependency of the subsequent download operations. If the operation queue were set to allow concurrent execution, the

@implementation EOCClass

had completed.

♦ Key-Value Observing of operation properties

Operations have many properties that are appropriate for KVO, such as isCancelled to determine whether it has been cancelled and isFinished to determine whether it has finishec Using KVO can be useful if you have code that wants to know when a certain task change: state and gives much finer-grained control than GOD over the tasks that are operating.

♦ Operation priorities

An operation has an associated priority that ranks it against other operations in a queue. Higher-priority operations are executed before lower-priority ones. The scheduling algorithn is opaque but most certainly will have been carefully thought out. GOD has no direct way c achieving the same thing. It does have queue priorities, but they set a priority for the entire queue rather than individual blocks. Writing your own scheduler on top of this is not something you really want to do. Priorities are therefore a useful feature of operations.

Operations also have an associated thread priority, which determines at what priority the thread will execute when the operation runs. You could do this yourself with GCD, but operations make it as simple as setting a property.

♦ Reuse of operations

Unless you use one of the built-in concrete subclasses of NSOperation, such as NSBlockOperation, you must create your own subclass. This class, being a normal Objective-C object, can store whatever information you want. When it runs, it has full use c this information and any methods that have been defined on the class. This makes it much more powerful than a simple block that is queued on a dispatch queue. These operation classes can be reused throughout your code, thereby following the “Don’t Repeat Yourself (DRY) principle of software development.

As you can see, there are many good reasons to use operation queues over dispatcl queues. Operation queues mostly provide instant solutions to many of the things you migh want to do when executing tasks. Instead of writing complex schedulers, or cance semantics or priorities yourself, you get them for free when using operation queues.

One API that makes use of operation queues rather than dispatch queues i: NSNotificationCenter, which has a method where you can register to observe a notificatioı through a block instead of calling a selector. The method prototype looks like this:

```
- (id)addO bs erverF orN ame: (N S
              String* )name
              object:(id)objectusingBlock:(void
              (^)(NSNotification*))block
```

Instead of taking an operation queue, this method could have taken a dispatch queue oı which to queue the notification-handler block. But clearly, the design decision was made t make use of the higher-level Objective-C API. In this case, there is little difference betweeı the two in terms of efficiency. The decision was possibly made because using a dispatcl queue would introduce an unnecessary dependency on GCD; remember that blocks are nc

@implementation EOCClass

GCD, so the block itself does not introduce this dependency. Or perhaps the developer:

wanted to keep it all in Objective-C.

You will often hear that you should always use the highest-level API possible, droppin down only when truly necessary. I subscribe to this mantra but with caution. Just because i can be done with the high-level Objective-C variant does not necessarily mean that it's bette Benchmarking is always the best way to know for sure what is best.

### Things to Remember

♦ Dispatch queues are not the only solution to multithreading and task management.

♦ Operation queues provide a high-level, Objective-C API that can do most of what plain GCD can do. These queues can also do much more complex things that would require additional code on top of GCD.

## Item 44: Use Dispatch Groups to Take Advantage of Platforn Scaling

Dispatch groups are a GCD feature that allows you to easily group tasks. You can the wait on that set of tasks to finish or be notified through a callback when the set of tasks ha finished. This feature is very useful for several reasons, the first and most interesting c which is when you want to perform multiple tasks concurrently but need to know when the have all finished. An example of this would be performing a task, such as compressing a se of files.

A dispatch group is created with the following function:

```
dispatch_group_t dispatch_group_create();
```

A group is a simple data structure with nothing distinguishing it, unlike a dispatch queue which has an identifier. You can associate tasks with a dispatch group in two ways. The firs is to use the following function:

```
void dispatch_group_async(dispatch_group_t group,
                          dispatch queue t queue,
                          dispatchblockt block);
```

This is a variant of the normal dispatch async function but takes an additional groupsecon way to associate a task with a dispatch group is to use the following pair of functions:

```
void dispatch_group_enter(dispatch_group_t group);

 void dispatch_group_leave(dispatch_group_t group);
```

The former causes the number of tasks the group thinks are currently running t increment; the latter does the opposite. Therefore, for each call to dispatch group enter, ther must be an associated dispatch_group_leave. This is similar to reference counting (see Iten 29), whereby retains and releases must be balanced to avoid leaks. In the case of dispatc groups, if an enter is not balanced with a leave, the group will never finish.

The following function can be used to wait on a dispatch group to finish:

```
@implementation EOCClass
long dispatch_group_wait(dispatch_group_t group, dispatchtimet timeout);
```

This takes the group to wait on and a timeout value. The timeout specifies how long thi function should block while waiting for the group to finish. If the group finishes before th timeout, zero is returned; otherwise, a nonzero value is returned. The constant DISPATCI TIME FOREVER can be used as the timeout value to indicate that the function should wai forever and never time out.

The following function is an alternative to blocking the current thread to wait for a dispatcl group to finish:

```
void dispatch_group_notify(dispatch_group_t group, dispatchqueuet queue, dispatchblockt block);
```

Slightly different from the wait function, this function allows you to specify a block that wi be run on a certain queue when the group is finished. Doing so can be useful if the curren thread should not be blocked, but you still need to know when all the tasks have finished. I both Mac OS X and iOS, for example, you should never block the main thread, as that' where all UI drawing and event handling are done.

An example of using this GCD feature is to perform a task on an array of objects and the wait for all tasks to finish. The following code does this:

```
dispatch queue t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0); dispatch group t
dispatchGroup = dispatch_group_create(); for (id object in collection) {
    dispatch_group_async(dispatchGroup,
                         queue,
                         ^{ [object performTask]; });
}dispatch_group_wait(dispatchGroup, DISPATCH TIME FOREVER);
// Continue processing after completing tasks
```

If the current thread should not be blocked, you can use the notify function instead c waiting:

```
dispatchqueuet notifyQueue = dispatch_get_main_queue();
dispatch_group_notify(dispatchGroup, notifyQueue,
                      ^{
                 // Continue processing after completing tasks
                      });
```

The queue on which the notify callback should be queued is entirely dependent o circumstances. Here, I've shown it being the main queue, which would be a fairly commo use case. But it could also be any custom serial queue or one of the global concurren queues.

In this example, the queue dispatched onto was the same one for all tasks. But thi doesn't have to be the case. You may want to put some tasks at a higher priority but sti group them all into the same dispatch group and be notified when all have finished:

```
dispatch queue t lowPriorityQueue =

  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0);
dispatchqueuet highPriority Queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
dispatch group t dispatchGroup = dispatch_group_create();
```

**@implementation EOCClass**

```
{ dispatch_group_async(dispatchGroup, lowP riority Queue,
                      ^{ [object performTask]; });
}

for (id object in highPriorityObjects) { dispatch_group_async(dispatchGroup, highP riorityQueue,

{ [object performTask]; });

}

dispatch queue t notifyQueue = dispatch_get_main_queue();
dispatch_group_notify(dispatchGroup, notifyQueue,
                    ^{
                // Continue processing after completing tasks
                  });
```

Instead of submitting tasks to concurrent queues as in the preceding examples, you ma instead use dispatch groups to track multiple tasks submitted to different serialserial queue Because the tasks will all execute serially anyway, you could simply queue another bloc after queuing the tasks, which is the equivalent of a dispatch group's notify callback block:

```
dispatchqueuet queue =
dispatch_queue_create("com effectiveobjectivec. queue", NULL);

for (id object in collection) {
dispatch_async(queue,

{ [object performTask]; });

}

                dispatch_async(queue,{ // Continue processing after completing tasks
});
```

This code shows that you don't always need to use something like dispatch groups Sometimes, the desired effect can be achieved by using a single queue and standar asynchronous dispatch.

Why did I mention performing tasks based on system resources? Well, if you look back t the example of dispatching onto a concurrent queue, it should become clear. GCI automatically creates new threads or reuses old ones as it sees fit to service blocks on queue. In the case of concurrent queues, this can be multiple threads, meaning that multipl blocks are executed concurrently. The number of concurrent threads processing a give concurrent queue depends on factors, mostly based on system resources, that GCD decide If the CPU has multiple cores, a queue having a lot of work to do will likely be given multipl threads on which to execute. Dispatch groups provide an easy way to perform a given set c tasks concurrently and be told when that group of tasks has finished. Through the nature c GCD's concurrent queues, the tasks will be executed concurrently and based on availabl system resources. This leaves you to code your business logic and not have to write an kind of complex scheduler to handle concurrent tasks.

The example of looping through a collection and performing a task on each item can als

@implementation EOCClass

```
void dispatch_apply(size_t iterations,
                    dispatch queue t queue,
                    void(^block)(size_t));
```

This function performs a given number of iterations of a block, each time passing a incrementing value from zero to the number of iterations minus one. It is used like this:

In effect, this is equivalent to a simple for loop that iterates from 0 to 9, like this:

```
for (int i = 0; i < 10; i++) {
    // Perform task
}
```

The key thing to note with dispatch apply is that the queue could be a concurrent queue. I so, the blocks will be executed in parallel according to system resources, just like th example of dispatch groups. If the collection in that example were an array, it could b rewritten using dispatch apply like this:

```
dispatch queue t queue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_apply(array. count, queue, ^(si/e_t i){ id object = array[i];
    [object performTask];
});
```

Once again, this example shows that dispatch groups are not always necessary. Howeve dispatch apply blocks until all iterations have finished. For this reason, if you try to run block on the current queue (or a serial queue above the current queue in the hierarchy), deadlock will result. If you want the tasks to be executed in the background, you need to us dispatch groups.

### Things to Remember

♦ Dispatch groups are used to group a set of tasks. You can optionally be notified when th group finishes executing.

♦ Dispatch groups can be used to execute multiple tasks concurrently through a concurren dispatch queue. In this case, GCD handles the scheduling of multiple tasks at the same time based on system resources. Writing this yourself would require a lot of code.

## Item 45: Use **dispatch_once** for Thread-Safe Single-Time Code Execution

The Singleton design pattern-no stranger in the Objective-C world-is usually achieve through a class method called something like sharedlnstance, which returns the singleto instance of a class instead of specifically allocating a new instance each time. A commo implementation of the shared-instance method for a class called EOCClass is the following

```
@implementation EOCClass
```

```
+ (id)sharedlnstance {
    static EOCClass *sharedlnstance =nil; @synchronized(self) { if (! sharedlnstance) {
        sharedlnstance = [[self alloc] init];
      }
    }
    return sharedlnstance;
}@end
```

I have found that the Singleton pattern generates hot debate, especially in Objective- C Thread safety is the primary candidate for debate. The preceding code creates the singletor instance enclosed in a synchronization block to make it thread safe. For better or worse, the pattern is commonly used, and such code is commonplace.

However, GCD introduced a feature that makes singleton instances much easier to implement. The function is as follows:

```
void dispatch_once(dispatch_once_t *token,
                   dispatchblockt block);
```

This function takes a special dispatch once t type, which I will call the “token,” and a block The function ensures that for a given token, the block is executed once and only once. The block is always executed the first time and, most important, is entirely thread safe. Note tha the token passed in needs to be exactly the same one for each block that should be executed once. This usually means declaring the token variable in static or global scope.

Rewriting the singleton shared-instance method with this function looks like this:

```
+ (id)sharedlnstance {
    static EOCClass *sharedlnstance =nil; static dispatch once t onceToken; dispatch_once(&onceToken, ^{ sharedlnstance = [[self
alloc] init];
    });
    return sharedlnstance;
}
```

Using dispatch once simplifies the code and ensures thread safety entirely, so you don' even have to think about locking or synchronization. All that is handled in the depths of GCD The token has been declared static because it needs to be exactly the same token each time Defining the variable in static scope means that the compiler ensures that instead of creating a new variable each time the sharedlnstance method is run, only a single variable is reused each time.

Furthermore, dispatch once is more efficient. Instead of using a heavyweightatomic access to the dispatch token to indicate whether the code has been run yet. A simple benchmark or my 64-bit, Mac OS X 10.8.2 machine, accessing the sharedlnstance method using the @synchronized approach versus the dispatch once approach showed an almost doubled speedup using dispatch once.

## Things to Remember

♦ Thread-safe single-code execution is a common task. GCD provides an easy-to-use too for this with the dispatch once function.

♦ The token should be declared in static or global scope such that it is the same token being

**@implementation EOCClass**

passed in for each block that should be executed once.

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

**@implementation EOCClass**