Unleash Multicore Performance with Grand Central Dispatch

Concurrent Programming in Mac OS X and iOS

O'REILLY®

Vandad Nahavandipoor

Concurrent Programming in Mac OS X and iOS

Now that multicore processors are coming to mobile devices, wouldn't it be great to take advantage of all those cores without having to manage threads? This concise book shows you how to use Apple's Grand Central Dispatch (GCD) to simplify programming on multicore iOS devices and Mac OS X.

Managing your application's resources on more than one core isn't easy, but it's vital. Apps that use only one core in a multicore environment will slow to a crawl. If you know how to program with Cocoa or Cocoa Touch, this guide will get you started with GCD right away, with many examples to help you write high-performing multithreaded apps.

- Package your code as block objects and invoke them with GCD
- Understand dispatch queues—the pools of threads managed by GCD
- Use different methods for executing UI and non-UI tasks
- Create a group of tasks that GCD can run all at once
- Instruct GCD to execute tasks only once or after a delay
- Discover how to construct your own dispatch queues

Twitter: @oreillymedia
facebook.com/oreilly

US $19.99 CAN $22.99

ISBN: 978-1-449-30563-5

9 781449 305635 51999

O'REILLY®
oreilly.com

Concurrent Programming in Mac OS X and iOS

Concurrent Programming in Mac OS X and iOS

Vandad Nahavandipoor

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Concurrent Programming in Mac OS X and iOS
by Vandad Nahavandipoor

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Teresa Elsey
Proofreader: Teresa Elsey
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
June 2011: First Edition.

ISBN: 978-1-449-30563-5

[LSI]

1305640435

Table of Contents

Preface

With the introduction of multicore devices such as the iPad 2 and the quad-core MacBook Pro, writing multithreaded apps that take advantage of multiple cores on a device has become one of the biggest headaches for developers. Take, for instance, the introduction of iPad 2. On the launch day, only a few applications, basically those released by Apple, were able to take advantage of its multiple cores. Applications like Safari performed very well on the iPad 2 compared to the original iPad, but some third-party browsers did not perform as well as Safari. The reason behind this is that Apple has utilized Grand Central Dispatch (GCD) in Safari's code base. GCD is a low-level C API that allows developers to write multithreaded applications without the need to manage threads at all. All developers have to do is define tasks and leave the rest to GCD.

The trend in the industry is *mobility*. Mobile devices, whether they are as compact as an iPhone or as strong and full-fledged as an Apple MacBook Pro, have many fewer resources than computers such as the Mac Pro, because all the hardware has to be placed inside the small devices' compact bodies. Because of this, it is very important to write applications that work smoothly on mobile devices such as the iPhone. We are not that far away from having quad-core or 8-core smartphones. Once we have 8 cores in the CPU, an app executed on only one of the cores will run *tremendously more slowly* than an app that has been optimized with a technology such as GCD, which allows the code to be scheduled on multiple cores without the programmer having to manage this synchronization.

Apple is pushing developers away from using threads and is slowly starting to integrate GCD into its various frameworks. For instance, prior to the introduction of GCD in iOS, operations and operation queues used threads. With the introduction of GCD, Apple completely changed the implementation of operations and operation queues by using GCD instead of threads.

This book is written for those of you who want to do what Apple suggests and what seems like the bright future for software development: migrating away from threads and allowing the operating system to take care of threads for you, by replacing thread programming with GCD.

Audience

In this book, I assume that you have a fairly basic understanding of the underlying technologies used in writing iOS and/or Mac OS X applications. We will not be discussing subjects related to Cocoa Touch or Cocoa. We will be using code that works, in principle and GCD layer, both with iOS and Mac OS X. Therefore, you will need to know the basics of Objective-C and your way around basic functionalities utilized by Core Foundation, such as string manipulation and arrays.

> O'Reilly's iOS 4 Programming Cookbook is a good source for more about object allocation, arrays, and UI-related code, in case you are looking to broaden your perspective toward iOS programming.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

> This icon signifies a tip, suggestion, or general note.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Concurrent Programming in Mac OS X and iOS* by Vandad Nahavandipoor (O'Reilly). Copyright 2011 Vandad Nahavandipoor, 9781449305635."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9781449305635

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Working with O'Reilly to write books has always been a pleasure and this book is not an exception. I must say I am very fortunate to have fantastic friends and a fantastic support team around me, for without them, I wouldn't be the person I am today and you wouldn't be reading this book.

Andy Oram and Brian Jepson have been incredibly supportive of my efforts and have, for the fourth time, given me a chance to reach out to those who want to be educated further in cutting-edge technologies such as Grand Central Dispatch.

I am grateful for my wonderful friends who have been a continuous source of inspiration and support. Thanks to my friends and colleagues Sushil Shirke, Shency Revindran, Angela Rory, Chris Harman, Natalie Szrajber, Simon Whitty, Shaun Puckrin, Gary McCarville, Mark Harris, and Kirk Pattinson.

I would also like to thank everybody from O'Reilly who has helped me so far with my sometimes-incredibly-annoying requests. Thanks to Sarah Schneider for helping me with SVN setup and other technical DocBook questions. Thanks to Rachel James for helping me manage readers' requests for my existing books. A big thank you goes to Betsy Waliszewski and Gretchen Giles of O'Reilly for arranging a three-day half-price offer on many O'Reilly titles to help with Japanese disaster relief. With all you wonderful readers' help, O'Reilly donated $200,000 to Japanese disaster relief in March 2011.

Last but not least, I would like to thank you for reading this book. Your belief in my work is what keeps me writing more books that help readers be more productive and creative.

CHAPTER 1
Introducing Block Objects

Block objects are *packages* of code that usually appear in the form of methods in Objective-C. Block objects, together with Grand Central Dispatch (GCD), create a harmonious environment in which you can deliver high-performance multithreaded apps in iOS and Mac OS X. What's so special about block objects and GCD, you might ask? It's simple: no more threads! All you have to do is to put your code in block objects and ask GCD to take care of the execution of that code for you.

In this chapter, you will learn the basics of block objects, followed by some more advanced subjects. You will understand everything you need to know about block objects before moving to the Grand Central Dispatch chapter. From my experience, the best way to learn block objects is through examples, so you will see a lot of them in this chapter. Make sure you try the examples for yourself in Xcode to really *get* the syntax of block objects.

Short Introduction to Block Objects

Block objects in Objective-C are what the programming field calls *first-class objects*. This means you can build code dynamically, pass a block object to a method as a parameter, and return a block object from a method. All of these things make it easier to choose what you want to do at runtime and change the activity of a program. In particular, block objects can be run in individual threads by GCD. Being Objective-C objects, block objects can be treated like any other object: you can retain them, release them, and so forth. Block objects can also be called *closures*.

> Block objects are sometimes referred to as *closures*.

Constructing block objects is similar to constructing traditional C functions, as we will see in "Constructing Block Objects and Their Syntax" on page 2. Block objects can have return values and can accept parameters. Block objects can be defined inline or treated as a separate block of code, similar to a C function. When created inline, the scope of variables accessible to block objects is considerably different from when a block object is implemented as a separate block of code.

GCD works with block objects. When performing tasks with GCD, you can pass a block object whose code can get executed synchronously or asynchronously, depending on which methods you use in GCD. Thus, you can create a block object that is responsible for downloading a URL passed to it as a parameter. That single block object can then be used in various places in your app synchronously or asynchronously, depending on how you would like to run it. You don't have to make the block object synchronous or asynchronous per se; you will simply call it with synchronous or asynchronous GCD methods and the block object will *just work*.

Block objects are quite new to programmers writing iOS and OS X apps. In fact, block objects are not as popular as threads yet, perhaps because their syntax is a bit different from pure Objective-C methods and more complicated. Nonetheless, block objects are enormously powerful and Apple is making a big push toward incorporating them into Apple libraries. You can already see these additions in classes such as `NSMutableArray`, where programmers can sort the array using a block object.

This chapter is dedicated entirely to constructing and using block objects in iOS and Mac OS X apps. I would like to stress that the only way to get used to block objects' syntax is to write a few of them for yourself. Have a look at the sample code in this chapter and try implementing your own block objects.

Constructing Block Objects and Their Syntax

Block objects can either be inline or coded as independent blocks of code. Let's start with the latter type. Suppose you have a method in Objective-C that accepts two integer values of type `NSInteger` and returns the difference of the two values, by subtracting one from the other, as an `NSInteger`:

```
- (NSInteger) subtract:(NSInteger)paramValue
                  from:(NSInteger)paramFrom{

  return paramFrom - paramValue;

}
```

That was very simple, wasn't it? Now let's translate this Objective-C code to a pure C function that provides the same functionality to get one step closer to learning the syntax of block objects:

```
NSInteger subtract(NSInteger paramValue, NSInteger paramFrom){

  return paramFrom - paramValue;

}
```

You can see that the C function is quite different in syntax from its Objective-C counterpart. Now let's have a look at how we could code the same function as a block object:

```
NSInteger (^subtract)(NSInteger, NSInteger) =
  ^(NSInteger paramValue, NSInteger paramFrom){

  return paramValue - paramValue;

};
```

Before I go into details about the syntax of block objects, let me show you a few more examples. Suppose we have a function in C that takes a parameter of type `NSUInteger` (an unsigned integer) and returns it as a string of type `NSString`. Here is how we implement this in C:

```
NSString* intToString (NSUInteger paramInteger){

  return [NSString stringWithFormat:@"%lu",
          (unsigned long)paramInteger];

}
```

> To learn about formatting strings with system-independent format specifiers in Objective-C, please refer to String Programming Guide, iOS Developer Library on Apple's website.

The block object equivalent of this C function is shown in Example 1-1.

Example 1-1. Example block object defined as function

```
NSString* (^intToString)(NSUInteger) = ^(NSUInteger paramInteger){
  NSString *result = [NSString stringWithFormat:@"%lu",
                      (unsigned long)paramInteger];

  return result;
};
```

The simplest form of an independent block object would be a block object that returns `void` and does not take any parameters in:

```
void (^simpleBlock)(void) = ^{
  /* Implement the block object here */
};
```

Block objects can be invoked in the exact same way as C functions. If they have any parameters, you pass the parameters to them like a C function and any return value can be retrieved exactly as you would retrieve a C function's return value. Here is an example:

```
NSString* (^intToString)(NSUInteger) = ^(NSUInteger paramInteger){
  NSString *result = [NSString stringWithFormat:@"%lu",
                      (unsigned long)paramInteger];
  return result;
};

- (void) callIntToString{

  NSString *string = intToString(10);
  NSLog(@"string = %@", string);

}
```

The `callIntToString` Objective-C method is calling the `intToString` block object by passing the value 10 as the only parameter to this block object and placing the return value of this block object in the `string` local variable.

Now that we know how to write block objects as independent blocks of code, let's have a look at passing block objects as parameters to Objective-C methods. We will have to think a bit abstractly to understand the goal of the following example.

Suppose we have an Objective-C method that accepts an integer and performs some kind of transformation on it, which may change depending on what else is happening in our program. We know that we'll have an integer as input and a string as output, but we'll leave the exact transformation up to a block object that can be different each time our method runs. This method, therefore, will accept as parameters both the integer to be transformed and the block that will transform it.

For our block object, we'll use the same `intToString` block object that we implemented earlier in Example 1-1. Now we need an Objective-C method that will accept an unsigned integer parameter and a block object as its parameter. The unsigned integer parameter is easy, but how do we tell our method that it has to accept a block object *of the same type* as the `intToString` block object? First we `typedef` the signature of the `intToString` block object, which tells the compiler what parameters our block object should accept:

```
typedef NSString* (^IntToStringConverter)(NSUInteger paramInteger);
```

This `typedef` just tells the compiler that block objects that accept an integer parameter and return a string can simply be represented by an identifier named `IntToStringConverter`. Now let's go ahead and write our Objective-C method that accepts both an integer and a block object of type `IntToStringConverter`:

```
- (NSString *) convertIntToString:(NSUInteger)paramInteger
                 usingBlockObject:(IntToStringConverter)paramBlockObject{

  return paramBlockObject(paramInteger);

}
```

All we have to do now is call the `convertIntToString:` method with our block object of choice (Example 1-2).

Example 1-2. Calling the block object in another method

```
- (void) doTheConversion{

  NSString *result = [self convertIntToString:123
                             usingBlockObject:intToString];

  NSLog(@"result = %@", result);

}
```

Now that we know something about independent block objects, let's turn to inline block objects. In the `doTheConversion` method we just saw, we passed the `intToString` block object as the parameter to the `convertIntToString:usingBlockObject:` method. What if we didn't have a block object ready to be passed to this method? Well, that wouldn't be a problem. As mentioned before, block objects are first-class functions and can be constructed at runtime. Let's have a look at an alternative implementation of the `doTheConversion` method (Example 1-3).

Example 1-3. Example block object defined as function

```
- (void) doTheConversion{

  IntToStringConverter inlineConverter = ^(NSUInteger paramInteger){
    NSString *result = [NSString stringWithFormat:@"%lu",
                        (unsigned long)paramInteger];
    return result;
  };

  NSString *result = [self convertIntToString:123
                             usingBlockObject:inlineConverter];

  NSLog(@"result = %@", result);

}
```

Compare Example 1-3 to the earlier Example 1-1. I have removed the initial code that provided the block object's signature, which consisted of a name and argument, `(^intToString)(NSUInteger)`. I left all the rest of the block object intact. It is now an anonymous object. But this doesn't mean I have no way to refer to the block object. I assign it using an equal sign to a type and a name: `IntToStringConverter inlineConverter`. Now I can use the data type to enforce proper use in methods, and use the name to actually pass the block object.

In addition to constructing block objects inline as just shown, we can construct a block object *while* passing it as a parameter:

```
- (void) doTheConversion{

  NSString *result =
  [self convertIntToString:123
          usingBlockObject:^NSString *(NSUInteger paramInteger) {

    NSString *result = [NSString stringWithFormat:@"%lu",
                        (unsigned long)paramInteger];
    return result;

  }];

  NSLog(@"result = %@", result);

}
```

Compare this example with Example 1-2. Both methods use a block object through the `usingBlockObject` syntax. But whereas the earlier version referred to a previously declared block object by name (`intToString`), this one simply creates a block object on the fly. In this code, we constructed an inline block object that gets passed to the `convertIntToString:usingBlockObject:` method as the second parameter.

I believe that at this point you know enough about block objects to be able to move to more interesting details, which we'll begin with in the following section.

Variables and Their Scope in Block Objects

Here is a brief summary of what you must know about variables in block objects:

- Local variables in block objects work exactly the same as in Objective-C methods.
- For inline block objects, local variables constitute not only variables defined within the block, but also the variables that have been defined in the method that implements that block object. (Examples will come shortly.)
- You *cannot* refer to `self` in independent block objects implemented in an Objective-C class. If you need to access `self`, you must pass that object to the block object as a parameter. We will see an example of this soon.
- You can refer to `self` in an inline block object only if `self` is present in the lexical scope inside which the block object is created.
- For inline block objects, local variables that are defined *inside* the block object's implementation can be read from and written to. In other words, the block object has read-write access to variables defined inside the block object's body.
- For inline block objects, variables local to the Objective-C method that implements that block can only be read from, not written to. There is an exception, though: a block object can write to such variables if they are defined with the `__block` storage type. We will see an example of this as well.

- Suppose you have an object of type `NSObject` and inside that object's implementation you are using a block object in conjunction with GCD. Inside this block object, you will have read-write access to declared properties of that `NSObject` inside which your block is implemented.
- You can access declared properties of your `NSObject` inside independent block objects *only if* you use the setter and getter methods of these properties. You cannot access declared properties of an object using dot notation inside an independent block object.

Let's first see how we can use variables that are local to the implementation of two block objects. One is an inline block object and the other an independent block object:

```
void (^independentBlockObject)(void) = ^(void){

  NSInteger localInteger = 10;

  NSLog(@"local integer = %lu", (unsigned long)localInteger);

  localInteger = 20;

  NSLog(@"local integer = %lu", (unsigned long)localInteger);

};
```

Invoking this block object, the values we assigned are printed to the console window:

```
local integer = 10
local integer = 20
```

So far, so good. Now let's have a look at inline block objects and variables that are local to them:

```
- (void) simpleMethod{

  NSUInteger outsideVariable = 10;

  NSMutableArray *array = [[NSMutableArray alloc]
                           initWithObjects:@"obj1",
                           @"obj2", nil];

  [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
    NSUInteger insideVariable = 20;

    NSLog(@"Outside variable = %lu", (unsigned long)outsideVariable);
    NSLog(@"Inside variable = %lu", (unsigned long)insideVariable);

    /* Return value for our block object */
    return NSOrderedSame;

  }];

  [array release];

}
```

> The `sortUsingComparator:` instance method of `NSMutableArray` attempts to sort a mutable array. The goal of this example code is just to demonstrate the use of local variables, so you don't have to know what this method actually does.

The block object can read and write its own `insideVariable` local variable. However, the block object has read-only access to the `outsideVariable` variable by default. In order to allow the block object to write to `outsideVariable`, we must prefix `outsideVariable` with the `__block` storage type:

```
- (void) simpleMethod{

  __block NSUInteger outsideVariable = 10;

  NSMutableArray *array = [[NSMutableArray alloc]
                           initWithObjects:@"obj1",
                           @"obj2", nil];


  [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {

     NSUInteger insideVariable = 20;
     outsideVariable = 30;

     NSLog(@"Outside variable = %lu", (unsigned long)outsideVariable);
     NSLog(@"Inside variable = %lu", (unsigned long)insideVariable);

     /* Return value for our block object */
     return NSOrderedSame;

   }];

  [array release];

}
```

Accessing `self` in inline block objects is fine as long as `self` is defined in the lexical scope inside which the inline block object is created. For instance, in this example, the block object will be able to access `self`, since `simpleMethod` is an instance method of an Objective-C class:

```
- (void) simpleMethod{

  NSMutableArray *array = [[NSMutableArray alloc]
                           initWithObjects:@"obj1",
                           @"obj2", nil];

  [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {

    NSLog(@"self = %@", self);

    /* Return value for our block object */
    return NSOrderedSame;
```

```
  }];

  [array release];

}
```

You cannot, without a change in your block object's implementation, access `self` in an independent block object. Attempting to compile this code will give you a compile-time error:

```
void (^incorrectBlockObject)(void) = ^{
  NSLog(@"self = %@", self); /* self is undefined here */
};
```

If you want to access `self` in an independent block object, simply pass the object that `self` represents as a parameter to your block object:

```
void (^correctBlockObject)(id) = ^(id self){

  NSLog(@"self = %@", self);

};

- (void) callCorrectBlockObject{

  correctBlockObject(self);

}
```

> You don't have to assign the name `self` to this parameter. You can simply call this parameter anything else. However, if you call this parameter `self`, you can simply grab your block object's code later and place it in an Objective-C method's implementation without having to change every instance of your variable's name to `self` for it to be understood by the compiler.

Let's have a look at declared properties and how block objects can access them. For inline block objects, you can use dot notation to read from or write to declared properties of `self`. For instance, let's say we have a declared property of type `NSString` called `stringProperty` in our class:

```
#import <UIKit/UIKit.h>

@interface GCDAppDelegate : NSObject <UIApplicationDelegate> {
@protected
  NSString    *stringProperty;
}

@property (nonatomic, retain) NSString    *stringProperty;

@end
```

Now we can simply access this property in an inline block object like so:

```
#import "GCDAppDelegate.h"

@implementation GCDAppDelegate

@synthesize stringProperty;

- (void) simpleMethod{

  NSMutableArray *array = [[NSMutableArray alloc]
                           initWithObjects:@"obj1",
                           @"obj2", nil];

  [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {

    NSLog(@"self = %@", self);

    self.stringProperty = @"Block Objects";

    NSLog(@"String property = %@", self.stringProperty);

    /* Return value for our block object */
    return NSOrderedSame;

  }];

  [array release];

}

- (void) dealloc{
  [stringProperty release];
  [super dealloc];
}

@end
```

In an independent block object, however, you cannot use dot notation to read from or write to a declared property:

```
void (^correctBlockObject)(id) = ^(id self){

  NSLog(@"self = %@", self);

  /* Should use setter method instead of this */
  self.stringProperty = @"Block Objects"; /* Compile-time Error */

  /* Should use getter method instead of this */
  NSLog(@"self.stringProperty = %@",
        self.stringProperty); /* Compile-time Error */

};
```

Instead of dot notation in this scenario, use the getter and the setter methods of this synthesized property:

```
void (^correctBlockObject)(id) = ^(id self){

  NSLog(@"self = %@", self);

  /* This will work fine */
  [self setStringProperty:@"Block Objects"];

  /* This will work fine as well */
  NSLog(@"self.stringProperty = %@",
        [self stringProperty]);

};
```

When it comes to inline block objects, there is one *very* important rule that you have to remember: inline block objects copy the value for the variables in their lexical scope. If you don't understand what that means, don't worry. Let's have a look at an example:

```
typedef void (^BlockWithNoParams)(void);

- (void) scopeTest{

  NSUInteger integerValue = 10;

  /*************** Definition of internal block object ***************/
  BlockWithNoParams myBlock = ^{
    NSLog(@"Integer value inside the block = %lu",
          (unsigned long)integerValue);
  };
  /*************** End definition of internal block object ***************/

  integerValue = 20;

  /* Call the block here after changing the
   value of the integerValue variable */
  myBlock();

  NSLog(@"Integer value outside the block = %lu",
        (unsigned long)integerValue);

}
```

We are declaring an integer local variable and initially assigning the value of 10 to it. We then implement our block object but *don't call the block object yet*. After the block object is *implemented*, we simply change the value of the local variable that the block object will later try to read when we call it. Right after changing the local variable's value to 20, we call the block object. You would expect the block object to print the value 20 for the variable, but it won't. It will print 10, as you can see here:

```
Integer value inside the block = 10
Integer value outside the block = 20
```

What's happening here is that the block object is keeping a read-only copy of the `integerValue` variable for itself right where the block is implemented. You might be thinking: why is the block object capturing a *read-only* value of the local variable integerValue? The answer is simple, and we've already learned it in this section. Unless prefixed with storage type __block, local variables in the lexical scope of a block object are just passed to the block object as read-only variables. Therefore, to change this behavior, we could change the implementation of our scopeTest method to prefix the integerValue variable with __block storage type, like so:

```
- (void) scopeTest{

  __block NSUInteger integerValue = 10;

  /************** Definition of internal block object **************/
  BlockWithNoParams myBlock = ^{
    NSLog(@"Integer value inside the block = %lu",
          (unsigned long)integerValue);
  };
  /************** End definition of internal block object **************/

  integerValue = 20;

  /* Call the block here after changing the
   value of the integerValue variable */
  myBlock();

  NSLog(@"Integer value outside the block = %lu",
        (unsigned long)integerValue);

}
```

Now if we get the results from the console window after the scopeTest method is called, we will see this:

```
Integer value inside the block = 20
Integer value outside the block = 20
```

This section should have given you sufficient information about using variables with block objects. I suggest that you write a few block objects and use variables inside them, assigning to them and reading from them, to get a better understanding of how block objects use variables. Keep coming back to this section if you forget the rules that govern variable access in block objects.

Invoking Block Objects

We've seen examples of invoking block objects in "Constructing Block Objects and Their Syntax" on page 2 and "Variables and Their Scope in Block Objects" on page 6. This section contains more concrete examples.

If you have an independent block object, you can simply invoke it just like you would invoke a C function:

```
void (^simpleBlock)(NSString *) = ^(NSString  *paramString){
  /* Implement the block object here and use the
   paramString parameter */
```

```
};

- (void) callSimpleBlock{

  simpleBlock(@"O'Reilly");

}
```

If you want to invoke an independent block object within another independent block object, follow the same instructions by invoking the new block object just as you would invoke a C method:

```
/*************** Definition of first block object ***************/
NSString *(^trimString)(NSString *) = ^(NSString *inputString){

  NSString *result = [inputString stringByTrimmingCharactersInSet:
                      [NSCharacterSet whitespaceCharacterSet]];
  return result;

};
/*************** End definition of first block object ***************/

/*************** Definition of second block object ***************/
NSString *(^trimWithOtherBlock)(NSString *) = ^(NSString *inputString){
  return trimString(inputString);
};
/*************** End definition of second block object ***************/

- (void) callTrimBlock{

  NSString *trimmedString = trimWithOtherBlock(@"   O'Reilly    ");
  NSLog(@"Trimmed string = %@", trimmedString);

}
```

In this example, go ahead and invoke the `callTrimBlock` Objective-C method:

```
[self callTrimBlock];
```

The `callTrimBlock` method will call the `trimWithOtherBlock` block object, and the `trimWithOtherBlock` block object will call the `trimString` block object in order to trim the given string. Trimming a string is an easy thing to do and can be done in one line of code, but this example code shows how you can call block objects within block objects.

In Chapter 2, you will learn how to invoke block objects using Grand Central Dispatch, synchronously or asynchronously, to unleash the real power of block objects.

Memory Management for Block Objects

iOS apps run in a reference-counted environment. That means every object has a retain count to ensure the Objective-C runtime keeps it as long as it might be used, and gets rid of it when no one can use it anymore. You can think of a retain count as the number of leashes on an animal. As long as there is at least one leash, the animal will stay where it is. If there are two leashes, the animal has to be unleashed twice to be released. As soon as all leashes are released, the animal is free. Substitute all occurrences of *animal* with *object* in the preceding sentences and you will understand how a reference-counted environment works. When we allocate an object in iOS, the retain count of that object becomes 1. Every allocation has to be paired with a release call invoked on the object to decrement the release count by 1. If you want to keep the object around in memory, you have to make sure you have retained that object so that its retain count is incremented by the runtime.

> For more information about memory management in iOS apps, please refer to iOS 4 Programming Cookbook (O'Reilly).

Block objects are objects as well, so they also can be copied, retained, and released. When writing an iOS app, you can simply treat block objects as normal objects and retain and release them as you would with other objects:

```
typedef NSString* (^StringTrimmingBlockObject)(NSString *paramString);

NSString* (^trimString)(NSString *) = ^(NSString *paramString){
  NSString *result = nil;

  result = [paramString
            stringByTrimmingCharactersInSet:
            [NSCharacterSet whitespaceCharacterSet]];

  return result;
};

- (void) callTrimString{

  StringTrimmingBlockObject trimStringCopy = Block_copy(trimString);

  NSString *trimmedString = trimStringCopy(@"   O'Reilly   ");

  NSLog(@"Trimmed string = %@", trimmedString);

  Block_release(trimStringCopy);

}
```

Use `Block_copy` on a block object to declare ownership of that block object for the period of time you wish to use it. While retaining ownership over a block object, you can be sure that iOS will not dispose of that block object and its memory. Once you are done with that block object, you must release ownership using `Block_release`.

If you are using block objects in your Mac OS X apps, you should follow the same rules, whether you are writing your app in a garbage-collected or a reference-counting environment. Here is the same example code from iOS, written for Mac OS X. You can compile it with and without garbage collection enabled for your project:

```
typedef NSString* (^StringTrimmingBlockObject)(NSString *paramString);

NSString* (^trimString)(NSString *) = ^(NSString *paramString){
  NSString *result = nil;

  result = [paramString
            stringByTrimmingCharactersInSet:
            [NSCharacterSet whitespaceCharacterSet]];

  return result;
};

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{

  StringTrimmingBlockObject trimmingBlockObject = Block_copy(trimString);

  NSString *trimmedString = trimmingBlockObject(@"   O'Reilly   ");

  NSLog(@"Trimmed string = %@", trimmedString);

  Block_release(trimmingBlockObject);

}
```

In iOS, you can also use autorelease block objects, like so:

```
NSString* (^trimString)(NSString *) = ^(NSString *paramString){
  NSString *result = nil;

  result = [paramString
            stringByTrimmingCharactersInSet:
            [NSCharacterSet whitespaceCharacterSet]];

  return result;
};

- (id) autoreleaseTrimStringBlockObject{

  return [trimString autorelease];

}
```

You can also define declared properties that hold a copy of a block object. Here is the *.h* file of our object that declares a property (`nonatomic, copy`) for a block object:

```
#import <UIKit/UIKit.h>

typedef NSString* (^StringTrimmingBlockObject)(NSString *paramString);

@interface GCDAppDelegate : NSObject <UIApplicationDelegate> {
@protected
  StringTrimmingBlockObject   trimmingBlock;
```

```
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, copy) StringTrimmingBlockObject   trimmingBlock;

@end
```

> This code is written inside the application delegate of a simple universal iOS app.

Now let's go ahead and implement our application's delegate object:

```
#import "GCDAppDelegate.h"

@implementation GCDAppDelegate

@synthesize window=_window;
@synthesize trimmingBlock;

NSString* (^trimString)(NSString *) = ^(NSString *paramString){
  NSString *result = nil;

  result = [paramString
            stringByTrimmingCharactersInSet:
            [NSCharacterSet whitespaceCharacterSet]];

  return result;
};

- (BOOL)            application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  self.trimmingBlock = trimString;

  NSString *trimmedString = self.trimmingBlock(@"   O'Reilly   ");

  NSLog(@"Trimmed string = %@", trimmedString);

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}

- (void)dealloc{
  [trimmingBlock release];
  [_window release];
  [super dealloc];
}

@end
```

What we want to achieve in this example is, first, to declare ownership over the `trimString` block object in our application delegate, and then to use that block object to trim a single string off its whitespaces.

> The `trimmingBlock` property is declared as `nonatomic`. This means that this property's thread-safeness must be managed by us, and we should make sure this property won't get accessed from more than one thread at a time. We won't really have to care about this at the moment as we are not doing anything fancy with threads right now. This property is also defined as `copy`, which tells the runtime to call the `copy` method on any object, including block objects, when we assign those objects to this property, as opposed to retaining those objects by calling the `retain` method on them.

As we saw before, the `trimString` block object accepts a string as its parameter, trims this string, and returns it to the caller. Inside the `application:didFinishLaunchingWithOptions:` instance method of our application delegate, we are simply using dot notation to assign the `trimString` block object to the `trimmingBlock` declared property. This means that the runtime will immediately call the `Block_copy` on the `trimString` block object and assign the resulting value to the `trimmingBlock` declared property. From this point on, until we release the block object, we have a copy of it in the `trimmingBlock` declared property.

Now we can use the `trimmingBlock` declared property to invoke the `trimString` block object, as shown in the following code:

```
NSString *trimmedString = self.trimmingBlock(@"   O'Reilly   ");
```

Once we are done, in the `dealloc` instance method of our object, we will release the `trimmingBlock` declared property by calling its `release` method.

With more insight into block objects and how they manage their variables and memory, it is finally time to move to Chapter 2 to learn about the wonder that is called Grand Central Dispatch. We will be using block objects with GCD a lot, so make sure you have really understood the material in this chapter before moving on to the next.

CHAPTER 2

Programming Grand Central Dispatch

Grand Central Dispatch, or GCD for short, is a low-level C API that works with block objects. The real use for GCD is to dispatch tasks to multiple cores without making you, the programmer, worry about which core is executing which task. On Mac OS X, multicore devices, including laptops, have been available to users for quite some time. With the introduction of multicore devices such as the iPad 2, programmers can write amazing multicore-aware multithreaded apps for iOS. See the preface for more background on the importance of multicores.

In Chapter 1 we learned how to use block objects. If you have not read that chapter, I strongly suggest that you do straight away, as GCD relies heavily on block objects and their dynamic nature. In this chapter, we will learn about really fun and interesting things that programmers can achieve with GCD in iOS and Mac OS X.

Short Introduction to Grand Central Dispatch

At the heart of GCD are dispatch queues. Dispatch queues, as we will see in "Different Types of Dispatch Queues" on page 21, are pools of threads managed by GCD on the host operating system, whether it is iOS or Mac OS X. You will not be working with these threads directly. You will just work with dispatch queues, dispatching *tasks* to these queues and asking the queues to invoke your tasks. GCD offers several options for running tasks: synchronously, asynchronously, after a certain delay, etc.

To start using GCD in your apps, you don't have to import any special library into your project. Apple has already incorporated GCD into various frameworks, including Core Foundation and Cocoa/Cocoa Touch. All methods and data types available in GCD start with a *dispatch_* keyword. For instance, `dispatch_async` allows you to dispatch a task on a queue for asynchronous execution, whereas `dispatch_after` allows you to run a block of code after a given delay.

Traditionally, programmers had to create their own threads to perform tasks in parallel. For instance, an iOS developer would create a thread similar to this to perform an operation 1000 times:

```
- (void) doCalculation{

  /* Do your calculation here */

}

- (void) calculationThreadEntry{

  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  NSUInteger counter = 0;

  while ([[NSThread currentThread] isCancelled] == NO){

    [self doCalculation];

    counter++;

    if (counter >= 1000){
      break;
    }

  }

  [pool release];

}

- (BOOL)              application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  /* Start the thread */
  [NSThread detachNewThreadSelector:@selector(calculationThreadEntry)
                           toTarget:self
                         withObject:nil];

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```

The programmer has to start the thread manually and then create the required structure for the thread (entry point, autorelease pool, and thread's main loop). When we write the same code with GCD, we really won't have to do much:

```
dispatch_queue_t queue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

size_t numberOfIterations = 1000;

dispatch_async(queue, ^(void) {
  dispatch_apply(numberOfIterations, queue, ^(size_t iteration){
    /* Perform the operation here */
  });
});
```

In this chapter, you will learn all there is to know about GCD and how to use it to write modern multithreaded apps for iOS and Mac OS X that will achieve blazing performance on multicore devices such as the iPad 2.

Different Types of Dispatch Queues

As mentioned in "Short Introduction to Grand Central Dispatch" on page 19, dispatch queues are pools of threads managed by GCD. We will be working with dispatch queues a lot, so please make sure that you fully understand the concept behind them. There are three types of dispatch queues:

Main Queue
: This queue performs all its tasks on the main thread, which is where Cocoa and Cocoa Touch require programmers to call all UI-related methods. Use the `dispatch_get_main_queue` function to retrieve the handle to the main queue.

Concurrent Queues
: These are queues that you can retrieve from GCD in order to execute asynchronous or synchronous tasks. Multiple concurrent queues can be executing multiple tasks in parallel, without breaking a sweat. No more thread management, yippee! Use the `dispatch_get_global_queue` function to retrieve the handle to a concurrent queue.

Serial Queues
: These are queues that, no matter whether you submit synchronous or asynchronous tasks to them, will always execute their tasks in a first-in-first-out (FIFO) fashion, meaning that they can only execute one block object at a time. However, they do *not* run on the main thread and therefore are perfect for a series of tasks that have to be executed in strict order without blocking the main thread. Use the `dispatch_queue_create` function to create a serial queue. Once you are done with the queue, you must release it using the `dispatch_release` function.

At any moment during the lifetime of your application, you can use multiple dispatch queues at the same time. Your system has only one main queue, but you can create as many serial dispatch queues as you want, within reason, of course, for whatever functionality you require for your app. You can also retrieve multiple concurrent queues and dispatch your tasks to them. Tasks can be handed to dispatch queues in two forms:

block objects or C functions, as we will see in "Dispatching Tasks to Grand Central Dispatch" on page 22.

Dispatching Tasks to Grand Central Dispatch

There are two ways to submit tasks to dispatch queues:

- Block Objects (see Chapter 1)
- C functions

Block objects are the best way of utilizing GCD and its enormous power. Some GCD functions have been extended to allow programmers to use C functions instead of block objects. However, the truth is that only a limited set of GCD functions allow programmers to use C functions, so please do read the chapter about block objects (Chapter 1) before proceeding with this chapter.

C functions that have to be supplied to various GCD functions should be of type `dispatch_function_t`, which is defined as follows in the Apple libraries:

```
typedef void (*dispatch_function_t)(void *);
```

So if we want to create a function named, for instance, `myGCDFunction`, we would have to implement it in this way:

```
void myGCDFunction(void * paraContext){

  /* Do the work here */

}
```

> The `paraContext` parameter refers to the context that GCD allows programmers to pass to their C functions when they dispatch tasks to them. We will learn about this shortly.

Block objects that get passed to GCD functions don't always follow the same structure. Some must accept parameters and some shouldn't, but none of the block objects submitted to GCD return a value.

In the next three sections you will learn how to submit tasks to GCD for execution whether they are in the form of block objects or C functions.

Performing UI-Related Tasks

UI-related tasks have to be performed on the main thread, so the main queue is the only candidate for UI task execution in GCD. We can use the `dispatch_get_main_queue` function to get the handle to the main dispatch queue.

There are two ways of dispatching tasks to the main queue. Both are asynchronous, letting your program continue even when the task is not yet executed:

`dispatch_async` *function*
: Executes a block object on a dispatch queue.

`dispatch_async_f` *function*
: Executes a C function on a dispatch queue.

> The `dispatch_sync` method *cannot* be called on the main queue because it will block the thread indefinitely and cause your application to deadlock. All tasks submitted to the main queue through GCD must be submitted asynchronously.

Let's have a look at using the `dispatch_async` function. It accepts two parameters:

Dispatch queue handle
: The dispatch queue on which the task has to be executed.

Block object
: The block object to be sent to the dispatch queue for asynchronous execution.

Here is an example. This code will display an alert, in iOS, to the user, using the main queue:

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();

dispatch_async(mainQueue, ^(void) {

  [[[[UIAlertView alloc]
     initWithTitle:NSLocalizedString(@"GCD", nil)
     message:NSLocalizedString(@"GCD is amazing!", nil)
     delegate:nil
     cancelButtonTitle:NSLocalizedString(@"OK", nil)
     otherButtonTitles:nil, nil] autorelease] show];

});
```

> As you've noticed, the `dispatch_async` GCD function has no parameters or return value. The block object that is submitted to this function must gather its own data in order to complete its task. In the code snippet that we just saw, the alert view has all the values that it needs to finish its task. However, this might not always be the case. In such instances, you must make sure the block object submitted to GCD has access in its scope to all the values that it requires.

Running this app in iOS Simulator, the user will get results similar to those shown in Figure 2-1.

Figure 2-1. An alert displayed using asynchronous GCD calls

This might not be that impressive. In fact, it is not impressive at all if you think about it. So what makes the main queue truly interesting? The answer is simple: when you are getting the maximum performance from GCD to do some heavy calculation on concurrent or serial threads, you might want to display the results to your user or move a component on the screen. For that, you *must* use the main queue, because it is UI-related work. The functions shown in this section are the *only* way to get out of a serial or a concurrent queue while still utilizing GCD to update your UI, so you can imagine how important it is.

Instead of submitting a block object for execution on the main queue, you can submit a C function object. Submit all UI-related C functions for execution in GCD to the `dispatch_async_f` function. We can get the same results as we got in Figure 2-1, using C functions instead of block objects, with a few adjustments to our code.

As mentioned before, with the `dispatch_async_f` function, we can submit a pointer to an application-defined context, which can then be used by the C function that gets called. So here is the plan: let's create a structure that holds values such as an alert view's title, message, and cancel-button's title. When our app starts, we will put all the values in this structure and pass it to our C function to display. Here is how we are defining our structure:

```
typedef struct{
  char *title;
  char *message;
  char *cancelButtonTitle;
} AlertViewData;
```

Now let's go and implement a C function that we will later call with GCD. This C function should expect a parameter of type `void *`, which we will then typecast to `AlertViewData *`. In other words, we expect the caller of this function to pass us a reference to the data for our alert view, encapsulated inside the `AlertViewData` structure:

```
void displayAlertView(void *paramContext){

  AlertViewData *alertData = (AlertViewData *)paramContext;

  NSString *title =
  [NSString stringWithUTF8String:alertData->title];

  NSString *message =
  [NSString stringWithUTF8String:alertData->message];

  NSString *cancelButtonTitle =
  [NSString stringWithUTF8String:alertData->cancelButtonTitle];

  [[[[UIAlertView alloc] initWithTitle:title
                               message:message
                              delegate:nil
                     cancelButtonTitle:cancelButtonTitle
                     otherButtonTitles:nil, nil] autorelease] show];

  free(alertData);

}
```

> The reason we are `free`ing the context passed to us in here instead of in the caller is that the caller is going to execute this C function asynchronously and cannot know when our C function will finish executing. Therefore, the caller has to `malloc` enough space for the `AlertViewData` context and our `displayAlertView` C function has to free that space.

And now let's call the `displayAlertView` function on the main queue and pass the context (the structure that holds the alert view's data) to it:

```
- (BOOL)             application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  dispatch_queue_t mainQueue = dispatch_get_main_queue();

  AlertViewData *context = (AlertViewData *)
    malloc(sizeof(AlertViewData));

  if (context != NULL){
    context->title = "GCD";
    context->message = "GCD is amazing.";
    context->cancelButtonTitle = "OK";

    dispatch_async_f(mainQueue,
                     (void *)context,
                     displayAlertView);
  }

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```

If you invoke the `currentThread` class method of the `NSThread` class, you will find out that the block objects or the C functions you dispatch to the main queue are indeed running on the main thread:

```
- (BOOL)             application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  dispatch_queue_t mainQueue = dispatch_get_main_queue();

  dispatch_async(mainQueue, ^(void) {
    NSLog(@"Current thread = %@", [NSThread currentThread]);
    NSLog(@"Main thread = %@", [NSThread mainThread]);
  });

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```

The output of this code would be similar to that shown here:

```
Current thread = <NSThread: 0x4b0e4e0>{name = (null), num = 1}
Main thread = <NSThread: 0x4b0e4e0>{name = (null), num = 1}
```

Now that you know how to perform UI-related tasks using GCD, it is time we moved to other subjects, such as performing tasks in parallel using concurrent queues (see "Performing Non-UI-Related Tasks Synchronously" on page 27 and "Performing Non-UI-Related Tasks Asynchronously" on page 29) and mixing our code with UI-related code if need be.

Performing Non-UI-Related Tasks Synchronously

There are times when you want to perform tasks that have nothing to do with the UI or interact with the UI as well as doing other tasks that take up a lot of time. For instance, you might want to download an image and display it to the user after it is downloaded. The downloading process has absolutely nothing to do with the UI.

For any task that doesn't involve the UI, you can use global concurrent queues in GCD. These allow either synchronous or asynchronous execution. But synchronous execution does *not* mean your program waits for the code to finish before continuing. It simply means that the concurrent queue will wait until your task has finished before it continues to the next block of code on the queue. When you put a block object on a concurrent queue, your own program *always* continues right away without waiting for the queue to execute the code. This is because concurrent queues, as their name implies, run their code on threads other than the main thread. (There is one exception to this: when a task is submitted to a concurrent or a serial queue using the `dispatch_sync` function, iOS will, if possible, run the task on the *current* thread, which *might* be the main thread, depending on where the code path is at the moment. This is an optimization that has been programmed on GCD, as we shall soon see.)

If you submit a task to a concurrent queue synchronously, and at the same time submit another synchronous task to *another* concurrent queue, these two synchronous tasks will run asynchronously in relation to each other because they are running two *different concurrent queues*. It's important to understand this because sometimes, as we'll see, you want to make sure task A finishes before task B starts. To ensure that, submit them synchronously to the *same* queue.

You can perform synchronous tasks on a dispatch queue using the `dispatch_sync` function. All you have to do is to provide it with the handle of the queue that has to run the task and a block of code to execute on that queue.

Let's look at an example. It prints the integers 1 to 1000 twice, one complete sequence after the other, without blocking the main thread. We can create a block object that does the counting for us and synchronously call the same block object twice:

```
void (^printFrom1To1000)(void) = ^{

  NSUInteger counter = 0;
  for (counter = 1;
       counter <= 1000;
       counter++){

    NSLog(@"Counter = %lu - Thread = %@",
          (unsigned long)counter,
          [NSThread currentThread]);

  }

};
```

Now let's go and invoke this block object using GCD:

```
dispatch_queue_t concurrentQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_sync(concurrentQueue, printFrom1To1000);
dispatch_sync(concurrentQueue, printFrom1To1000);
```

If you run this code, you might notice the counting taking place on the main thread, even though you've asked a concurrent queue to execute the task. It turns out this is an optimization by GCD. The `dispatch_sync` function will use the current thread—the thread you're using when you dispatch the task—whenever possible, as a part of an optimization that has been programmed into GCD. Here is what Apple says about it:

> As an optimization, this function invokes the block on the current thread when possible.
>
> —Grand Central Dispatch (GCD) Reference

To execute a C function instead of a block object, synchronously, on a dispatch queue, use the `dispatch_sync_f` function. Let's simply translate the code we've written for the `printFrom1To1000` block object to its equivalent C function, like so:

```
void printFrom1To1000(void *paramContext){

  NSUInteger counter = 0;
  for (counter = 1;
       counter <= 1000;
       counter++){

    NSLog(@"Counter = %lu - Thread = %@",
          (unsigned long)counter,
          [NSThread currentThread]);

  }

}
```

And now we can use the `dispatch_sync_f` function to execute the `printFrom1To1000` function on a concurrent queue, as demonstrated here:

```
dispatch_queue_t concurrentQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_sync_f(concurrentQueue,
                NULL,
                printFrom1To1000);

dispatch_sync_f(concurrentQueue,
                NULL,
                printFrom1To1000);
```

The first parameter of the `dispatch_get_global_queue` function specifies the priority of the concurrent queue that GCD has to retrieve for the programmer. The higher the priority, the more CPU timeslices will be provided to the code getting executed on that queue. You can use any of these values for the first parameter to the `dispatch_get_global_queue` function:

`DISPATCH_QUEUE_PRIORITY_LOW`
: Fewer timeslices will be applied to your task than normal tasks.

`DISPATCH_QUEUE_PRIORITY_DEFAULT`
: The default system priority for code execution will be applied to your task.

`DISPATCH_QUEUE_PRIORITY_HIGH`
: More timeslices will be applied to your task than normal tasks.

> The second parameter of the `dispatch_get_global_queue` function is reserved and you should always pass the value 0 to it.

In this section you saw how you can dispatch tasks to concurrent queues for synchronous execution. The next section shows asynchronous execution on concurrent queues, while "Constructing Your Own Dispatch Queues" on page 43 will show how to execute tasks synchronously and asynchronously on serial queues that you create for your applications.

Performing Non-UI-Related Tasks Asynchronously

This is where GCD can show its true power: executing blocks of code asynchronously on the main, serial, or concurrent queues. I promise that, by the end of this section, you will be completely convinced GCD is the future of multithread applications, completely replacing threads in modern apps.

In order to execute asynchronous tasks on a dispatch queue, you must use one of these functions:

`dispatch_async`
: Submits a block object to a dispatch queue (both specified by parameters) for asynchronous execution.

`dispatch_async_f`
: Submits a C function to a dispatch queue, along with a context reference (all three specified by parameters), for asynchronous execution.

Let's have a look at a real example. We'll write an iOS app that is able to download an image from a URL on the Internet. After the download is finished, the app should display the image to the user. Here is the plan and how we will use what we've learned so far about GCD in order to accomplish it:

1. We are going to launch a block object asynchronously on a concurrent queue.
2. Once in this block, we will launch another block object *synchronously*, using the `dispatch_sync` function, to download the image from a URL. Synchronously downloading a URL from an asynchronous code block holds up just the queue running the synchronous function, not the main thread. The whole operation still is asynchronous when we look at it from the main thread's perspective. All we care about is that we are not blocking the main thread while downloading our image.
3. Right after the image is downloaded, we will synchronously execute a block object on the *main queue* (see "Performing UI-Related Tasks" on page 22) in order to display the image to the user on the UI.

The skeleton for our plan is as simple as this:

```
dispatch_queue_t concurrentQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(concurrentQueue, ^{

  __block UIImage *image = nil;

  dispatch_sync(concurrentQueue, ^{
    /* Download the image here */
  });

  dispatch_sync(dispatch_get_main_queue(), ^{
    /* Show the image to the user here on the main queue*/
  });

});
```

The second `dispatch_sync` call, which displays the image, will be executed on the queue after the first synchronous call, which downloads our image. That's exactly what we want, because we *have* to wait for the image to be fully downloaded before we can display it to the user. So after the image is downloaded, we execute the second block object, but this time on the main queue.

Let's download the image and display it to the user now. We will do this in the `viewDidAppear:` instance method of a view controller displayed in an iPhone app:

```
- (void) viewDidAppear:(BOOL)paramAnimated{

  dispatch_queue_t concurrentQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

  dispatch_async(concurrentQueue, ^{

    __block UIImage *image = nil;

    dispatch_sync(concurrentQueue, ^{
      /* Download the image here */

      /* iPad's image from Apple's website. Wrap it into two
```

```
     lines as the URL is too long to fit into one line */
    NSString *urlAsString = @"http://images.apple.com/mobileme/features"\
                            "/images/ipad_findyouripad_20100518.jpg";

    NSURL *url = [NSURL URLWithString:urlAsString];

    NSURLRequest *urlRequest = [NSURLRequest requestWithURL:url];

    NSError *downloadError = nil;
    NSData *imageData = [NSURLConnection
                         sendSynchronousRequest:urlRequest
                         returningResponse:nil
                         error:&downloadError];

    if (downloadError == nil &&
        imageData != nil){

      image = [UIImage imageWithData:imageData];
      /* We have the image. We can use it now */

    }
    else if (downloadError != nil){
      NSLog(@"Error happened = %@", downloadError);
    } else {
      NSLog(@"No data could get downloaded from the URL.");
    }

  });

  dispatch_sync(dispatch_get_main_queue(), ^{
    /* Show the image to the user here on the main queue*/

    if (image != nil){
      /* Create the image view here */
      UIImageView *imageView = [[UIImageView alloc]
                                initWithFrame:self.view.bounds];

      /* Set the image */
      [imageView setImage:image];

      /* Make sure the image is not scaled incorrectly */
      [imageView setContentMode:UIViewContentModeScaleAspectFit];

      /* Add the image to this view controller's view */
      [self.view addSubview:imageView];

      /* Release the image view */
      [imageView release];

    } else {
      NSLog(@"Image isn't downloaded. Nothing to display.");
    }

  });
```

```
    });

}
```

As you can see in Figure 2-2, we have successfully downloaded our image and also created an image view to display the image to the user on the UI.

Figure 2-2. Downloading and displaying images to users, using GCD

Let's move on to another example. Let's say that we have an array of 10,000 random numbers that have been stored in a file on disk and we want to load this array into memory, sort the numbers in an ascending fashion (with the smallest number appearing first in the list), and then display the list to the user. The control used for the display depends on whether you are coding this for iOS (ideally, you'd use an instance of `UITableView`) or Mac OS X (`NSTableView` would be a good candidate). Since we don't have an array, why don't we create the array first, then load it, and finally display it?

Here are two methods that will help us find the location where we want to save the array of 10,000 random numbers on disk on the device:

```
- (NSString *) fileLocation{

  /* Get the document folder(s) */
  NSArray *folders =
  NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
```

```
                                           NSUserDomainMask,
                                           YES);

  /* Did we find anything? */
  if ([folders count] == 0){
    return nil;
  }

  /* Get the first folder */
  NSString *documentsFolder = [folders objectAtIndex:0];

  /* Append the file name to the end of the documents path */
  return [documentsFolder
          stringByAppendingPathComponent:@"list.txt"];

}

- (BOOL) hasFileAlreadyBeenCreated{

  BOOL result = NO;

  NSFileManager *fileManager = [[NSFileManager alloc] init];
  if ([fileManager fileExistsAtPath:[self fileLocation]] == YES){
    result = YES;
  }
  [fileManager release];

  return result;
}
```

Now the important part: we want to save an array of 10,000 random numbers to disk *if and only if* we have not created this array before on disk. If we have, we will load the array from disk immediately. If we have not created this array before on disk, we will first create it and then move on to loading it from disk. At the end, if the array was successfully read from disk, we will sort the array in an ascending fashion and finally display the results to the user on the UI. I will leave displaying the results to the user up to you:

```
  dispatch_queue_t concurrentQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

  /* If we have not already saved an array of 10,000
   random numbers to the disk before, generate these numbers now
   and then save them to the disk in an array */
  dispatch_async(concurrentQueue, ^{

    NSUInteger numberOfValuesRequired = 10000;

    if ([self hasFileAlreadyBeenCreated] == NO){
      dispatch_sync(concurrentQueue, ^{

        NSMutableArray *arrayOfRandomNumbers =
        [[NSMutableArray alloc] initWithCapacity:numberOfValuesRequired];
```

```
      NSUInteger counter = 0;
      for (counter = 0;
           counter < numberOfValuesRequired;
           counter++){
        unsigned int randomNumber =
        arc4random() % ((unsigned int)RAND_MAX + 1);

        [arrayOfRandomNumbers addObject:
         [NSNumber numberWithUnsignedInt:randomNumber]];
      }

      /* Now let's write the array to disk */
      [arrayOfRandomNumbers writeToFile:[self fileLocation]
                             atomically:YES];

      [arrayOfRandomNumbers release];

    });
  }

  __block NSMutableArray *randomNumbers = nil;

  /* Read the numbers from disk and sort them in an
   ascending fashion */
  dispatch_sync(concurrentQueue, ^{

    /* If the file has now been created, we have to read it */
    if ([self hasFileAlreadyBeenCreated] == YES){
      randomNumbers = [[NSMutableArray alloc]
                       initWithContentsOfFile:[self fileLocation]];

      /* Now sort the numbers */
      [randomNumbers sortUsingComparator:
       ^NSComparisonResult(id obj1, id obj2) {

        NSNumber *number1 = (NSNumber *)obj1;
        NSNumber *number2 = (NSNumber *)obj2;
        return [number1 compare:number2];

       }];
    }
  });


  dispatch_async(dispatch_get_main_queue(), ^{
    if ([randomNumbers count] > 0){
      /* Refresh the UI here using the numbers in the
       randomNumbers array */
    }
    [randomNumbers release];
  });

});
```

There is a lot more to GCD than synchronous and asynchronous block or function execution. In "Running a Group of Tasks Together" on page 40 you will learn how to group block objects together and prepare them for execution on a dispatch queue. I also suggest that you have a look at "Performing Tasks After a Delay" on page 35 and "Performing a Task at Most Once" on page 38 to learn about other functionalities that GCD is capable of providing to programmers.

Performing Tasks After a Delay

With Core Foundation, you can invoke a selector in an object after a given period of time, using the `performSelector:withObject:afterDelay:` method of the `NSObject` class. Here is an example:

```
- (void) printString:(NSString *)paramString{

  NSLog(@"%@", paramString);

}

- (BOOL)              application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  @selector(performSelector:withObject:afterDelay:)

  [self performSelector:@selector(printString:)
             withObject:@"Grand Central Dispatch"
             afterDelay:3.0];

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```

In this example we are asking the runtime to call the `printString:` method after 3 seconds of delay. We can do the same thing in GCD using the `dispatch_after` and `dispatch_after_f` functions, each of which is described here:

`dispatch_after`
: Dispatches a block object to a dispatch queue after a given period of time, specified in nanoseconds. These are the parameters that this function requires:

  Delay in nanoseconds
  : The number of nanoseconds GCD has to wait on a given dispatch queue (specified by the second parameter) before it executes the given block object (specified by the third parameter).

  Dispatch queue
  : The dispatch queue on which the block object (specified by the third parameter) has to be executed after the given delay (specified by the first parameter).

Block object

The block object to be invoked after the specified number of nanoseconds on the given dispatch queue. This block object should have no return value and should accept no parameters (see "Constructing Block Objects and Their Syntax" on page 2).

`dispatch_after_f`

Dispatches a C function to GCD for execution after a given period of time, specified in nanoseconds. This function accepts four parameters:

Delay in nanoseconds

The number of nanoseconds GCD has to wait on a given dispatch queue (specified by the second parameter) before it executes the given function (specified by the fourth parameter).

Dispatch queue

The dispatch queue on which the C function (specified by the fourth parameter) has to be executed after the given delay (specified by the first parameter).

Context

The memory address of a value in the heap to be passed to the C function (for an example, see "Performing UI-Related Tasks" on page 22).

C function

The address of the C function that has to be executed after a certain period of time (specified by the first parameter) on the given dispatch queue (specified by the second parameter).

> Although the delays are in nanoseconds, it is up to iOS to decide the granularity of dispatch delay, and this delay might not be as precise as what you hope when you specify a value in nanoseconds.

Let's have a look at an example for `dispatch_after` first:

```
double delayInSeconds = 2.0;

dispatch_time_t delayInNanoSeconds =
  dispatch_time(DISPATCH_TIME_NOW, delayInSeconds * NSEC_PER_SEC);

dispatch_queue_t concurrentQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_after(delayInNanoSeconds, concurrentQueue, ^(void){
  /* Perform your operations here */
});
```

As you can see, the nanoseconds delay parameter for both the `dispatch_after` and `dispatch_after_f` functions has to be of type `dispatch_time_t`, which is an abstract representation of absolute time. To get the value for this parameter, you can use the `dispatch_time` function as demonstrated in this sample code. Here are the parameters that you can pass to the `dispatch_time` function:

Base time
: If this value was denoted with *B* and the delta parameter was denoted with *D*, the resulting time from this function would be equal to *B+D*. You can set this parameter's value to `DISPATCH_TIME_NOW` to denote *now* as the base time and then specify the delta from now using the delta parameter.

Delta to add to base time
: This parameter is the nanoseconds that will get added to the base time parameter to create the result of this function.

For example, to denote a time 3 seconds from now, you could write your code like so:

```
dispatch_time_t delay =
dispatch_time(DISPATCH_TIME_NOW, 3.0f * NSEC_PER_SEC);
```

Or to denote half a second from now:

```
dispatch_time_t delay =
dispatch_time(DISPATCH_TIME_NOW, (1.0 / 2.0f) * NSEC_PER_SEC);
```

Now let's have a look at how we can use the `dispatch_after_f` function:

```
void processSomething(void *paramContext){

  /* Do your processing here */
  NSLog(@"Processing...");

}

- (BOOL)              application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  double delayInSeconds = 2.0;

  dispatch_time_t delayInNanoSeconds =
    dispatch_time(DISPATCH_TIME_NOW, delayInSeconds * NSEC_PER_SEC);

  dispatch_queue_t concurrentQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

  dispatch_after_f(delayInNanoSeconds,
                    concurrentQueue,
                   NULL,
                   processSomething);

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```

Performing a Task at Most Once

Allocating and initializing a singleton is one of the tasks that has to happen exactly once during the lifetime of an app. I am sure you know of other scenarios where you had to make sure a piece of code was executed only once during the lifetime of your application.

GCD lets you specify an identifier for a piece of code when you attempt to execute it. If GCD detects that this identifier has been passed to the framework before, it won't execute that block of code again. The function that allows you to do this is `dispatch_once`, which accepts two parameters:

Token
: A token of type `dispatch_once_t` that holds the token generated by GCD when the block of code is executed for the first time. If you want a piece of code to be executed at most once, you must specify the same token to this method whenever it is invoked in the app. We will see an example of this soon.

Block object
: The block object to get executed at most once. This block object returns no values and accepts no parameters.

> `dispatch_once` always executes its task on the current queue being used by the code that issues the call, be it a serial queue, a concurrent queue, or the main queue.

Here is an example:

```
static dispatch_once_t onceToken;

void (^executedOnlyOnce)(void) = ^{

  static NSUInteger numberOfEntries = 0;
  numberOfEntries++;
  NSLog(@"Executed %lu time(s)", (unsigned long)numberOfEntries);

};

- (BOOL)            application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  dispatch_queue_t concurrentQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

  dispatch_once(&onceToken, ^{
    dispatch_async(concurrentQueue,
                   executedOnlyOnce);
  });
```

```
  dispatch_once(&onceToken, ^{
    dispatch_async(concurrentQueue,
                   executedOnlyOnce);
  });

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```

As you can see, although we are attempting to invoke the `executedOnlyOnce` block object twice, using the `dispatch_once` function, in reality GCD is only executing this block object once, since the identifier passed to the `dispatch_once` function is the same both times.

Apple, in its Cocoa Fundamentals Guide, shows programmers how to create a singleton. However, we can change this model to make use of GCD and the `dispatch_once` function in order to initialize a shared instance of an object, like so:

```
#import "MySingleton.h"

@implementation MySingleton

static MySingleton *sharedMySingleton = NULL;

+ (MySingleton *) sharedInstance{

  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, ^{
    if (sharedMySingleton == NULL){
      sharedMySingleton = [[super allocWithZone:NULL] init];
    }
  });

  return sharedMySingleton;
}

+ (id) allocWithZone:(NSZone *)paramZone{
  return [[self sharedInstance] retain];

}

- (id) copyWithZone:(NSZone *)paramZone{
  return self;
}

- (void) release{
  /* Do nothing */
}

- (id) autorelease{
  return self;
}

- (NSUInteger) retainCount{
```

```
    return NSUIntegerMax;
}

- (id) retain{
    return self;
}

@end
```

Running a Group of Tasks Together

GCD lets us create *groups*, which allow you to place your tasks in one place, run all of them, and get a notification at the end from GCD. This has many valuable applications. For instance, suppose you have a UI-based app and want to reload the components on your UI. You have a table view, a scroll view, and an image view. You want to reload the contents of these components using these methods:

```
- (void) reloadTableView{
    /* Reload the table view here */
    NSLog(@"%s", __FUNCTION__);
}

- (void) reloadScrollView{
    /* Do the work here */
    NSLog(@"%s", __FUNCTION__);
}

- (void) reloadImageView{
    /* Reload the image view here */
    NSLog(@"%s", __FUNCTION__);
}
```

At the moment, these methods are empty, but later you can put the relevant UI code in them. Now we want to call these three methods, one after the other, and we want to know when GCD has finished calling these methods so that we can display a message to the user. For this, we should be using a group. You should know about four functions when working with groups in GCD:

`dispatch_group_create`
: Creates a group handle. Once you are done with this group handle, you should dispose of it using the `dispatch_release` function.

`dispatch_group_async`
: Submits a block of code for execution on a group. You must specify the dispatch queue on which the block of code has to be executed *as well as* the group to which this block of code belongs.

`dispatch_group_notify`
: Allows you to submit a block object that should be executed once all tasks added to the group for execution have finished their work. This function also allows you to specify the dispatch queue on which that block object has to be executed.

`dispatch_release`
: Use this function to dispose of any dispatch groups that you create using the `dispatch_group_create` function.

Let's have a look at an example. As explained, in our example we want to invoke the `reloadTableView`, `reloadScrollView`, and `reloadImageView` methods one after the other and then display a message to the user once we are done. We can utilize GCD's powerful grouping facilities in order to accomplish this:

```
dispatch_group_t taskGroup =  dispatch_group_create();
dispatch_queue_t mainQueue =  dispatch_get_main_queue();

/* Reload the table view on the main queue */
dispatch_group_async(taskGroup, mainQueue, ^{
  [self reloadTableView];
});

/* Reload the scroll view on the main queue */
dispatch_group_async(taskGroup, mainQueue, ^{
  [self reloadScrollView];
});

/* Reload the image view on the main queue */
dispatch_group_async(taskGroup, mainQueue, ^{
  [self reloadImageView];
});

/* At the end when we are done, dispatch the following block */
dispatch_group_notify(taskGroup, mainQueue, ^{
  /* Do some processing here */

  [[[[UIAlertView alloc] initWithTitle:@"Finished"
                               message:@"All tasks are finished"
                              delegate:nil
                     cancelButtonTitle:@"OK"
                     otherButtonTitles:nil, nil] autorelease] show];

});

/* We are done with the group */
dispatch_release(taskGroup);
```

In addition to `dispatch_group_async`, you can also dispatch asynchronous C functions to a dispatch group using the `dispatch_group_async_f` function.

> `GCDAppDelegate` is simply the name of the class from which this example is taken. We have to use this class name in order to typecast a context object so that the compiler will understand our commands.

Like so:

```
- (void) reloadTableView{
  /* Reload the table view here */
  NSLog(@"%s", __FUNCTION__);
}

- (void) reloadScrollView{
  /* Do the work here */
  NSLog(@"%s", __FUNCTION__);
}

- (void) reloadImageView{
  /* Reload the image view here */
  NSLog(@"%s", __FUNCTION__);
}

void reloadAllComponents(void *context){

  GCDAppDelegate *self = (GCDAppDelegate *)context;
  [self reloadTableView];
  [self reloadScrollView];
  [self reloadImageView];

}

- (BOOL)              application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  dispatch_group_t taskGroup =  dispatch_group_create();
  dispatch_queue_t mainQueue =  dispatch_get_main_queue();

  dispatch_group_async_f(taskGroup,
                         mainQueue,
                         (void *)self,
                         reloadAllComponents);

  /* At the end when we are done, dispatch the following block */
  dispatch_group_notify(taskGroup, mainQueue, ^{
    /* Do some processing here */

    [[[[UIAlertView alloc] initWithTitle:@"Finished"
                                 message:@"All tasks are finished"
                                delegate:nil
                       cancelButtonTitle:@"OK"
                       otherButtonTitles:nil, nil] autorelease] show];

  });

  /* We are done with the group */
  dispatch_release(taskGroup);

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```

Since the `dispatch_group_async_f` function accepts a C function as the block of code to be executed, the C function must have a reference to `self` to be able to invoke instance methods of the current object in which the C function is implemented. That is the reason behind passing `self` as the context pointer in the `dispatch_group_async_f` function. For more information about contexts and C functions, please refer to "Performing UI-Related Tasks" on page 22.

Once all the given tasks are finished, the user will see a result similar to that shown in Figure 2-3.

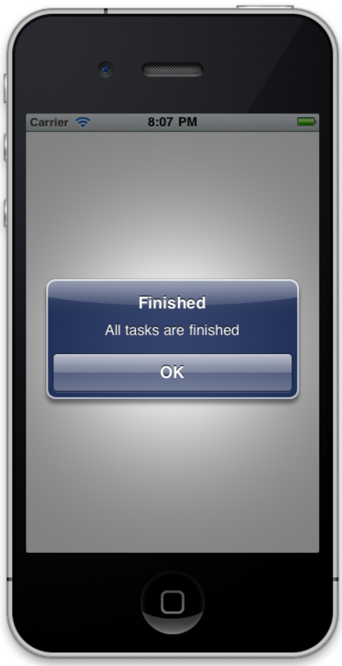

Figure 2-3. Managing a group of tasks with GCD

Constructing Your Own Dispatch Queues

With GCD, you can create your own serial dispatch queues (see "Different Types of Dispatch Queues" on page 21 for serial queues). Serial dispatch queues run their tasks in a first-in-first-out (FIFO) fashion. The asynchronous tasks on serial queues will *not* be performed on the main thread, however, making serial queues highly desirable for concurrent FIFO tasks.

All synchronous tasks submitted to a serial queue will be executed on the current thread being used by the code that is submitting the task, whenever possible. But asynchronous tasks submitted to a serial queue will always be executed on a thread other than the main thread.

We'll use the `dispatch_queue_create` function to create serial queues. The first parameter in this function is a C string (`char *`) that will uniquely identify that serial queue in the *system*. The reason I am emphasizing *system* is because this identifier is a system-wide identifier, meaning that if your app creates a new serial queue with the identifier of *`serialQueue1`* and somebody else's app does the same, the results of creating a new serial queue with the same name are undefined by GCD. Because of this, Apple strongly recommends that you use a reverse DNS format for identifiers. Reverse DNS identifiers are usually constructed in this way: com.*`COMPANY`*.*`PRODUCT`*.*`IDENTIFIER`*. For instance, I could create two serial queues and assign these names to them:

```
com.pixolity.GCD.serialQueue1
com.pixolity.GCD.serialQueue2
```

After you've created your serial queue, you can start dispatching tasks to it using the various GCD functions you've learned in this book. Once you are done with the serial dispatch queue that you've just created, you *must* dispose of it using the `dispatch_release` function.

Would you like to see an example? I thought so!

```
dispatch_queue_t firstSerialQueue =
  dispatch_queue_create("com.pixolity.GCD.serialQueue1", 0);

dispatch_async(firstSerialQueue, ^{
  NSUInteger counter = 0;
  for (counter = 0;
       counter < 5;
       counter++){
    NSLog(@"First iteration, counter = %lu", (unsigned long)counter);
  }
});

dispatch_async(firstSerialQueue, ^{
  NSUInteger counter = 0;
  for (counter = 0;
       counter < 5;
       counter++){
    NSLog(@"Second iteration, counter = %lu", (unsigned long)counter);
  }
});

dispatch_async(firstSerialQueue, ^{
  NSUInteger counter = 0;
  for (counter = 0;
       counter < 5;
       counter++){
    NSLog(@"Third iteration, counter = %lu", (unsigned long)counter);
  }
```

```
});

dispatch_release(firstSerialQueue);
```

If you run this code and have a look at the output printed to the console window, you will see results similar to these:

```
First iteration, counter = 0
First iteration, counter = 1
First iteration, counter = 2
First iteration, counter = 3
First iteration, counter = 4
Second iteration, counter = 0
Second iteration, counter = 1
Second iteration, counter = 2
Second iteration, counter = 3
Second iteration, counter = 4
Third iteration, counter = 0
Third iteration, counter = 1
Third iteration, counter = 2
Third iteration, counter = 3
Third iteration, counter = 4
```

It's obvious that although we dispatched our block objects asynchronously to the serial queue, the queue has executed their code in a FIFO fashion. We can modify the same sample code to make use of `dispatch_async_f` function instead of the `dispatch_async` function, like so:

```
void firstIteration(void *paramContext){
  NSUInteger counter = 0;
  for (counter = 0;
       counter < 5;
       counter++){
    NSLog(@"First iteration, counter = %lu", (unsigned long)counter);
  }
}

void secondIteration(void *paramContext){
  NSUInteger counter = 0;
  for (counter = 0;
       counter < 5;
       counter++){
    NSLog(@"Second iteration, counter = %lu", (unsigned long)counter);
  }
}

void thirdIteration(void *paramContext){
  NSUInteger counter = 0;
  for (counter = 0;
       counter < 5;
       counter++){
    NSLog(@"Third iteration, counter = %lu", (unsigned long)counter);
  }
}
```

```
- (BOOL)            application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  dispatch_queue_t firstSerialQueue =
    dispatch_queue_create("com.pixolity.GCD.serialQueue1", 0);

  dispatch_async_f(firstSerialQueue, NULL, firstIteration);
  dispatch_async_f(firstSerialQueue, NULL, secondIteration);
  dispatch_async_f(firstSerialQueue, NULL, thirdIteration);

  dispatch_release(firstSerialQueue);

  // Override point for customization after application launch.
  [self.window makeKeyAndVisible];
  return YES;
}
```