

SAMPLE CHAPTER

Objective-C Fundamentals



Christopher K. Fairbairn
Johannes Fahrenkrug
Collin Ruffenach

 MANNING



Objective-C Fundamentals

by Christopher K. Fairbairn
Johannes Fahrenkrug
Collin Ruffenach

Chapter 13

brief contents

PART 1	GETTING STARTED WITH OBJECTIVE-C.....	1
1	■ Building your first iOS application	3
2	■ Data types, variables, and constants	28
3	■ An introduction to objects	55
4	■ Storing data in collections	74
PART 2	BUILDING YOUR OWN OBJECTS	95
5	■ Creating classes	97
6	■ Extending classes	124
7	■ Protocols	144
8	■ Dynamic typing and runtime type information	163
9	■ Memory management	177
PART 3	MAKING MAXIMUM USE OF FRAMEWORK FUNCTIONALITY	201
10	■ Error and exception handling	203
11	■ Key-Value Coding and NSPredicate	212
12	■ Reading and writing application data	228
13	■ Blocks and Grand Central Dispatch	257
14	■ Debugging techniques	276

13

Blocks and Grand Central Dispatch

This chapter covers

- Understanding block syntax
- Handling memory management
- Using blocks in system frameworks
- Learning Grand Central Dispatch
- Performing work asynchronously

Have you ever been to New York City's Grand Central Terminal? It's a huge place with trains arriving and departing constantly. Imagine having to manage such an operation by hand—Which train should come in on which track? Which one should change tracks at what time? Which engine should be attached to which train?—and so on and so forth. You'd probably quit your job after 30 minutes and go back to programming!

Being a programmer, however, you might find yourself in a similar situation when you have to write a multithreaded application with different parts of your application being executed at the same time: you must make sure you have enough resources to run another thread; you must make sure that multiple threads can't manipulate the same data at the same time; you must somehow handle the case of one thread being dependent on other threads, and so on. It's a headache. And it's error prone, just like trying to run Grand Central Terminal by yourself.

Thankfully, Apple introduced a technology called Grand Central Dispatch (GCD) in Mac OS X 10.6 and brought it over to iOS devices in iOS 4. GCD dramatically simplifies multithreaded programming because it takes care of all the complicated heavy lifting. Along with GCD, Apple added a new language feature to C: blocks. Blocks are just that: blocks of code, or anonymous functions, if you will. GCD and blocks make for a very powerful duo. You can now write little pieces of code and hand them over to GCD to be executed in a parallel thread without any of the pain multithreaded programming normally causes. Parallel execution has never been easier, and with GCD there's no longer an excuse not to do it.

Before we dig any deeper into GCD, let's examine blocks: how to create them, how to run them, and what to watch out for.

13.1 *The syntax of blocks*

The syntax of blocks might scare you at first—it scared us! But we'll hold your hand as we decrypt and demystify it together, so don't fret.

Let's first look at a simple block example in the following listing.

Listing 13.1 Simple block example

```
int (^myMultiplier)(int, int) = ^int (int a, int b){
    return a * b;
};

int result = myMultiplier(7, 8);
```

We told you it would look a little scary. This listing creates a new block that takes two integers as arguments, returns an integer, and stores it in a variable called `myMultiplier`. Then the block gets executed, and the result of 56 gets stored in the integer variable called `result`.

The listing has two parts: the variable declaration and the block literal (the part after the equals sign). Let's look at both in depth (see figure 13.1).

You can think of a block variable declaration as a C function declaration because the syntax is almost identical. The difference is that the block name is enclosed in parentheses and has to be introduced by a caret (^). The following listing might clarify it even more.

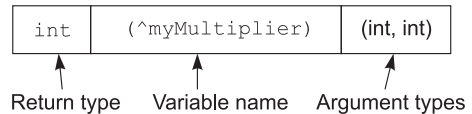


Figure 13.1 The syntax of a block variable declaration

Listing 13.2 Declaring a block variable and assigning to it

```
int (^myMultiplier) (int, int);

myMultiplier = ^int (int a, int b){
    return a * b;
};

myMultiplier(4, 2);
```

What if you want to declare multiple block variables of the same type (meaning taking the same arguments and having the same return type)? The following listing illustrates the wrong way and the right way to do it.

Listing 13.3 Using typedef to declare a block type for reuse

```
int (^myTimes2Multiplier) (int);
int (^myTimes5Multiplier) (int);
int (^myTimes10Multiplier) (int);

myTimes2Multiplier = ^(int a) { return a * 2; };
myTimes5Multiplier = ^(int a) { return a * 5; };
myTimes10Multiplier = ^(int a) { return a * 10; };

typedef int (^MultiplierBlock) (int);

MultiplierBlock myX2Multi = ^(int a) { return a * 2; };
MultiplierBlock myX5Multi = ^(int a) { return a * 5; };
MultiplierBlock myX10Multi = ^(int a) { return a * 10; };
```

The wrong way is to declare each block variable the long way. Although the code will compile and work fine, it isn't good style to do it this way, plus it might get difficult to maintain this kind of code later. The right way to use typedef is to define your own block type. It works exactly the same way as declaring a block variable, but this time the name in the parentheses is treated as a type name, not as a variable name. That way, you can declare as many blocks of the type `MultiplierBlock` as you like.

With the block variable declaration out of the way, let's look at the more interesting part, the part you'll use much more often: block literals. In the previous code examples, you saw a lot of block literals, but let's take a closer look at their syntax now (see figure 13.2).

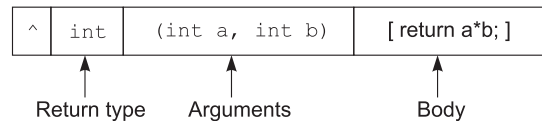


Figure 13.2 The syntax of a block literal

A block literal always starts with the caret (^), followed by the return type, the arguments, and finally the body—the actual code—in curly braces. Both the return type and the arguments are optional. If the return type is void, you can omit it altogether, and the same applies to arguments. The return type is also optional if it can be inferred automatically by the compiler through the `return` statement in the block's body. The next listing shows some valid long and short ways to write block literals.

Listing 13.4 Some long and short ways to write block literals

```
^void (void) { NSLog(@"Hello Block!"); };
^{ NSLog(@"Hello Block!"); };

^int { return 2001; };
^{ return 2001; };

^int (int a, int b) { return a + b; };
^(int a, int b) { return a + b; };
```

You see that it's convenient to omit unnecessary void arguments or return types in block literals. Note that if you try to run the code in listing 13.4, it won't do anything because you never actually execute these blocks. It's like declaring a list of functions that are never called. Blocks are executed the same way functions are: by adding a pair of opening and a closing parentheses to them. Normally, you'd assign a block literal to a variable and then execute the block using the variable (as you saw in listings 13.1 and 13.2). You can also execute block literals that don't take any arguments directly. The following listing shows both approaches.

Listing 13.5 Executing blocks

```
int (^myMultiplier)(int, int) = ^int (int a, int b){
    return a * b;
};

myMultiplier(7, 8);

^{ NSLog(@"Hello Block!"); }();
```

So far, blocks don't seem that different from functions—but they're very different in one special way.

13.1.1 *Blocks are closures*

Blocks have access to the variables that are available or defined in their lexical scope. What does that mean? Let's look at an example in the following listing.

Listing 13.6 Capturing variables

```
void (^myBlock) (void);

int year = 2525;

myBlock = ^{ NSLog(@"In the year %i", year); };
myBlock();
```

You can access the local variable `year` from inside the block. Technically, though, you're not accessing the variable. Instead, the variable is copied or frozen into the block by the time you create it. The following listing illustrates frozen variables.

Listing 13.7 Captured variables are frozen

```
void (^myBlock) (void);

int year = 2525;

myBlock = ^{ NSLog(@"In the year %i", year); };
myBlock();

year = 1984;

myBlock();
```

Changing the value of a variable after it's captured by a block doesn't affect the frozen copy in the block. But what if you want it to do that? Or what if you want to also

change the value from inside the block? Then you need to use the `__block` storage type, which practically marks the variables inside a block as mutable. The effect is that changes made to the variable outside the block are picked up inside the block, and vice versa (see the following listing).

Listing 13.8 The `__block` storage type

```
void (^myBlock) (void);

__block int year = 2525;
__block int runs = 0;

myBlock = ^{
    NSLog(@"In the year %i", year);
    runs++;
};

myBlock();

year = 1984;

myBlock();

NSLog(@"%i runs.", runs);
```

The interesting thing is that the local variables that are captured by a block live on even when the function or method they're defined in ends. That's a very powerful feature that we explore in more depth later in this chapter. For now, let's at least look at an example in the following listing.

Listing 13.9 Captured variables survive the end of a function

```
typedef void (^MyTestBlock) (void);

MyTestBlock createBlock() {
    int year = 2525;

    MyTestBlock myBlock = ^{
        NSLog(@"In the year %i", year);
    };

    return Block_copy(myBlock);
}

void runTheBlock() {
    MyTestBlock block = createBlock();

    block();

    Block_release(block);
}
```

Listing 13.9 clearly shows that the local variable `year` gets captured in the block and is still available in the block after the `createBlock` function returns. Also notice the use of `Block_copy` and `Block_release`, which brings us to our next topic.

13.1.2 *Blocks and memory management*

Blocks start their life on the stack, just like any other local variable in a function or method. If you want to use your block after the destruction of the scope in which it was declared (for example, after a function returns as in listing 13.9), you must use `Block_copy` to copy it to the heap. To avoid memory leaks, you must always use `Block_release` to release any block that you've copied with `Block_copy` when you don't need it anymore.

In Objective-C, blocks are also always Objective-C objects, so you can send them the familiar `copy` and `release` messages too.

What about objects that are captured by a block? In Objective-C all objects referenced inside a block are automatically sent a `retain` message. When the block gets released, all those objects are sent a `release` message. The only exceptions are objects with the `__block` storage type: they aren't automatically retained and released. When objects that are instance variables are referenced inside a block, the owning object instead of the instance object is sent a `retain` message. The following listing should make this clearer.

Listing 13.10 Automatic retain and release

```
typedef void (^SimpleBlock)(void);
@interface MyBlockTest : NSObject
{
    NSMutableArray *things;
}
- (void)runMemoryTest;
- (SimpleBlock)makeBlock;
@end
@implementation MyBlockTest
- (id)init {
    if ((self = [super init])) {
        things = [[NSMutableArray alloc] init];
        NSLog(@"1) retain count: %i", [self retainCount]);
    }
    return self;
}
- (SimpleBlock)makeBlock {
    __block MyBlockTest *mySelf = self;
    SimpleBlock block = ^{
        [things addObject:@"Mr. Horse"];
        NSLog(@"2) retain count: %i", [mySelf retainCount]);
    };
    return Block_copy(block);
}
- (void)dealloc {
    [things release];
}
```

← **SimpleBlock type definition**

← **ivar used inside of a block**

← **Pointer to MyTestBlock**

← **"things" ivar of current MyTestBlock**

← **Print retainCount with non-auto-retained reference**

```

    [super dealloc];
}
- (void)runMemoryTest {
    SimpleBlock block = [self makeBlock];
    block();
    Block_release(block);
    NSLog(@"3) retain count: %i", [self retainCount]);
}
@end

```

When you create an instance of the class `MyBlockTest` and run its `runMemoryTest` method, you'll see this result in the console:

- 1 retain count: 1
- 2 retain count: 2
- 3 retain count: 1

Let's examine what's happening here. You have an instance variable called `things`. In the `makeBlock` method, you create a reference to the current instance of `MyBlockTest` with the `__block` storage type. Why? You don't want the block to retain `self` because you use `self` in the `NSLog` statement inside the block, you want the block to retain `self` because you use one of its instance variables, `things`. Next, you reference `things` in the block and print the current `retainCount` of the object that owns `things` to the console. Finally, you return a copy of the block.

In the `runMemoryTest` method, you call the `makeBlock` method, run the returned block, and release it. Finally you print the `retainCount` again. This example demonstrates that your instance of `MyBlockTest` has been automatically retained because you used one of its instance variables—`things`—in a block that you copied to the heap. To make this even clearer, comment out the first line of the block (`[things addObject...]`) and run it again. You'll see that the retain count of the `MyBlockTest` instance is always 1. Because you're no longer referencing any of its instance variables inside the block, the instance of `MyBlockTest` is no longer automatically retained.

A final caveat you about working with blocks: A block literal (`^{...}`) is the memory address of a stack-local data structure representing the block. That means that the scope of that data structure is its enclosing statement (for example, a `for` loop or the body of an `if` statement). Why is that important to know? Because you'll enter a world of pain if you write code similar to the following listing.

Listing 13.11 Be careful about block literal scopes

```

void (^myBlock) (void);
if (true) {
    myBlock = ^{
        NSLog(@"I will die right away.");
    };
} else {

```

```

    myBlock = Block_copy(^{
        NSLog(@"I will live on.");
    });
}
myBlock();

```

Remember that block literals are valid only in the scope in which they are defined. If you want to use them outside of that scope, you must copy them to the heap with `Block_copy`. If you don't, your program will crash.

Now that you have a fundamental understanding of blocks, let's look at some common places where you'll encounter them in Cocoa Touch.

13.1.3 Block-based APIs in Apple's iOS frameworks

A great and growing number of Apple's framework classes take blocks as parameters, often greatly simplifying and reducing the amount of code you have to write. Typically, blocks are used as completion, error, and notification handlers, for sorting and enumeration, and for view animations and transitions.

In this section we look at a few simple but practical examples so you can get familiar with how blocks are used in the system frameworks.

The following listing shows how you can easily invoke a block for each line in a string.

Listing 13.12 Enumerating every line in a string with a block

```

NSString *string = @"Soylent\nGreen\nis\npeople";

[string enumerateLinesUsingBlock:
    ^(NSString *line, BOOL *stop) {
        NSLog(@"Line: %@", line);
    }];

```

You can surely guess what the output will look like. Pretty nifty, right?

Listing 13.13 demonstrates the use of two block-based APIs. The first one, `objectsPassingTest:`, invokes a given block for each object in a set. The block then returns either YES or NO for each object, and the method returns a new set consisting of all the objects that passed the test. The second one, `enumerateObjectsUsingBlock:`, invokes the given block once for every object in the set. It's basically equivalent to a for loop.

Listing 13.13 Filtering and enumeration using blocks

```

NSSet *set = [NSSet setWithObjects:@"a", @"b", @"cat",
    @"c", @"mouse", @"ox", @"d", nil];

NSSet *longStrings =
    [set objectsPassingTest:
        ^BOOL (id obj, BOOL *stop) {
            return [obj length] > 1;
        }];

```

```
[longStrings enumerateObjectsUsingBlock:
    ^(id obj, BOOL *stop) {
        NSLog(@"string: %@", obj);
    }];
```

Running the code in listing 13.13 will write the words `cat`, `mouse`, and `ox` to the console because they passed the test of being longer than one character.

The example in the following listing shows how to use a block as a notification handler.

Listing 13.14 Using a block as a notification handler

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[[[UIDevice currentDevice]
 beginGeneratingDeviceOrientationNotifications];

[nc addObserverForName:
    UIDeviceOrientationDidChangeNotification
    object:nil
    queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *notif){
        UIDeviceOrientation orientation;
        orientation = [[UIDevice currentDevice] orientation];
        if (UIDeviceOrientationIsPortrait(orientation)) {
            NSLog(@"portrait");
        } else {
            NSLog(@"landscape");
        }
    }];
```

Listing 13.14 first gets a reference to the default notification center and then tells the current device instance to start sending notifications when the device's orientation has changed. Finally, a block is added to the notification center as an observer for the orientation change notification. Don't worry about the use of `NSOperationQueue` here—we cover that later. The great thing about using blocks as handlers for various events is that you have the code that handles the event right there. You don't have to search for some target method somewhere else in your code. Your code is much more readable, less cluttered, more concise, and easier to understand.

It's clear how versatile blocks are and that they will undoubtedly make your work easier. Next we talk about a very special area in which blocks really shine: asynchronous and parallel execution.

13.2 Performing work asynchronously

Performing work asynchronously means doing multiple things at the same time—or at least making it seem as if you do.

To illustrate, think of a big supermarket with 20 employees who can work the register but only a single register is open. What a disaster! The employees would fight over the register, and the customers would get furious because they have to wait in long lines. It's much better when the store has 20 cash registers open: 20 customers can be

helped at the same time, everything moves a lot quicker, customers don't get mad, and everyone has something to do. In your iOS app, you don't want to have only a single cash register open—your customers would get mad too. Certain tasks, such as downloading data from the internet, can take a long time. Performing such tasks synchronously—one after another—would block your application and render it unresponsive to the user until the tasks are done. You don't want that. Instead, you want your application to stay responsive at all times and to perform tasks—especially long-running ones—asynchronously, notifying your user or updating the UI once the task has finished. GCD in connection with blocks makes this extremely easy to do.

13.2.1 Meet GCD

GCD manages a pool of threads and safely runs blocks on those threads depending on how many system resources are available. You don't have to worry about any of the thread management: GCD does all the work for you.

To explain how GCD works, let's go back to our analogy of New York City's Grand Central Terminal. Everyone knows that trains are made up of cars and that they run on tracks. A train station often has multiple tracks so that multiple trains can enter and leave the station at the same time. In GCD, blocks are the cars that make up a train. And dispatch queues are the tracks that these trains run on. GCD comes with four prebuilt queues: three global queues with low, default, and high priority and one main queue that corresponds with your application's main thread. GCD also allows you to build your own dispatch queues and run blocks on them. Both the main dispatch queue and the dispatch queues that you create on your own are serial queues. The blocks you put on a serial queue are executed one after the other, first in, first out (FIFO). Blocks that you put on the same serial queue are guaranteed to never run concurrently, but blocks on different serial queues do run concurrently, just like multiple trains on different tracks can run parallel to each other.

The only exceptions are the three concurrent global queues: they do run blocks concurrently. They start them in the order they were added to the queue, but they don't wait for one block to finish before they start running the next one.

All this might still seem a bit confusing and theoretical, so let's look at some code.

13.2.2 GCD fundamentals

The following listing shows how to run an anonymous block on the default priority global queue.

Listing 13.15 Running a block on a global queue

```
dispatch_async(  
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),  
    ^{  
        NSLog(@"Hello GCD!");  
    });
```

The function `dispatch_async` takes two arguments: the queue the block should be run on and the block itself. `dispatch_get_global_queue` does just that: it returns one of the three global queues (low, default, or high priority). The second parameter should always be 0 because it's reserved for future use. The `dispatch_async` function returns right away and dispatches the block to GCD to be run on the given queue. Pretty easy, right?

You can also create your own serial dispatch queues, as shown in the following listing.

Listing 13.16 Creating your own serial dispatch queue

```
dispatch_queue_t queue;
queue = dispatch_queue_create("com.springenwerk.Test", NULL);

dispatch_async(queue, ^{
    NSLog(@"Hello from my own queue!");
});

dispatch_release(queue);
```

The function `dispatch_queue_create` takes a name and a second parameter, which should always be `NULL` because it's reserved for future use. To avoid naming conflicts, you should use your reverse domain name to name the serial dispatch queues you create. You then pass them to `dispatch_async` just like any other queue. Because dispatch queues are reference-counted objects, you have to release them to avoid memory leaks.

That's all the basics you need to get up and running with GCD. That wasn't too bad, was it?

It might still seem like dry theory to you, though, so let's build a small application that uses GCD and shows how all this works in real life.

Real estate agents like to show their clients pictures of beautiful homes in the area they're interested in. That's exactly what you'll build: an application called `RealEstateViewer` that lets you search for images of real estate in any location of your choice.

13.2.3 Building `RealEstateViewer`

Create a new Window-based application in Xcode and call it `RealEstateViewer`. Add a new `UITableViewController` subclass to your project (Cocoa Touch Class > `UIViewController` subclass with the `UITableViewController` subclass selected). Call it `ImageTableViewController`. Next, include it in your application delegate and set the window's view to the `ImageTableViewController`'s view, as shown in listings 13.17 and 13.18 (see figure 13.3).

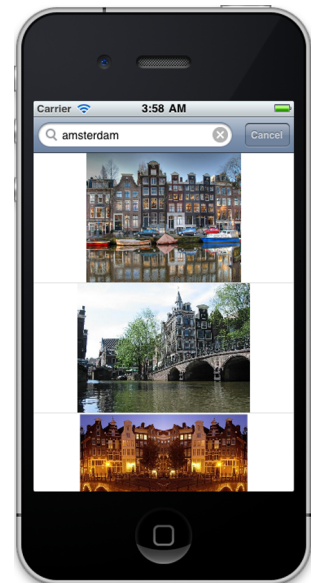


Figure 13.3 The finished `RealEstateViewer` application

Listing 13.17 `RealEstateViewerAppDelegate.h`

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

@interface RealEstateViewerAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    ImageTableViewController *imageTableViewController;
}

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

@end
```

Be sure to also implement the following listing.

Listing 13.18 `RealEstateViewerAppDelegate.m`

```
#import "RealEstateViewerAppDelegate.h"

@implementation RealEstateViewerAppDelegate
@synthesize window;

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

    imageTableViewController = [[ImageTableViewController alloc] init];

    [window addSubview:[imageTableViewController view]];
    [window makeKeyAndVisible];

    return YES;
}

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

@end
```

That's all the boilerplate code you have to write. The rest of the action takes place inside the `ImageTableViewController`. Because you'll be talking to Google's Image Search API, which returns data in the JSON format, you must add Stig Brautaset's JSON framework to your application. That sounds more complicated than it is: just download it from <http://stig.github.com/json-framework/> and copy all the files from the `Classes` folder to your application's `Classes` folder (or you can take the files from the source code for this chapter).

Now you'll add a search bar to the top of the table, a few delegate methods, an ivar to hold your search results, and a little bit of code to perform the image search. The next two listings show you what your `ImageTableViewController` should look like at this point.

Listing 13.19 ImageTableViewController.h

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

@interface ImageTableViewController : UITableViewController
    <UISearchBarDelegate> {

    NSArray *results;
}

@property (nonatomic, retain) NSArray *results;

@end
```

Listing 13.20 ImageTableViewController.m

```
#import "ImageTableViewController.h"
#import "JSON.h"

@implementation ImageTableViewController
@synthesize results;

#pragma mark -
#pragma mark Initialization

- (id)initWithStyle:(UITableViewStyle)style {
    if ((self = [super initWithStyle:style])) {
        results = [NSArray array];

        UISearchBar *searchBar =
            [[UISearchBar alloc]
             initWithFrame:CGRectMakeMake(0, 0,
             self.tableView.frame.size.width, 0)];
        searchBar.delegate = self;
        searchBar.showsCancelButton = YES;
        [searchBar sizeToFit];

        self.tableView.tableHeaderView = searchBar;
        [searchBar release];

        self.tableView.rowHeight = 160;
    }
    return self;
}

#pragma mark -
#pragma mark UISearchBarDelegate methods

- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar {
    NSLog(@"Searching for: %@", searchBar.text);
    NSString *api = @"http://ajax.googleapis.com/ajax/"
        "services/search/images?v=1.0&rsz=large&q=";
    NSString *urlString =
        [NSString
         stringWithFormat:@"%real%20estate%20%",
         api,
         [searchBar.text
          stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding]];
    NSURL *url = [NSURL URLWithString:urlString];
```

← Create UISearchBar;
set as headerView

← Create a URL
with a search
string

```

[NSThread sleepForTimeInterval:1.5];
NSData *data = [NSData dataWithContentsOfURL:url];
NSString *res = [[NSString alloc] initWithData:data
                encoding:NSUTF8StringEncoding];

self.results = [[[res JSONValue] objectForKey:@"responseData"]
                objectForKey:@"results"];

[res release];
[searchBar resignFirstResponder];
[self.tableView reloadData];
}

- (void)searchBarCancelButtonClicked:(UISearchBar *)searchBar {
    [searchBar resignFirstResponder];
}

#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {

    return [results count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier]
                autorelease];
    } else {
        for (UIView *view in cell.contentView.subviews) {
            [view removeFromSuperview];
        }
    }

    UIImage *image =
        [[results objectAtIndex:indexPath.row] objectForKey:@"image"];

    if (!image) {
        image = [UIImage imageWithData:
                [NSData dataWithContentsOfURL:
                 [NSURL URLWithString:
                  [[results objectAtIndex:indexPath.row]
                   objectForKey:@"unescapedUrl"]]]];
    }
}

```

Sleep for 1.5 seconds; fake slow network

Parse JSON data; assign to results ivar

Use blocking method to load network results

Attempt to get cached image for requested row

Load image with a blocking method

```

        [[results objectAtIndex:indexPath.row]
         setValue:image forKey:@"image"];
    }

    UIImageView *imageView =
        [[[UIImageView alloc] initWithImage:image] autorelease];

    imageView.contentMode = UIViewContentModeScaleAspectFit;
    imageView.autoresizingMask =
        UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight;
    imageView.frame = cell.contentView.frame;

    [cell.contentView addSubview:imageView];

    return cell;
}

#pragma mark -
#pragma mark Memory management

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

@end

```

← Cache the image

Now build and run your application. You should have a fully functional real estate image searcher. But you'll notice immediately how terrible the user experience is: when you put in a search term, the whole application freezes for a few seconds, and when you scroll down, it freezes a couple of more times. Completely unacceptable! What's happening here? This code is doing a horribly wrong thing: executing long-running blocking tasks—getting the search results and downloading the images—on the main thread. The main thread must always be free to handle UI updates and incoming events. That's why you should never do anything “expensive” on the main thread. How can you use blocks and GCD to fix these problems? You can put both the code that queries the image search API and the code that downloads an image into blocks and then hand those blocks to GCD. GCD will execute them in a parallel thread and thus not block the main thread.

13.2.4 Making the image search asynchronous

The following listing shows what the GCD-based asynchronous image search looks like.

Listing 13.21 Asynchronous image search with GCD

```

- (void)searchBarButtonClicked:(UISearchBar *)searchBar {
    NSLog(@"Searching for: %@", searchBar.text);
    NSString *api = @"http://ajax.googleapis.com/ajax/"
        "services/search/images?v=1.0&rsz=large&q=";
    NSString *urlString = [NSString
        stringWithFormat:@"%@real%20estate%20%@",
        api,
        [searchBar.text
        stringByAddingPercentEscapesUsingEncoding:

```

```

       :NSUTF8StringEncoding]];
NSURL *url = [NSURL URLWithString:urlString];

// get the global default priority queue
dispatch_queue_t defQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

void (^imageAPIBlock)(void);

imageAPIBlock = ^{
    [NSThread sleepForTimeInterval:1.5];

    NSData *data = [NSData dataWithContentsOfURL:url];

    NSString *res = [[NSString alloc]
        initWithData:data
        encoding:NSUTF8StringEncoding];

    NSArray *newResults =
        [[res valueForKey:@"responseData"]
        valueForKey:@"results"];

    [res release];

    dispatch_async(dispatch_get_main_queue(), ^{
        self.results = newResults;
        [self.tableView reloadData];
    });
};

dispatch_async(defQueue, imageAPIBlock);

[searchBar resignFirstResponder];
}

```

What does this code do? First it references one of the three global concurrent queues: the default priority queue. Then it declares a block that performs the network communication with the image search API and takes care of the JSON parsing. When it's done, it calls `dispatch_async` again (that's right, you can call `dispatch_async` from inside a block). The target is the main queue, which corresponds to the application's main thread (the one that takes care of the UI and events). You pass in an anonymous block that sets the new results and tells the tableview to reload. Why don't you do this right in the `imageAPIBlock`? For two reasons: First, UI elements should be updated only from the main thread. Second, an ugly race condition is prevented: imagine starting two searches in very quick succession. Because the three global queues execute blocks concurrently, it could happen that two blocks try to update the results array at the same time, which would most likely make your application crash. Because the main queue always waits for one block to be done before it executes the next, you can always be sure that only one block tries to update the results array at any given time.

When you run your application now, notice that the search works much more smoothly. But it still freezes for a short time and multiple times when you scroll. You still have to get the loading of the images off of the main thread. Let's do that next.

13.2.5 Making the image loading asynchronous

With listing 13.22, you change your `tableView:cellForRowAtIndexPath:` method quite a bit. You check whether you already have the requested image. If so, you just set up a `UIImageView` with it and return the cell. If not, you load the image in a block, and the block calls back to the main thread by dispatching another block to the main queue, which caches the image and tells the `tableView` to reload the cell for the affected row. That causes the `tableView:cellForRowAtIndexPath:` method to be called again, but this time it finds a cached image and is happy.

Listing 13.22 Asynchronous image loading with GCD

```
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier] autorelease];
    } else {
        for (UIView *view in cell.contentView.subviews) {
            [view removeFromSuperview];
        }
    }

    __block UIImage *image =
        [[results objectAtIndex:indexPath.row] objectForKey:@"image"];

    if (!image) {
        void (^imageLoadingBlock)(void);

        UIActivityIndicatorView *spinner =
            [[UIActivityIndicatorView alloc]
             initWithActivityIndicatorStyle:
             UIActivityIndicatorViewStyleGray];

        spinner.autoresizingMask =
            UIViewAutoresizingFlexibleLeftMargin |
            UIViewAutoresizingFlexibleRightMargin |
            UIViewAutoresizingFlexibleTopMargin |
            UIViewAutoresizingFlexibleBottomMargin;

        spinner.contentMode = UIViewContentModeCenter;
        spinner.center = cell.contentView.center;
        [spinner startAnimating];

        [cell.contentView addSubview:spinner];
        [spinner release];

        imageLoadingBlock = ^{
            image = [UIImage imageWithData:
                    [NSData dataWithContentsOfURL:

```

**Attempt to fetch cached image;
mark variable editable**

**Create variable
to hold a block**

**Create a spinner to add
to the table view cell**

**Create block to
load the image**

```

[NSURL URLWithString:
 [[results objectAtIndex:indexPath.row]
  objectForKey:@"unescapedUrl"]]];

[image retain];

dispatch_async(dispatch_get_main_queue(), ^{
  [[results objectAtIndex:indexPath.row]
   setValue:image
   forKey:@"image"];
  [image release];
  [spinner stopAnimating];

  // reload the affected row
  [self.tableView
   reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
   withRowAnimation:NO];
});

dispatch_async(
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
  imageLoadingBlock);
} else {
  UIImageView *imageView =
    [[[UIImageView alloc] initWithImage:image] autorelease];

  imageView.contentMode = UIViewContentModeScaleAspectFit;
  imageView.autoresizingMask =
    UIViewAutoresizingFlexibleWidth |
    UIViewAutoresizingFlexibleHeight;
  imageView.frame = cell.contentView.frame;

  [cell.contentView addSubview:imageView];
}

return cell;
}

```

Dispatch anonymous block to main queue

Cache image

Reload affected row

Asynchronously dispatch image loading block

Use and display cached image

This code first checks whether a cell is available for reuse. If so, it removes all of the cell's subviews (the image view and the spinner). Then the `__block` storage type is used for the image variable because you want to be able to set its value from inside the block in case you don't have a cached version of it yet. Inside the block, you need to retain the image because `__block` keeps it from being retained automatically. Finally you release the image again in the block that's run on the main queue because you first added it to the matching results dictionary, which retains the image for you.

When you run your application now, everything should work smoothly and never freeze once.

13.3 Summary

We covered a lot of ground in this chapter. We looked at blocks, a powerful and versatile new addition to the C language, and we got our feet wet with GCD, an easy way to add concurrency to your application and keep it responsive at all times. There's a lot

more to learn about GCD, but that would go beyond the scope of this chapter. We covered the most important use case: performing work in the background and calling back to the main thread to update the UI. To dig deeper into GCD, you should look at Apple's Concurrent Programming Guide at <http://developer.apple.com/library/ios>.

Chapter 14, our final chapter, covers advanced debugging techniques.

Objective-C Fundamentals

Fairbairn • Fahrenkrug • Ruffenach



Objective-C Fundamentals guides you gradually from your first line of Objective-C code through the process of building native apps for the iPhone. Starting with chapter one, you'll dive into iPhone development by building a simple game that you can run immediately. You'll use tools like Xcode 4 and the debugger that will help you become a more efficient programmer. By working through numerous easy-to-follow examples, you'll learn practical techniques and patterns you can use to create solid and stable apps. And you'll find out how to avoid the most common pitfalls.

What's Inside

- Objective-C from the ground up
- Developing with Xcode 4
- Examples work unmodified on iPhone

No iOS or mobile experience is required to benefit from this book but familiarity with programming in general is helpful.

Christopher Fairbairn, Johannes Fahrenkrug, and Collin Ruffenach are professional mobile app developers, each with over a decade of experience using different systems including iOS, Palm, Windows Mobile, and Java.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/ObjectiveCFundamentals

“A handy and complete reference.”

—Glenn Stokol
Oracle Corporation.

“The essential iOS programming how-to guide.”

—Dave Bales, Whitescape

“A tour-de-force of Objective-C...I want to grok this stuff!”

—Dave Mateer, Mateer IT

“A superb introduction to essential iPhone application development tools.”

—Carl Douglas, NZX

“Become a hot commodity on the market... with this book.”

—Ted Neward, Principal,
Neward & Associates

