Working with Blocks

An Objective-C class defines an object that combines data with related behavior. Sometimes, it makes sense just to represent a single task or unit of behavior, rather than a collection of methods.

Blocks are a language-level feature added to C, Objective-C and C++, which allow you to create distinct segments of code that can be passed around to methods or functions as if they were values. Blocks are Objective-C objects, which means they can be added to collections like `NSArray` or `NSDictionary`. They also have the ability to capture values from the enclosing scope, making them similar to *closures* or *lambdas* in other programming languages.

This chapter explains the syntax to declare and refer to blocks, and shows how to use blocks to simplify common tasks such as collection enumeration. For further information, see *Blocks Programming Topics*.

Block Syntax

The syntax to define a block literal uses the caret symbol (^), like this:

```
^{
     NSLog(@"This is a block");
}
```

As with function and method definitions, the braces indicate the start and end of the block. In this example, the block doesn't return any value, and doesn't take any arguments.

In the same way that you can use a function pointer to refer to a C function, you can declare a variable to keep track of a block, like this:

```
void (^simpleBlock)(void);
```

If you're not used to dealing with C function pointers, the syntax may seem a little unusual. This example declares a variable called `simpleBlock` to refer to a block that takes no arguments and doesn't return a value, which means the variable can be assigned the block literal shown above, like this:

```
simpleBlock = ^{
    NSLog(@"This is a block");
};
```

This is just like any other variable assignment, so the statement must be terminated by a semi-colon after the closing brace. You can also combine the variable declaration and assignment:

```
void (^simpleBlock)(void) = ^{
    NSLog(@"This is a block");
};
```

Once you've declared and assigned a block variable, you can use it to invoke the block:

```
simpleBlock();
```

Note: If you attempt to invoke a block using an unassigned variable (a `nil` block variable), your app will crash.

Blocks Take Arguments and Return Values

Blocks can also take arguments and return values just like methods and functions.

As an example, consider a variable to refer to a block that returns the result of multiplying two values:

```
double (^multiplyTwoValues)(double, double);
```

The corresponding block literal might look like this:

```
^ (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

The `firstValue` and `secondValue` are used to refer to the values supplied when the block is invoked, just like any function definition. In this example, the return type is inferred from the return statement inside the block.

If you prefer, you can make the return type explicit by specifying it between the caret and the argument list:

```
^ double (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

Once you've declared and defined the block, you can invoke it just like you would a function:

```
double (^multiplyTwoValues)(double, double) =
                          ^(double firstValue, double secondValue) {
                              return firstValue * secondValue;
                          };

double result = multiplyTwoValues(2,4);

NSLog(@"The result is %f", result);
```

Blocks Can Capture Values from the Enclosing Scope

As well as containing executable code, a block also has the ability to capture state from its enclosing scope.

If you declare a block literal from within a method, for example, it's possible to capture any of the values accessible within the scope of that method, like this:

```
- (void)testMethod {
    int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
```

```
    };

    testBlock();
}
```

In this example, `anInteger` is declared outside of the block, but the value is captured when the block is defined.

Only the value is captured, unless you specify otherwise. This means that if you change the external value of the variable between the time you define the block and the time it's invoked, like this:

```
    int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
    };

    anInteger = 84;

    testBlock();
```

the value captured by the block is unaffected. This means that the log output would still show:

```
Integer is: 42
```

It also means that the block cannot change the value of the original variable, or even the captured value (it's captured as a `const` variable).

Use __block Variables to Share Storage

If you need to be able to change the value of a captured variable from within a block, you can use the `__block` storage type modifier on the original variable declaration. This means that the variable lives in storage that is shared between the lexical scope of the original variable and any blocks declared within that scope.

As an example, you might rewrite the previous example like this:

```
    __block int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
    };

    anInteger = 84;

    testBlock();
```

Because `anInteger` is declared as a `__block` variable, its storage is shared with the block declaration. This means that the log output would now show:

```
Integer is: 84
```

It also means that the block can modify the original value, like this:

```
    __block int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
        anInteger = 100;
    };

    testBlock();
    NSLog(@"Value of original variable is now: %i", anInteger);
```

This time, the output would show:

```
Integer is: 42
Value of original variable is now: 100
```

You Can Pass Blocks as Arguments to Methods or Functions

Each of the previous examples in this chapter invokes the block immediately after it's defined. In practice, it's common to pass blocks to functions or methods for invocation elsewhere. You might use Grand Central Dispatch to invoke a block in the background, for example, or define a block to represent a task to be invoked repeatedly, such as when enumerating a collection. Concurrency and enumeration are covered later in this chapter.

Blocks are also used for callbacks, defining the code to be executed when a task completes. As an example, your app might need to respond to a user action by creating an object that performs a complicated task, such as requesting information from a web service. Because the task might take a long time, you should display some kind of progress indicator while the task is occurring, then hide that indicator once the task is complete.

It would be possible to accomplish this using delegation: You'd need to create a suitable delegate protocol, implement the required method, set your object as the delegate of the task, then wait for it to call a delegate method on your object once the task finished.

Blocks make this much easier, however, because you can define the callback behavior at the time you initiate the task, like this:

```
- (IBAction)fetchRemoteInformation:(id)sender {
    [self showProgressIndicator];

    XYZWebTask *task = ...

    [task beginTaskWithCallbackBlock:^{
        [self hideProgressIndicator];
    }];
}
```

This example calls a method to display the progress indicator, then creates the task and tells it to start. The callback block specifies the code to be executed once the task completes; in this case, it simply calls a method to hide the progress indicator. Note that this callback block captures `self` in order to be able to call the `hideProgressIndicator` method when invoked. It's important to take care when capturing `self` because it's easy to create a strong reference cycle, as described later in Avoid Strong Reference Cycles when Capturing self.

In terms of code readability, the block makes it easy to see in one place exactly what will happen before and after the task completes, avoiding the need to trace through delegate methods to find out what's going to happen.

The declaration for the `beginTaskWithCallbackBlock:` method shown in this example would look like this:

```
- (void)beginTaskWithCallbackBlock:(void (^)(void))callbackBlock;
```

The `(void (^)(void))` specifies that the parameter is a block that doesn't take any arguments or return any values. The implementation of the method can invoke the block in the usual way:

```
- (void)beginTaskWithCallbackBlock:(void (^)(void))callbackBlock {
    ...
    callbackBlock();
}
```

Method parameters that expect a block with one or more arguments are specified in the same way as with a block variable:

```
- (void)doSomethingWithBlock:(void (^)(double, double))block {
    ...
    block(21.0, 2.0);
}
```

A Block Should Always Be the Last Argument to a Method

It's best practice to use only one block argument to a method. If the method also needs other non-block arguments, the block should come last:

```
- (void)beginTaskWithName:(NSString *)name completion:(void(^)(void))callback;
```

This makes the method call easier to read when specifying the block inline, like this:

```
    [self beginTaskWithName:@"MyTask" completion:^{
        NSLog(@"The task is complete");
    }];
```

Use Type Definitions to Simplify Block Syntax

If you need to define more than one block with the same signature, you might like to define your own type for that signature.

As an example, you can define a type for a simple block with no arguments or return value, like this:

```
typedef void (^XYZSimpleBlock)(void);
```

You can then use your custom type for method parameters or when creating block variables:

```
    XYZSimpleBlock anotherBlock = ^{
        ...
    };
```

```
- (void)beginFetchWithCallbackBlock:(XYZSimpleBlock)callbackBlock {
    ...
    callbackBlock();
```

```
}
```

Custom type definitions are particularly useful when dealing with blocks that return blocks or take other blocks as arguments. Consider the following example:

```
void (^(^complexBlock)(void (^)(void)))(void) = ^ (void (^aBlock)(void)) {
    ...
    return ^{
        ...
    };
};
```

The `complexBlock` variable refers to a block that takes another block as an argument (`aBlock`) and returns yet another block.

Rewriting the code to use a type definition makes this much more readable:

```
XYZSimpleBlock (^betterBlock)(XYZSimpleBlock) = ^ (XYZSimpleBlock aBlock) {
    ...
    return ^{
        ...
    };
};
```

Objects Use Properties to Keep Track of Blocks

The syntax to define a property to keep track of a block is similar to a block variable:

```
@interface XYZObject : NSObject
@property (copy) void (^blockProperty)(void);
@end
```

Note: You should specify `copy` as the property attribute, because a block needs to be copied to keep track of its captured state outside of the original scope. This isn't something you need to worry about when using Automatic Reference Counting, as it will happen automatically, but it's best practice for the property attribute to show the resultant behavior. For more information, see *Blocks Programming Topics*.

A block property is set or invoked like any other block variable:

```
    self.blockProperty = ^{
        ...
    };
    self.blockProperty();
```

It's also possible to use type definitions for block property declarations, like this:

```
typedef void (^XYZSimpleBlock)(void);

@interface XYZObject : NSObject
@property (copy) XYZSimpleBlock blockProperty;
```

```
@end
```

Avoid Strong Reference Cycles when Capturing self

If you need to capture `self` in a block, such as when defining a callback block, it's important to consider the memory management implications.

Blocks maintain strong references to any captured objects, including `self`, which means that it's easy to end up with a strong reference cycle if, for example, an object maintains a `copy` property for a block that captures `self`:

```
@interface XYZBlockKeeper : NSObject
@property (copy) void (^block)(void);
@end
```

```
@implementation XYZBlockKeeper
- (void)configureBlock {
    self.block = ^{
        [self doSomething];    // capturing a strong reference to self
                               // creates a strong reference cycle
    };
}
...
@end
```

The compiler will warn you for a simple example like this, but a more complex example might involve multiple strong references between objects to create the cycle, making it more difficult to diagnose.

To avoid this problem, it's best practice to capture a weak reference to `self`, like this:

```
- (void)configureBlock {
    XYZBlockKeeper * __weak weakSelf = self;
    self.block = ^{
        [weakSelf doSomething];   // capture the weak reference
                                  // to avoid the reference cycle
    }
}
```

By capturing the weak pointer to `self`, the block won't maintain a strong relationship back to the `XYZBlockKeeper` object. If that object is deallocated before the block is called, the `weakSelf` pointer will simply be set to `nil`.

Blocks Can Simplify Enumeration

In addition to general completion handlers, many Cocoa and Cocoa Touch API use blocks to simplify common tasks, such as collection enumeration. The `NSArray` class, for example, offers three block-based methods, including:

```
- (void)enumerateObjectsUsingBlock:(void (^)(id obj, NSUInteger idx, BOOL
*stop))block;
```

This method takes a single argument, which is a block to be invoked once for each item in the array:

```
NSArray *array = ...
[array enumerateObjectsUsingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
    NSLog(@"Object at index %lu is %@", idx, obj);
}];
```

The block itself takes three arguments, the first two of which refer to the current object and its index in the array. The third argument is a pointer to a Boolean variable that you can use to stop the enumeration, like this:

```
[array enumerateObjectsUsingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
    if (...) {
        *stop = YES;
    }
}];
```

It's also possible to customize the enumeration by using the `enumerateObjectsWithOptions:usingBlock:` method. Specifying the `NSEnumerationReverse` option, for example, will iterate through the collection in reverse order.

If the code in the enumeration block is processor-intensive—and safe for concurrent execution—you can use the `NSEnumerationConcurrent` option:

```
[array enumerateObjectsWithOptions:NSEnumerationConcurrent
                        usingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
    ...
}];
```

This flag indicates that the enumeration block invocations may be distributed across multiple threads, offering a potential performance increase if the block code is particularly processor intensive. Note that the enumeration order is undefined when using this option.

The `NSDictionary` class also offers block-based methods, including:

```
NSDictionary *dictionary = ...
[dictionary enumerateKeysAndObjectsUsingBlock:^ (id key, id obj, BOOL *stop) {
    NSLog(@"key: %@, value: %@", key, obj);
}];
```

This makes it more convenient to enumerate each key-value pair than when using a traditional loop, for example.

Blocks Can Simplify Concurrent Tasks

A block represents a distinct unit of work, combining executable code with optional state captured from the surrounding scope. This makes it ideal for asynchronous invocation using one of the concurrency options available for OS X and iOS. Rather than having to figure out how to work with low-level mechanisms like threads, you can simply define your tasks using blocks and then let the system perform those tasks as processor resources become available.

OS X and iOS offer a variety of technologies for concurrency, including two task-scheduling mechanisms: Operation queues and Grand Central Dispatch. These mechanisms revolve around the idea of a queue of tasks waiting to be invoked. You add your blocks to a queue in the order you need them to be invoked, and the system dequeues them for invocation when processor time and resources become available.

A *serial queue* only allows one task to execute at a time—the next task in the queue won't be dequeued and invoked until the previous task has finished. A *concurrent queue* invokes as many tasks as it can, without waiting for previous tasks to finish.

Use Block Operations with Operation Queues

An operation queue is the Cocoa and Cocoa Touch approach to task scheduling. You create an `NSOperation` instance to encapsulate a unit of work along with any necessary data, then add that operation to an `NSOperationQueue` for execution.

Although you can create your own custom `NSOperation` subclass to implement complex tasks, it's also possible to use the `NSBlockOperation` to create an operation using a block, like this:

```
NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^{
    ...
}];
```

It's possible to execute an operation manually but operations are usually added either to an existing operation queue or a queue you create yourself, ready for execution:

```
// schedule task on main queue:
NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];
[mainQueue addOperation:operation];

// schedule task on background queue:
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[queue addOperation:operation];
```

If you use an operation queue, you can configure priorities or dependencies between operations, such as specifying that one operation should not be executed until a group of other operations has completed. You can also monitor changes to the state of your operations through key-value observing, which makes it easy to update a progress indicator, for example, when a task completes.

For more information on operations and operation queues, see Operation Queues.

Schedule Blocks on Dispatch Queues with Grand Central Dispatch

If you need to schedule an arbitrary block of code for execution, you can work directly with *dispatch queues* controlled by Grand Central Dispatch (GCD). Dispatch queues make it easy to perform tasks either synchronously or asynchronously with respect to the caller, and execute their tasks in a first-in, first-out order.

You can either create your own dispatch queue or use one of the queues provided automatically by GCD. If you need to schedule a task for concurrent execution, for example, you can get a reference to an existing queue by using the `dispatch_get_global_queue()` function and specifying a queue priority, like this:

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

To dispatch the block to the queue, you use either the `dispatch_async()` or `dispatch_sync()` functions. The `dispatch_async()` function returns immediately, without waiting for the block to be invoked:

```
dispatch_async(queue, ^{
    NSLog(@"Block for asynchronous execution");
});
```

The `dispatch_sync()` function doesn't return until the block has completed execution; you might use it in a situation where a concurrent block needs to wait for another task to complete on the main thread before continuing, for example.

For more information on dispatch queues and GCD, see Dispatch Queues.

Copyright © 2014 Apple Inc. All Rights Reserved. Terms of Use | Privacy Policy | Updated: 2014-09-17