A Short Practical Guide to Blocks

Blocks are a powerful C-language feature that is part of Cocoa application development. They are similar to “closures” and “lambdas” you may find in scripting and programming languages such as Ruby, Python, and Lisp. Although the syntax and storage details of blocks might at first glance seem cryptic, you’ll find that it’s actually quite easy to incorporate blocks into your projects’ code.

The following discussion gives a high-level survey of the features of blocks and illustrates the typical ways in which they are used. Refer to Blocks Programming Topics for the definitive description of blocks.

Contents:

Why Use Blocks?

Blocks are objects that encapsulate a unit of work—or, in less abstract terms, a segment of code—that can be executed at any time. They are essentially portable and anonymous functions that one can pass in as arguments of methods and functions or that can be returned from methods and functions. Blocks themselves have a typed argument list and may have inferred or declared returned type. You may also assign a block to a variable and then call it just as you would a function.

The caret symbol (^) is used as a syntactic marker for blocks. For example, the following code declares a block variable taking two integers and returning an integer value. It provides the parameter list after the second caret and the implementing code within the braces, and assigns these to the `Multiply` variable:

```
int (^Multiply)(int, int) = ^(int num1, int num2) {
    return num1 * num2;
};
int result = Multiply(7, 4); // Result is 28.
```

As method or function arguments, blocks are a type of callback and could

be considered a form of delegation limited to the method or function. By passing in a block, calling code can customize the behavior of a method or function. When invoked, the method or function performs some work and, at the appropriate moments, calls back to the invoking code—via the block—to request additional information or to obtain application-specific behavior from it.

An advantage of blocks as function and method parameters is that they enable the caller to provide the callback code at the point of invocation. Because this code does not have to be implemented in a separate method or function, your implementation code can be simpler and easier to understand. Take notifications of the `NSNotification` variety as an example. In the "traditional" approach, an object adds itself as an observer of a notification and then implements a separate method (identified by a selector in the `addObserver:..` method) to handle the notification:

```
- (void)viewDidLoad {
   [super viewDidLoad];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardWillShow:)
        name:UIKeyboardWillShowNotification object:nil];
}

- (void)keyboardWillShow:(NSNotification *)notification {
    // Notification-handling code goes here.
}
```

With the `addObserverForName:object:queue:usingBlock:` method you can consolidate the notification-handling code with the method invocation:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [[NSNotificationCenter defaultCenter]
addObserverForName:UIKeyboardWillShowNotification
```

```
        object:nil queue:[NSOperationQueue mainQueue]
usingBlock:^(NSNotification *notif) {

            // Notification-handling code goes here.

    }];

}
```

An even more valuable advantage of blocks over other forms of callback is that a block shares data in the local lexical scope. If you implement a method and in that method define a block, the block has access to the local variables and parameters of the method (including stack variables) as well as to functions and global variables, including instance variables. This access is read-only by default, but if you declare a variable with the `__block` modifier, you can change its value within the block. Even after the method or function enclosing a block has returned and its local scope is destroyed, the local variables persist as part of the block object as long as there is a reference to the block.

Blocks in the System Framework APIs

One obvious motivation for using blocks is that an increasing number of the methods and functions of the system frameworks take blocks as parameters. One can discern a half-dozen or so use cases for blocks in framework methods:

- Completion handlers
- Notification handlers
- Error handlers
- Enumeration
- View animation and transitions
- Sorting

The sections that follow describe each of these cases. But before we get to that, here is a quick overview on interpreting block declarations in framework methods. Consider the following method of the `NSSet` class:

```
- (NSSet *)objectsPassingTest:(BOOL (^)(id obj, BOOL
*stop))predicate
```

The block declaration indicates that the method passes into the block (for each enumerated item) a dynamically typed object and a by-reference Boolean value; the block returns a Boolean value. (What these parameters and return value are actually for are covered in Enumeration.) When specifying your block, begin with a caret (`^`) and the parenthesized

argument list; follow this with the block code itself, enclosed by braces.

```
[mySet objectsPassingTest:^(id obj, BOOL *stop) {

    // Code goes here: Return YES if obj passes the test and NO
if obj does not pass the test.

}];
```

Completion and Error Handlers

Completion handlers are callbacks that allow a client to perform some action when a framework method or function completes its task. Often the client uses a completion handler to free state or update the user interface. Several framework methods let you implement completion handlers as blocks (instead of, say, delegation methods or notification handlers).

The `UIView` class has several class methods for animations and view transitions that have completion-handler block parameters. (View Animation and Transitions gives an overview of these methods.) The example in Listing 1-1 shows an implementation of the `animateWithDuration:animations:completion:` method. The completion handler in this example resets the animated view back to its original location and transparency (alpha) value a few seconds after the animation concludes.

Listing 1-1 A completion-handler block

```
- (IBAction)animateView:(id)sender {

    CGRect cacheFrame = self.imageView.frame;

    [UIView animateWithDuration:1.5 animations:^{

        CGRect newFrame = self.imageView.frame;

        newFrame.origin.y = newFrame.origin.y + 150.0;

        self.imageView.frame = newFrame;

        self.imageView.alpha = 0.2;

    }

                     completion:^ (BOOL finished) {

                         if (finished) {
```

```
                                // Revert image view to original.
                                self.imageView.frame = cacheFrame;
                                self.imageView.alpha = 1.0;
                            }
    }];
}
```

Some framework methods have error handlers, which are block parameters similar to completion handlers. The method invokes them (and passes in an `NSError` object) when it cannot complete its task because of some error condition. You typically implement an error handler to inform the user about the error.

Notification Handlers

The `NSNotificationCenter` method `addObserverForName:object:queue:usingBlock:` lets you implement the handler for a notification at the point you set up the observation. Listing 1-2 illustrates how you might call this method, defining a block handler for the notification. As with notification-handler methods, an `NSNotification` object is passed in. The method also takes an `NSOperationQueue` instance, so your application can specify an execution context on which to run the block handler.

Listing 1-2 Adding an object as an observer and handling a notification using a block

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

    opQ = [[NSOperationQueue alloc] init];

    [[NSNotificationCenter defaultCenter]
addObserverForName:@"CustomOperationCompleted"

            object:nil queue:opQ

        usingBlock:^(NSNotification *notif) {

        NSNumber *theNum = [notif.userInfo
objectForKey:@"NumberOfItemsProcessed"];

        NSLog(@"Number of items processed: %i", [theNum
intValue]);
```

```
    }];

}
```

Enumeration

The collection classes of the Foundation framework—`NSArray`, `NSDictionary`, `NSSet`, and `NSIndexSet`—declare methods that perform the enumeration of a particular type of collection and that specify blocks for clients to supply code to handle or test each enumerated item. In other words, the methods perform the equivalent of the fast-enumeration construct:

```
for (id item in collection) {

    // Code to operate on each item in turn.

}
```

There are two general forms of the enumeration methods that take blocks. The first are methods whose names begin with `enumerate` and do not return a value. The block for these methods performs some work on each enumerated item. The block parameter of the second type of method is preceded by `passingTest`; this kind of method returns an integer or an `NSIndexSet` object. The block for these methods performs a test on each enumerated item and returns `YES` if the item passes the test. The integer or index set identifies the object or objects in the original collection that passed the test.

The code in Listing 1-3 calls an `NSArray` method of each of these types. The block of the first method (a “passing test” method) returns `YES` for every string in an array that has a certain prefix. The code subsequently creates a temporary array using the index set returned from the method. The block of the second method trims away the prefix from each string in the temporary array and adds it to a new array.

Listing 1-3 Processing enumerated arrays using two blocks

```
NSString *area = @"Europe";

NSArray *timeZoneNames = [NSTimeZone knownTimeZoneNames];

NSMutableArray *areaArray = [NSMutableArray arrayWithCapacity:
1];

NSIndexSet *areaIndexes = [timeZoneNames
indexesOfObjectsWithOptions:NSEnumerationConcurrent
```

```
                                    passingTest:^(id obj,
NSUInteger idx, BOOL *stop) {

    NSString  *tmpStr = (NSString *)obj;

    return [tmpStr hasPrefix:area];

}];


NSArray *tmpArray = [timeZoneNames
objectsAtIndexes:areaIndexes];

[tmpArray enumerateObjectsWithOptions:NSEnumerationConcurrent|
NSEnumerationReverse

                              usingBlock:^(id obj, NSUInteger idx,
BOOL *stop) {

                                  [areaArray addObject:[obj
substringFromIndex:[area length]+1]];

}];

NSLog(@"Cities in %@ time zone:%@", area, areaArray);
```

The `stop` parameter in each of these enumeration methods (which is not used in the example) allows the block to pass `YES` by reference back to the method to tell it to quit enumerating. You do this when you just want to find the first item in the collection that matches some criteria.

Even though it does not represent a collection, the `NSString` class also has two methods with block parameters whose names begin with `enumerate`: `enumerateSubstringsInRange:options:usingBlock:` and `enumerateLinesUsingBlock:`. The first method enumerates a string by a text unit of a specified granularity (line, paragraph, word, sentence, and so on); the second method enumerates it by line only. Listing 1-4 illustrates how you might use the first method.

Listing 1-4 Using a block to find matching substrings in a string

```
NSString *musician = @"Beatles";

NSString *musicDates = [NSString stringWithContentsOfFile:

    @"/usr/share/calendar/calendar.music"

    encoding:NSASCIIStringEncoding error:NULL];
```

```
[musicDates enumerateSubstringsInRange:NSMakeRange(0,
[musicDates length]-1)

    options:NSStringEnumerationByLines

    usingBlock:^(NSString *substring, NSRange substringRange,
NSRange enclosingRange, BOOL *stop) {

            NSRange found = [substring rangeOfString:musician];

            if (found.location != NSNotFound) {

                 NSLog(@"%@", substring);

            }

      }];
```

View Animation and Transitions

The `UIView` class in iOS 4.0 introduced several class methods for animation and view transitions that take blocks. The block parameters are of two kinds (not all methods take both kinds):

- Blocks that change the view properties to be animated
- Completion handlers

Listing 1-5 shows an invocation of `animateWithDuration:animations:completion:`, a method that has both kinds of block parameters. In this example, the animation makes the view disappear (by specifying zero alpha) and the completion handler removes it from its superview.

Listing 1-5 Simple animation of a view using blocks

```
[UIView animateWithDuration:0.2 animations:^{

        view.alpha = 0.0;

    } completion:^(BOOL finished){

        [view removeFromSuperview];

    }];
```

Other `UIView` class methods perform transitions between two views, including flips and curls. The example invocation of `transitionWithView:duration:options:animations:completion:` in Listing 1-6 animates the replacement of a subview as a flip-left transition. (It does not implement a completion handler.)

Listing 1-6 Implementing a flip transition between two views

```
[UIView transitionWithView:containerView duration:0.2

options:UIViewAnimationOptionTransitionFlipFromLeft
                animations:^{
                    [fromView removeFromSuperview];
                    [containerView addSubview:toView]
                }
                completion:NULL];
```

Sorting

The Foundation framework declares the `NSComparator` type for comparing two items:

```
typedef NSComparisonResult (^NSComparator)(id obj1, id obj2);
```

`NSComparator` is a block type that takes two objects and returns an `NSComparisonResult` value. It is a parameter in methods of `NSSortDescriptor`, `NSArray`, and `NSDictionary` and is used by instances of those classes for sorting. Listing 1-7 gives an example of its use.

Listing 1-7 Sorting an array using an NSComparator block

```
NSArray *stringsArray = [NSArray arrayWithObjects:
                                  @"string 1",
                                  @"String 21",
                                  @"string 12",
                                  @"String 11",
                                  @"String 02", nil];
static NSStringCompareOptions comparisonOptions =
NSCaseInsensitiveSearch | NSNumericSearch |
        NSWidthInsensitiveSearch | NSForcedOrderingSearch;
NSLocale *currentLocale = [NSLocale currentLocale];
NSComparator finderSort = ^(id string1, id string2) {
```

```
    NSRange string1Range = NSMakeRange(0, [string1 length]);

    return [string1 compare:string2 options:comparisonOptions
range:string1Range locale:currentLocale];

};

NSLog(@"finderSort: %@", [stringsArray
sortedArrayUsingComparator:finderSort]);
```

This example is taken from Blocks Programming Topics.

Blocks and Concurrency

Blocks are portable and anonymous objects encapsulating a unit of work that can be performed asynchronously at a later time. Because of this essential fact, blocks are a central feature of both Grand Central Dispatch (GCD) and the `NSOperationQueue` class, the two recommended technologies for concurrent processing.

- The two central functions of GCD, `dispatch_sync(3) OS X Developer Tools Manual Page` (for synchronous dispatching) or `dispatch_async(3) OS X Developer Tools Manual Page` (for asynchronous dispatching) take as their second arguments a block.
- An `NSOperationQueue` is an object that schedules tasks to be performed concurrently or in an order defined by dependency relationships. The tasks are represented by `NSOperation` objects, which frequently use blocks to implement their tasks.

For more about GCD, `NSOperationQueue`, and `NSOperation`, see Concurrency Programming Guide.