Technical Note TN2109

Simple and Reliable Threading with NSOperation

Threads are a great way to solve a number of tricky problems. For example, if you have some long-running computation to do, you really want to push that work off to a secondary thread so as to keep your user interface responsive. However, it's very easy to get into trouble when using threads in an iOS application. Threads are tricky to program with in general and, of course, large parts of Cocoa Touch can only be called from the main thread.

This technote describes one way to manage your threaded code, namely, via NSOperation. This is a specific example of a general technique known as **thread confinement**, where you strictly limit the scope of your threaded code so as to avoid traditional threading problems, like accessing data structures while they are being mutated by another thread, as well as platform-specific issues, like accidentally calling UIKit from a secondary thread.

This article is specifically written for iOS application developers, but the core techniques discussed here also apply to Mac OS X.

Introduction
Threads And Their Problems
NSOperation To The Rescue!
    Using NSOperation in ListAdder
    AdderOperation: Beyond the Basics
    Thread Confinement
NSOperations Hints and Tips
    NSOperation and GCD
    The Deallocation Problem
    Concurrent Operations
    Unfinished Operations
Document Revision History

Introduction

Threads let you solve two tricky problems:

- parallel computation — In this world of multicore CPUs, many tasks will run faster if you divide the work between threads that are scheduled on different cores.

  This approach can be useful even on a single core system (as is the case for all current iOS devices). Specifically, if your computation thread spends some fraction of its time blocked (for example, waiting for disk I/O), you can use multiple threads to do useful computation on one thread while the other thread is blocked.

- asynchronous computation — If you have some long-running computation to do and, as is the case on iOS, you must keep the main thread responsive, you can move that computation to a secondary thread. This allows your main thread to treat the computation as it would any other asynchronous operation.

However, threads are not always the right solution. Threads have a significant cost, in terms of both memory and, more importantly, code complexity. In many cases it's a good idea to use the asynchronous APIs available in iOS instead. Specifically, a thread is a poor choice if you expect it to spend a significant amount of its time blocked. For example:

- periodic execution — Starting a thread just to execute code periodically is never a good idea. You should use NSTimer instead.

- networking — On a typical iOS device, the CPU is much faster than the network so, if you start a thread to do your networking synchronously, that thread will spend most of its time blocked. A better approach is to use the asynchronous networking APIs provided by iOS.

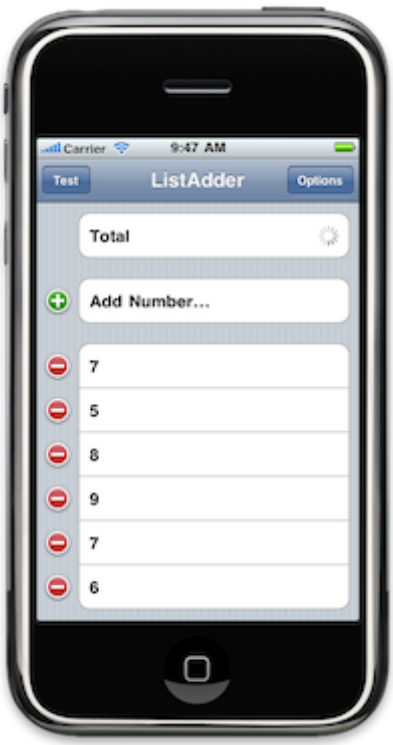
Once you've decided that it's appropriate to solve your problem with threads, you need to decide on the overall structure of your code. A common approach is to use the NSThread API directly, or perhaps take advantage of Cocoa's `-performSelectorInBackground:withObject:` method. It is, however, very easy to run into problems with these mechanism. The first section of this technote (Threads And Their Problems) describe some of those problems, just so you can get a feel for why it's so important to be careful with threads. The next section (NSOperation To The Rescue!) describes how you can use NSOperation to address those problems in a simple and reliable way. Finally, the last section (NSOperations Hints and Tips) offers some hints and tips for using NSOperation effectively.

Back to Top

Threads And Their Problems

Imagine you have an application that displays a list of numbers with a total at the top. Actually, you don't have to imagine this, you can just look at Sample Code 'ListAdder', which was specifically created for this technote. Figure 1 shows the basic UI.

Figure 1: The ListAdder UI



Such an application might have a view controller class with a property that is the list of numbers, represented as a mutable array. Listing 1 shows how this might be declared.

Listing 1: The numbers array declaration

```
@interface ListAdderViewController () [...]  @property (nonatomic, retain, readonly
) NSMutableArray *   numbers;  [...]  @end
```

Now imagine, just for the sake of this example, that adding up the list of numbers takes a very long time. So, as the user changes the list, you have to update the total, but you can't do that on the main thread because it would block the user interface too long. Thus you have to do the summation asynchronously. One obvious way to do that is to start a thread. Listing 2 shows how you might do

this. You'll note that the core calculation loop contains a long delay so that you can actually see the various threading problems as they crop up.

Listing 2: Recalculating on a thread

```
// ---- Broken Code ---- Do Not Use ---- Broken Code ---- Do Not Use ----

- (void)recalculateTotalUsingThread
{
    self.recalculating = YES;
    [self performSelectorInBackground:@selector(threadRecalculateNumbers)
withObject:self.numbers];
}

- (void)threadRecalculateNumbers
{
    NSAutoreleasePool *     pool;
    NSInteger               total;
    NSUInteger              numberCount;
    NSUInteger              numberIndex;
    NSString *              totalStr;

    pool = [[NSAutoreleasePool alloc] init];
    assert(pool != nil);

    total = 0;
    numberCount = [self.numbers count];
    for (numberIndex = 0; numberIndex < numberCount; numberIndex++) {
        NSNumber *  numberObj;

        // Sleep for a while. This makes it easiest to test various problematic
cases.

        [NSThread sleepForTimeInterval:1.0];

        // Do the mathematics.

        numberObj = [self.numbers objectAtIndex:numberIndex];
        assert([numberObj isKindOfClass:[NSNumber class]]);

        total += [numberObj integerValue];
    }

    // The user interface is adjusted by a KVO observer on recalculating.

    totalStr = [NSString stringWithFormat:@"%ld", (long) total];
    self.formattedTotal = totalStr;
    self.recalculating = NO;

    [pool drain];
}

// ---- Broken Code ---- Do Not Use ---- Broken Code ---- Do Not Use ----
```

This code has numerous issues. The first one is pretty obvious: the end of `-threadRecalculateNumbers` applies the results to the user interface. In doing so, it ends up calling UIKit on a secondary thread. That's because other parts of the program use key-value observing (KVO) to observe the `recalculating` property and reload the table view accordingly. If you change `recalculating` on a secondary thread, this code ends up calling UIKit on a secondary thread, which is not allowed.

Fixing that problem is relatively easy. Listing 3 shows a replacement for `-threadRecalculateNumbers` with the fix.

Listing 3: Recalculating on a thread, second try

```
// ---- Broken Code ---- Do Not Use ---- Broken Code ---- Do Not Use ----

- (void)threadRecalculateNumbers
{
    NSAutoreleasePool *     pool;
    NSInteger               total;
    NSUInteger              numberCount;
    NSUInteger              numberIndex;
```

```
    NSString *              totalStr;

    pool = [[NSAutoreleasePool alloc] init];
    assert(pool != nil);

    total = 0;
    numberCount = [self.numbers count];
    for (numberIndex = 0; numberIndex < numberCount; numberIndex++) {
        NSNumber *  numberObj;

        // Sleep for a while. This makes it easiest to test various problematic
cases.

        [NSThread sleepForTimeInterval:1.0];

        // Do the mathematics.

        numberObj = [self.numbers objectAtIndex:numberIndex];
        assert([numberObj isKindOfClass:[NSNumber class]]);

        total += [numberObj integerValue];
    }

    // Update the user interface on the main thread.

    totalStr = [NSString stringWithFormat:@"%ld", (long) total];
    [self performSelectorOnMainThread:@selector(threadRecalculateDone:)
withObject:totalStr waitUntilDone:NO];

    [pool drain];
}

- (void)threadRecalculateDone:(NSString *)result
{
    // The user interface is adjusted by a KVO observer on recalculating.

    self.formattedTotal = result;
    self.recalculating = NO;
}

// ---- Broken Code ---- Do Not Use ---- Broken Code ---- Do Not Use ----
```

So, we're done, right? Wrong! While this code fixes the problem mentioned above, it still has other problems. One such problem relates to the `numbers` property. This is a mutable array, and the code is now accessing the array on one thread (the secondary thread) while potentially mutating the array on another (the main thread). So, what happens if the main thread changes the `numbers` array while a secondary thread is recalculating? Bad things. Imagine this scenario:

1. The user adds an item. This starts a long recalculation on thread A.
2. A short while later, before thread A completes, the user then removes an item. This starts another long recalculation on thread B.

Now think back to the main recalculation loop. For convenience, it's reproduced in Listing 4, with all the extraneous stuff cut away.

Listing 4: A stripped down recalculation loop

```
NSUInteger              numberCount;
NSUInteger              numberIndex;

numberCount = [self.numbers count];
for (numberIndex = 0; numberIndex < numberCount; numberIndex++) {
    NSNumber *  numberObj;

    numberObj = [self.numbers objectAtIndex:numberIndex];
    total += [numberObj integerValue];
}
```

As you can see, the code first gets the count of the number of elements in the array, then iterates over that many elements. If the user removes an item from the `numbers` array after the secondary thread has initialized `numberCount`, this code will eventually access an element off the end of the array and crash.

Note: To make things clearer the examples in this technote use old school array iteration techniques rather than the Objective-C 2 for..in syntax. However, this problem applies even if you use for..in syntax. Mutating an array while you're iterating it using for..in syntax will result in the `Collection xxx was mutated while being enumerated` exception.

Of course, fixing this is relatively easy: when you start the thread you make an immutable copy of the `numbers` array and have the thread operate on that. That way the thread is isolated from the changes done by the main thread. Listing 5 shows the basic idea.

Listing 5: Recalculating on a thread, third try

```
// ---- Broken Code ---- Do Not Use ---- Broken Code ---- Do Not Use ----

- (void)recalculateTotalUsingThread
{
    NSArray *       immutableNumbers;

    self.recalculating = YES;

    immutableNumbers = [[self.numbers copy] autorelease];
    assert(immutableNumbers != nil);
    [self performSelectorInBackground:@selector(threadRecalculateNumbers:)
withObject:immutableNumbers];
}

- (void)threadRecalculateNumbers:(NSArray *)immutableNumbers
{
    NSAutoreleasePool *     pool;
    NSInteger               total;
    NSUInteger              numberCount;
    NSUInteger              numberIndex;
    NSString *              totalStr;

    pool = [[NSAutoreleasePool alloc] init];
    assert(pool != nil);

    total = 0;
    numberCount = [immutableNumbers count];
    for (numberIndex = 0; numberIndex < numberCount; numberIndex++) {
        NSNumber *  numberObj;

        // Sleep for a while. This makes it easiest to test various problematic
cases.

        [NSThread sleepForTimeInterval:1.0];

        // Do the mathematics.

        numberObj = [immutableNumbers objectAtIndex:numberIndex];
        assert([numberObj isKindOfClass:[NSNumber class]]);

        total += [numberObj integerValue];
    }

    totalStr = [NSString stringWithFormat:@"%ld", (long) total];
    [self performSelectorOnMainThread:@selector(threadRecalculateDone:)
withObject:totalStr waitUntilDone:NO];

    [pool drain];
}

// ---- Broken Code ---- Do Not Use ---- Broken Code ---- Do Not Use ----
```

This fixes all of the crashing bugs associated with threaded recalculation. However, there are still a number of other serious bugs in this code. The most obvious is associated with the order in which results arrive. Imagine the following sequence:

1. The user adds a bunch of items to the list. This kicks off a recalculation that takes a long time (because calculation time is proportional to the length of the array). This calculation runs on thread A.

2. The user removes all but one item from the list. This kicks off another recalculation, running on thread B, that completes quickly.
3. Thread B completes and applies its results to the user interface. The user interface is now showing the correct number.
4. Thread A completes and applies its results to the user interface. This leaves the user interface showing stale results.

> **Note:** Again, this isn't something you have to imagine. The Defaults/Minimum button in Sample Code 'ListAdder' toggles between a long list and a short list, allowing you to easily reproduce this problem.

The solution to this problem is to apply some sort of tag to the recalculation operation, and only commit its results if that tag is still valid. This in itself isn't hard to do, but at this point you should start to realize that a more structured approach is needed. One good structure to use here is NSOperation, discussed in detail in the next section. However, before we go there, you should consider two more outstanding problems with the threaded code:

- cancellation — In the above scenario you end up with thread A and B running simultaneously, with thread A's results destined to be discarded. It would be nice if we could stop thread A so that it doesn't waste CPU cycles, battery life, and memory.
- thread-safe deallocation — If you run any threaded code within UIKit objects (such as a view controller), there's a chance that the object's `-dealloc` method will be called on a secondary thread. This is bad (in the Ghost Busters sense of that word), especially in something like a view controller, where the `-dealloc` method typically releases UIView objects, and hence might cause their `-dealloc` method to run on the secondary thread. You can learn more about this problem and its solutions in The Deallocation Problem.

NSOperation provides the infrastructure for solving all of the problems discussed above. In the next section we'll look at how that works.

Back to Top

NSOperation To The Rescue!

NSOperation is a class that allows you to model asynchronous operations in a structured fashion. It provides a way for the high-level parts of your application to start asynchronous operations without worrying about the details of how they are executed. An NSOperation could be run on a thread, or asynchronously via run loop callbacks, or via other, more exotic means. The important thing is that your application's high-level code does not care. It just starts the operation and gets notified when it's finished.

> **Important:** NSOperation is commonly used to run CPU intensive code on a thread, but it's not just about threads. NSOperation can be used to model any asynchronous operation. For example, Sample Code 'LinkedImageFetcher' shows how to use NSOperation for networking operations.

NSOperation encourages a programming model called **thread confinement**. In this model the resources used by the thread are owned by that thread, not shared between threads. This is a great way to write threaded code without tripping over the sorts of problems discussed in the previous section. A good way to use NSOperation is:

1. Initialize the operation with an immutable copy of the data it needs to do its work.
2. Execute the operation; while it's executing, it only operates on the data it was initialized with and on data that it creates for itself and does not share with other threads.
3. When the operation completes, it makes its results available to the rest of the application, at which point the operation itself no longer touches those results.

> **Note:** If creating an immutable copy of your data in step 1 is too expensive, you can use some internal state management to mark the data as being owned by the NSOperation, and thus avoid having other threads operate on it. This technique, which is also used by the NSOperation to pass back its results in step 3, is known as **serial thread confinement**.

Using a separate NSOperation object solves a number of problems with general threaded code:

- you can store ancillary data associated with the operation as properties of the NSOperation object
- you can use these properties, or even the NSOperation object itself, as a tag to decide whether the results of the operation are stale
- the NSOperation object is a handle by which you can cancel the operation
- you can guarantee that all threaded code runs within the operation, and hence avoid deallocation problems

NSOperation also has a number of other useful features:

- Each operation is executed by an operation queue (NSOperationQueue). Each queue has a maximum number of operations that it will execute in parallel (`maxConcurrentOperationCount`), commonly known as the **queue width**. This width defaults to a value that's appropriate for the device on which you're running, but you can set it to a value that's appropriate for the task at hand. For example, you can guarantee that operations are serialized (executed one at a time) by setting the queue width to 1. Or, if your operations hit the network, you can use the queue width to limit the number of concurrent network operations.
- You can create operation queues to model your problem space. For example, a file copying program could use one operation queue per disk to avoid thrashing the disk head.
- You can make operations dependent on other operations, and thereby establish chains of operation execution and operation fan in.

It's important to realize that NSOperation and NSOperationQueue don't do anything that you couldn't do yourself. If you really wanted to, you could replicate all of this structure with your own code. However, if you don't have an existing structure for managing asynchronous code, and you don't feel like reimplementing the wheel, NSOperation is for you.

The rest of this section describes how Sample Code 'ListAdder' uses NSOperation to implement asynchronous recalculation.

Using NSOperation in ListAdder

The next few sections describe the minimum amount of work necessary to adopt NSOperation in Sample Code 'ListAdder'.

A Minimal Header

The first step in implementing asynchronous recalculation using NSOperation is to create a subclass of NSOperation that does the calculation. You should pay special attention to the operation's interface, as expressed in its header file. Most importantly, everything in that interface should have a well-defined thread safeness strategy.

Listing 6 is the absolute minimal interface required by the AdderOperation from Sample Code 'ListAdder'.

Listing 6: Minimal operation interface

```
@interface AdderOperation : NSOperation  - (id)initWithNumbers:(NSArray *)numbers;
// only meaningful after the operation is finished  @property (copy,   readonly )
NSString * formattedTotal;  @end
```

The operation's initialization method (in this example, `-initWithNumbers:`) should take, as parameters, all the information that's absolutely necessary for the operation to run. In this case, it

simply takes the list of numbers to add.

The only other item in this minimal interface is the properties that the client can use to get at the operation's result. In this case the `formattedTotal` property is the string that's meant to be displayed in the total value of the list view.

An operation inherits various important properties from NSOperation. The most critical is the `isFinished` property. The client can observe this property (in the key-value observing (KVO) sense) to determine when the operation is done. Alternatively, if you're not a big fan of KVO, you can use other mechanisms for the operation to signal that it's done. For example, your operation might post a notification (using NSNotificationCenter) when it's done, or it might define a delegate protocol.

> **Warning:** Regardless of how your operation signals that it's done, your client must be aware of the thread on which that signalling takes place. This issue is discussed in more detail below.

There's one other thing to note about the minimal header shown in Listing 6: the `formattedTotal` property is atomic. That's because there's a distinct possibility that this property will be accessed from multiple threads concurrently, and the property must be atomic for that to be safe. As a rule, all public properties of an operation should be atomic.

Minimal Implementation

Given this header, the operation must include an implementation for the following:

- `-initWithNumbers:` method — This is a little subtle, and is discussed in detail next.
- `formattedTotal` property — This property can just be synthesized.
- `-dealloc` method — There are no big surprises here; the only gotcha is that the method may be executed by any thread.
- `-main` method — This is where your operation does its computation, and is discussed in detail below.

Listing 7 shows a minimal implementation of the `-initWithNumbers:` method.

Listing 7: A minimal initialization method

```
- (id)initWithNumbers:(NSArray *)numbers
{
    assert(numbers != nil);
    self = [super init];
    if (self != nil) {
        self->_numbers = [numbers copy];
        assert(self->_numbers != nil);
    }
    return self;
}
```

There are two points to note here:

- The method creates an immutable copy of the incoming `numbers` array. This prevents the problem discussed earlier, where the main thread might mutate the array while the secondary thread running the operation is working on it.
- It's your decision as to whether to make your initialization method thread safe. Because you control the allocation of the operation, you can guarantee that it will be called on the main thread (or some other specific secondary thread, for that matter). However, if it's relatively easy to make your initialization method thread safe, you should do so.

The operation's `-main` method does the actual computation. Listing 8 shows a minimal implementation.

Listing 8: A minimal main method

```
- (void)main
{
    NSUInteger      numberCount;
    NSUInteger      numberIndex;
    NSInteger       total;

    // This method is called by a thread that's set up for us by the
NSOperationQueue.

    assert( ! [NSThread isMainThread] );

    // Do the heavy lifting (-:

    total = 0;
    numberCount = [self.numbers count];
    for (numberIndex = 0; numberIndex < numberCount; numberIndex++) {
        NSNumber *  numberObj;

        // Check for cancellation.

        if ([self isCancelled]) {
            break;
        }

        // Sleep for a second. This makes it easiest to test cancellation
        // and so on.

        [NSThread sleepForTimeInterval:1.0];

        // Do the mathematics.

        numberObj = [self.numbers objectAtIndex:numberIndex];
        assert([numberObj isKindOfClass:[NSNumber class]]);

        total += [numberObj integerValue];
    }

    // Set our output properties base on the value we calculated. Our client
    // shouldn't look at these until -isFinished goes to YES (which happens when
    // we return from this method).

    self.formattedTotal = [self.formatter stringFromNumber:[NSNumber
numberWithInteger:total]];
}
```

The critical points here include:

- This method is expected to run on a secondary thread. Everything it touches must have some sort of thread safeness strategy. In this example:
  - The `numbers` property is thread safe because it can only be changed by the `-initWithNumbers:` method, which must necessarily have completed before this code can run.
  - The `numberCount`, `numberIndex`, `total` and `numberObj` variables are safe because they are local variables, and thus won't be shared between threads.
  - The `formattedTotal` property is safe because the client should not be looking at it until the operation finishes. Even if the client ignores this requirement, their access will be safe because the property is atomic: the client will either get the original value (that is, `nil`) or the correct value, never some weird mishmash.
- The operation tests for cancellation by calling `isCancelled` within its main computation loop.

Important: While it is not strictly necessary for a standard operation to check for cancellation, is it strongly recommended that you do this. If an operation does not periodically check for cancellation, it will waste resources computing results that are no longer needed.

Supporting cancellation in concurrent operations is a bit trickier. If you're creating a concurrent operation, see Concurrent Operations for information on how to do this correctly.

Using the Operation

Once you have an operation that does the task at hand, you have to use it in your application. The first step is creating an NSOperationQueue on which to run the operation. In the case of Sample Code 'ListAdder', the main view controller declares a private property for the queue and then initializes it when the view controller is initialized. Listing 9 shows the details.

Listing 9: Creating the NSOperation queue

```
@interface ListAdderViewController () [...]

@property (nonatomic, retain, readonly ) NSOperationQueue * queue;
@property (nonatomic, retain, readwrite) AdderOperation *   inProgressAdder;

[...]

@end

@implementation ListAdderViewController

[...]

- (id)init
{
    self = [super initWithStyle:UITableViewStyleGrouped];
    if (self != nil) {
        [...]

        self->_queue = [[NSOperationQueue alloc] init];
        assert(self->_queue != nil);

        [...]
    }
    return self;
}

[...]

@end
```

> **Warning:** Hosting an operation queue within a view controller is tricky if that view controller can ever be deallocated. In Sample Code 'ListAdder' that's not possible because ListAdderViewController is the root view controller, and thus can't be deallocated. If your view controller might be deallocated, you must take steps to avoid its `-dealloc` method running on a secondary thread. See The Deallocation Problem for details.

Once you have an operation queue, you can start operations by simply adding them to the queue. Listing 10 shows this process.

Listing 10: Queuing the operation

```
- (void)recalculateTotalUsingOperation
{
    // If we're already calculating, cancel that operation.

    if (self.inProgressAdder != nil) {
        [self.inProgressAdder cancel];
    }

    // Start up a replacement operation.

    self.inProgressAdder = [[[AdderOperation alloc] initWithNumbers:self.numbers]
autorelease];
    assert(self.inProgressAdder != nil);

    [self.inProgressAdder addObserver:self
        forKeyPath:@"isFinished"
        options:0
        context:&self->_formattedTotal
    ];
```

```
    [self.queue addOperation:self.inProgressAdder];

    // The user interface is adjusted by a KVO observer on recalculating.

    self.recalculating = YES;
}
```

There are a couple of things to note here:

- The code keeps track of the most recently queued operation in the `inProgressAdder` property. When it starts a new operation, it replaces that value with the operation it just started. This has two benefits:
  - When it starts an operation, it can cancel the previous one. This means that the new operation doesn't just keep executing, consuming valuable CPU cycles, battery life, and memory.
  - When an operation completes, it can determine if the operation's results are still meaningful. More on this below.
- Before queuing an operation, it adds an observation of the `isFinished` property. This lets it determine when the operation is complete.

Finally, when the operation is complete, you must check and commit the results. Listing 11 shows how Sample Code 'ListAdder' does this.

Listing 11: Committing the operation results

```
- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context
{
    if (context == &self->_formattedTotal) {
        AdderOperation *    op;

        // If the operation has finished, call -adderOperationDone: on the main
thread to deal
        // with the results.

        // can be running on any thread
        assert([keyPath isEqual:@"isFinished"]);
        op = (AdderOperation *) object;
        assert([op isKindOfClass:[AdderOperation class]]);
        assert([op isFinished]);

        [self performSelectorOnMainThread:@selector(adderOperationDone:)
            withObject:op
            waitUntilDone:NO
        ];
    } [...]
}

- (void)adderOperationDone:(AdderOperation *)op
{
    assert([NSThread isMainThread]);

    assert(self.recalculating);

    // Always remove our observer, regardless of whether we care about
    // the results of this operation.

    [op removeObserver:self forKeyPath:@"isFinished"];

    // Check to see whether these are the results we're looking for.
    // If not, we just discard the results; later on we'll be notified
    // of the latest add operation completing.

    if (op == self.inProgressAdder) {
        assert( ! [op isCancelled] );

        // Commit the value to our model.

        self.formattedTotal = op.formattedTotal;

        // Clear out our record of the operation. The user interface is adjusted
```

```
        // by a KVO observer on recalculating.

        self.inProgressAdder = nil;
        self.recalculating = NO;
    }
}
```

There are two parts to this process. First, when the operation's `isFinished` property changes, KVO calls the `-observeValueForKeyPath:ofObject:change:context:` method. This method runs on the same thread as the code that changed the `isFinished` property, which in this case means it's running on a secondary thread. It can't manipulate the user interface from a secondary thread, so it defers the work to the main thread by using `-performSelectorOnMainThread:withObject:waitUntilDone:` to call the `-adderOperationDone:` method.

The `-adderOperationDone:` method does three things:

- It removes the observation of the `isFinished` property. This balances out the code that added the observation in Listing 10.
- It checks to see whether the results of the operation are still meaningful by comparing the completed operation (`op`) against the most recently started operation (`inProgressAdder`). If these don't match, the operation's results don't matter, and are discarded.
- If the operation's results are still meaningful, they are committed to the user interface.

One non-obvious aspect of this whole process is that the `inProgressAdder` property is only ever accessed by the main thread. This means that it does not have to be atomic but, more importantly, it doesn't need any concurrency control. Code sequences like that shown in Listing 12 would not be valid without this implicit serialization.

Listing 12: Valid without locking because of main thread serialization

```
if (op == self.inProgressAdder) {     [... do something with inProgressAdder ...]
self.inProgressAdder = nil; }
```

This wraps up the basic process for using NSOperation to run long-running computations asynchronously. The next section is going to go beyond the minimal implementation shown earlier, and cover some more realistic situations.

AdderOperation: Beyond the Basics

Listing 6 showed the minimal interface to AdderOperation. This is a cut down version of the interface actually used by Sample Code 'ListAdder'. The full interface is shown in Listing 13.

Listing 13: Actual operation interface

```
@interface AdderOperation : NSOperation
{
    [... instance variables elided ...]
}

- (id)initWithNumbers:(NSArray *)numbers;

// set up by the init method that can't be changed

@property (copy,   readonly ) NSArray *         numbers;
@property (assign, readonly ) NSUInteger        sequenceNumber;

// must be configured before the operation is started

@property (assign, readwrite) NSTimeInterval    interNumberDelay;

// only meaningful after the operation is finished

@property (assign, readonly ) NSInteger         total;
@property (copy,   readonly ) NSString *        formattedTotal;
```

```
@end
```

Moreover, this interface is extended with a bunch of internal properties declared in the class extension.

Listing 14: Operation class extension

```
@interface AdderOperation ()

// only accessed by the operation thread

@property (retain, readwrite) NSNumberFormatter *   formatter;

// read/write versions of public properties

@property (assign, readwrite) NSInteger             total;
@property (copy,   readwrite) NSString *            formattedTotal;

@end
```

The thing to note here is that every property declared in these interfaces has a well-defined thread safeness strategy. The strategies include:

- **immutability** — Some properties are set up at initialization time and not modified thereafter. The `numbers` array is a good example of this.
- **configuration time only** — Some properties, like `interNumberDelay`, can be safely set between the point where the operation is created and the point where the operation is started. After that point it's either not safe or not effective to modify such properties.
- **thread confinement** — Some properties are thread safe by virtue of being accessed only by the thread running the operation. The `formatter` property is an example of this technique.
- **serial thread confinement** — Some properties, like `total` and `formattedTotal`, are initially only used by the operation and then, when the operation is finished, are available to other code.

Important: When implementing threaded code, is is critical to have a thread safeness strategy for every piece of data you touch from your threaded code. NSOperation makes it easier to develop such strategies.

Thread Confinement

It's important to realize that the AdderOperation `formatter` property is a rather artificial example of thread confinement. This is necessary because of the very limited nature of Sample Code 'ListAdder' itself. In a real application thread confinement is a much more powerful technique. For example, Sample Code 'SeismicXML' has an operation to parse XML asynchronously. Within this operation there is an NSXMLParser object, and within that object is a libxml2 XML parser object. Those objects can be called from an arbitrary thread, but they can only be called from one thread at a time. Thread confinement makes it possible to safely use objects like this from threaded code, and NSOperation is a great way of implementing thread confinement.

Back to Top

NSOperations Hints and Tips

This section contains a bunch of general hints and tips for using NSOperation.

NSOperation and GCD

It's not immediately obvious how Grand Central Dispatch (GCD), introduced in iOS 4, relates to NSOperation. The short answer is that these two technologies complement each other nicely. GCD is a low-level API that gives you the flexibility to structure your code in a variety of different ways. In contrast, NSOperation provides you with a default structure that you can use for your asynchronous code. If you're looking for an existing, well-defined structure that's perfectly tailored for Cocoa applications, use NSOperation. If you're looking to create your own structure that exactly matches your problem space, use GCD.

The Deallocation Problem

One of the biggest problems with using secondary threads from a UIKit object, like a view controller, is ensuring that your object is deallocated safely. This section explains how this problem arises, and what you can do about it.

When you start a secondary thread, it's common for that thread to retain the target object. This happens in numerous circumstances, including:

- when you start a secondary thread with any of the following methods:
  - `-performSelectorInBackground:withObject:`
  - `-performSelector:onThread:withObject:waitUntilDone:`
  - `-performSelector:onThread:withObject:waitUntilDone:modes:`
- when you start a secondary thread with NSThread
- when you run a block asynchronously and the block references `self` or an instance variable

When a secondary thread retains the target object, you have to ensure that the thread releases that reference before the main thread releases its last reference to the object. If you don't do this, the last reference to the object is released by the secondary thread, which means that the object's `-dealloc` method runs on that secondary thread. This is problematic if the object's `-dealloc` method does things that are not safe to do on a secondary thread, something that's common for UIKit objects like a view controller.

For a concrete example of this, consider the code in Listing 15.

Listing 15: An example of the deallocation problem

```
- (IBAction)buttonAction:(id)sender
{
    #pragma unused(sender)
    [self performSelectorInBackground:@selector(recalculate) withObject:nil];
}

- (void)recalculate
{
    while ( ! self.cancelled ) {
        [... calculate ...]
    }
}

- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
    self.cancelled = YES;
    // race starts here
}
```

In Listing 15 the `-viewWillDisappear:` method sets `cancelled` to stop the secondary thread executing its calculation. That starts a race between the main thread, which is busily tearing down the view controller, and the secondary thread, which is noticing the `cancelled` property being set and actually quitting. In most circumstances the secondary thread will win the race, release its reference first, and everything will be OK. However, if the main thread wins the race, and releases its reference before the secondary thread, then the secondary thread's release will be the last release, and the view controller's `-dealloc` method will run on a secondary thread.

You might think to get around this problem by polling the secondary thread's `isFinished` property to ensure that it finishes before returning from `-viewWillDisappear:`. However, due to obscure implementation details within NSThread, that's not guaranteed to fix the problem.

Similar problems can arise when you use key-value observing (KVO) to observe the `isFinished` property of an NSOperation. While KVO does not retain either the observer or the observee, it's still possible that, even if you remove the observer in your `-viewWillDisappear:` method, a KVO notification might already be in flight for your object. If that happens, the thread running the notification could end up calling a deallocated object!

Solving this problem in the general case is quite tricky. However, if you restrict yourself to using NSOperation, there are two relatively simple paths to a solution:

- do all your key-value observing in a persistent object, one that's never deallocated
- use the QWatchedOperationQueue class from Sample Code 'LinkedImageFetcher'

Concurrent Operations

NSOperation supports two types of operations:

- **standard operations** — These operations, also known as **non-concurrent operations**, require the NSOperationQueue to provide concurrency on their behalf. NSOperationQueue organizes to run such operations on a thread.
- **concurrent operations** — These operations bring their own concurrency. NSOperationQueue does not have to dedicate a thread to running such operations.

Standard operations are a great way to run tasks asynchronously when the underlying facilities are synchronous. They are typically used for long-running computations, but they can also be useful for fast, reliable I/O (like disk I/O).

In contrast, concurrent operations are great when the underlying facilities are asynchronous—there's no point tying up a thread to wait for an asynchronous API to finish. A good example of a concurrent operation is one that executes an HTTP request using the NSURLConnection API.

Implementing a concurrent operation correctly is a little tricky. You should look at Sample Code 'LinkedImageFetcher' for an example of how to do this.

> **Important:** The beauty of NSOperation is that it allows you to model the asynchronous operation abstractly, regardless of the underlying implementation technique. Your high-level code just starts the operation and waits for it to complete; it doesn't care how the operation actually gets the work done. For example:

- the operation might run on a thread
- the operation might run asynchronously, courtesy of some run loop based API
- the operation might run asynchronously, courtesy of some GCD based API
- the operation might run in a separate process

Moreover, the operation can change how it runs without requiring the high-level code to change. For example, it would be very easy to change the Sample Code 'ListAdder' to call a web service to perform the addition. Only the AdderOperation would need to change; the high-level code would be exactly the same.

Unfinished Operations

Some poorly implemented operations trigger KVO notification of `isFinished` before they are actually finished. If you're using KVO to determine whether the operation is finished, it's a good idea to get the `isFinished` property and confirm that the operation is finished before you move on, even if it's only in a debug-time assertion. For an example of this, check out the code in Listing 11.

Back to Top

Document Revision History

Date	Notes
2010-08-27	New document that describes how Cocoa application developers can use NSOperation to solve many of the problems inherent in threaded code.