Operation Queues

Cocoa operations are an object-oriented way to encapsulate work that you want to perform asynchronously. Operations are designed to be used either in conjunction with an operation queue or by themselves. Because they are Objective-C based, operations are most commonly used in Cocoa-based applications in OS X and iOS.

This chapter shows you how to define and use operations.

About Operation Objects

An *operation object* is an instance of the `NSOperation` class (in the Foundation framework) that you use to encapsulate work you want your application to perform. The `NSOperation` class itself is an abstract base class that must be subclassed in order to do any useful work. Despite being abstract, this class does provide a significant amount of infrastructure to minimize the amount of work you have to do in your own subclasses. In addition, the Foundation framework provides two concrete subclasses that you can use as-is with your existing code. Table 2-1 lists these classes, along with a summary of how you use each one.

Table 2-1 Operation classes of the Foundation framework

Class	Description
`NSInvocationOperation`	A class you use as-is to create an operation object based on an object and selector from your application. You can use this class in cases where you have an existing method that already performs the needed task. Because it does not require subclassing, you can also use this class to create operation objects in a more dynamic fashion. For information about how to use this class, see Creating an NSInvocationOperation Object.
`NSBlockOperation`	A class you use as-is to execute one or more block objects concurrently. Because it can execute more than one block, a block operation object operates using a group semantic; only when all of the associated blocks have finished executing is the operation itself considered finished. For information about how to use this class, see Creating an NSBlockOperation Object. This class is available in OS X v10.6 and later. For more information about blocks, see *Blocks Programming Topics*.
`NSOperation`	The base class for defining custom operation objects. Subclassing `NSOperation` gives you complete control over the implementation of your own operations, including the ability to alter the default way in which your operation executes and reports its status. For information about how to define custom operation objects, see Defining a Custom Operation Object.

All operation objects support the following key features:

- Support for the establishment of graph-based dependencies between operation objects. These dependencies prevent a given operation from running until all of the operations on which it depends have finished running. For information about how to configure dependencies, see Configuring Interoperation Dependencies.

- Support for an optional completion block, which is executed after the operation's main task finishes. (OS X v10.6 and later only.) For information about how to set a completion block, see Setting Up a Completion Block.
- Support for monitoring changes to the execution state of your operations using KVO notifications. For information about how to observe KVO notifications, see *Key-Value Observing Programming Guide*.
- Support for prioritizing operations and thereby affecting their relative execution order. For more information, see Changing an Operation's Execution Priority.
- Support for canceling semantics that allow you to halt an operation while it is executing. For information about how to cancel operations, see Canceling Operations. For information about how to support cancellation in your own operations, see Responding to Cancellation Events.

Operations are designed to help you improve the level of concurrency in your application. Operations are also a good way to organize and encapsulate your application's behavior into simple discrete chunks. Instead of running some bit of code on your application's main thread, you can submit one or more operation objects to a queue and let the corresponding work be performed asynchronously on one or more separate threads.

Concurrent Versus Non-concurrent Operations

Although you typically execute operations by adding them to an operation queue, doing so is not required. It is also possible to execute an operation object manually by calling its `start` method, but doing so does not guarantee that the operation runs concurrently with the rest of your code. The `isConcurrent` method of the `NSOperation` class tells you whether an operation runs synchronously or asynchronously with respect to the thread in which its `start` method was called. By default, this method returns `NO`, which means the operation runs synchronously in the calling thread.

If you want to implement a *concurrent operation*—that is, one that runs asynchronously with respect to the calling thread—you must write additional code to start the operation asynchronously. For example, you might spawn a separate thread, call an asynchronous system function, or do anything else to ensure that the `start` method starts the task and returns immediately and, in all likelihood, before the task is finished.

Most developers should never need to implement concurrent operation objects. If you always add your operations to an operation queue, you do not need to implement concurrent operations. When you submit a nonconcurrent operation to an operation queue, the queue itself creates a thread on which to run your operation. Thus, adding a nonconcurrent operation to an operation queue still results in the asynchronous execution of your operation object code. The ability to define concurrent operations is only necessary in cases where you need to execute the operation asynchronously without adding it to an operation queue.

For information about how to create a concurrent operation, see Configuring Operations for Concurrent Execution and *NSOperation Class Reference*.

Creating an NSInvocationOperation Object

The `NSInvocationOperation` class is a concrete subclass of `NSOperation` that, when run, invokes the selector you specify on the object you specify. Use this class to avoid defining large numbers of custom operation objects for each task in your application; especially if you are modifying an existing application and already have the objects and methods needed to perform the necessary tasks. You can also use it when the method you want to call can change depending on the circumstances. For example, you could use an invocation operation to perform a selector that is chosen dynamically based on user input.

The process for creating an invocation operation is straightforward. You create and initialize a new instance of the class, passing the desired object and selector to execute to the initialization method. Listing 2-1 shows two methods from a custom class that demonstrate the creation process. The

`taskWithData:` method creates a new invocation object and supplies it with the name of another method, which contains the task implementation.

Listing 2-1 Creating an `NSInvocationOperation` object

```
@implementation MyCustomClass

- (NSOperation*)taskWithData:(id)data {

    NSInvocationOperation* theOp = [[NSInvocationOperation alloc]
initWithTarget:self

                    selector:@selector(myTaskMethod:) object:data];


   return theOp;

}


// This is the method that does the actual work of the task.

- (void)myTaskMethod:(id)data {

    // Perform the task.

}

@end
```

Creating an NSBlockOperation Object

The `NSBlockOperation` class is a concrete subclass of `NSOperation` that acts as a wrapper for one or more block objects. This class provides an object-oriented wrapper for applications that are already using operation queues and do not want to create dispatch queues as well. You can also use block operations to take advantage of operation dependencies, KVO notifications, and other features that might not be available with dispatch queues.

When you create a block operation, you typically add at least one block at initialization time; you can add more blocks as needed later. When it comes time to execute an `NSBlockOperation` object, the object submits all of its blocks to the default-priority, concurrent dispatch queue. The object then waits until all of the blocks finish executing. When the last block finishes executing, the operation object marks itself as finished. Thus, you can use a block operation to track a group of executing blocks, much like you would use a thread join to merge the results from multiple threads. The difference is that because the block operation itself runs on a separate thread, your application's other threads can continue doing work while waiting for the block operation to complete.

Listing 2-2 shows a simple example of how to create an `NSBlockOperation` object. The block itself has no parameters and no significant return result.

Listing 2-2 Creating an `NSBlockOperation` object

```
NSBlockOperation* theOp = [NSBlockOperation blockOperationWithBlock: ^{
      NSLog(@"Beginning operation.\n");
      // Do some work.
   }];
```

After creating a block operation object, you can add more blocks to it using the `addExecutionBlock:` method. If you need to execute blocks serially, you must submit them directly to the desired dispatch queue.

Defining a Custom Operation Object

If the block operation and invocation operation objects do not quite meet the needs of your application, you can subclass `NSOperation` directly and add whatever behavior you need. The `NSOperation` class provides a general subclassing point for all operation objects. The class also provides a significant amount of infrastructure to handle most of the work needed for dependencies and KVO notifications. However, there may still be times when you need to supplement the existing infrastructure to ensure that your operations behave correctly. The amount of extra work you have to do depends on whether you are implementing a nonconcurrent or a concurrent operation.

Defining a nonconcurrent operation is much simpler than defining a concurrent operation. For a nonconcurrent operation, all you have to do is perform your main task and respond appropriately to cancellation events; the existing class infrastructure does all of the other work for you. For a concurrent operation, you must replace some of the existing infrastructure with your custom code. The following sections show you how to implement both types of object.

Performing the Main Task

At a minimum, every operation object should implement at least the following methods:

- A custom initialization method
- `main`

You need a custom initialization method to put your operation object into a known state and a custom `main` method to perform your task. You can implement additional methods as needed, of course, such as the following:

- Custom methods that you plan to call from the implementation of your `main` method
- Accessor methods for setting data values and accessing the results of the operation
- Methods of the `NSCoding` protocol to allow you to archive and unarchive the operation object

Listing 2-3 shows a starting template for a custom `NSOperation` subclass. (This listing does not show how to handle cancellation but does show the methods you would typically have. For information about handling cancellation, see Responding to Cancellation Events.) The initialization method for this class takes a single object as a data parameter and stores a reference to it inside the operation object. The `main` method would ostensibly work on that data object before returning the results back to your application.

Listing 2-3 Defining a simple operation object

```
@interface MyNonConcurrentOperation : NSOperation
@property id (strong) myData;
-(id)initWithData:(id)data;
@end

@implementation MyNonConcurrentOperation
- (id)initWithData:(id)data {
   if (self = [super init])
      myData = data;
   return self;
}

-(void)main {
   @try {
      // Do some work on myData and report the results.
   }
```

```
    @catch(...) {
        // Do not rethrow exceptions.
    }
}
@end
```

For a detailed example of how to implement an `NSOperation` subclass, see *NSOperationSample*.

Responding to Cancellation Events

After an operation begins executing, it continues performing its task until it is finished or until your code explicitly cancels the operation. Cancellation can occur at any time, even before an operation begins executing. Although the `NSOperation` class provides a way for clients to cancel an operation, recognizing the cancellation event is voluntary by necessity. If an operation were terminated outright, there might not be a way to reclaim resources that had been allocated. As a result, operation objects are expected to check for cancellation events and to exit gracefully when they occur in the middle of the operation.

To support cancellation in an operation object, all you have to do is call the object's `isCancelled` method periodically from your custom code and return immediately if it ever returns `YES`. Supporting cancellation is important regardless of the duration of your operation or whether you subclass `NSOperation` directly or use one of its concrete subclasses. The `isCancelled` method itself is very lightweight and can be called frequently without any significant performance penalty. When designing your operation objects, you should consider calling the `isCancelled` method at the following places in your code:

- Immediately before you perform any actual work
- At least once during each iteration of a loop, or more frequently if each iteration is relatively long
- At any points in your code where it would be relatively easy to abort the operation

Listing 2-4 provides a very simple example of how to respond to cancellation events in the `main` method of an operation object. In this case, the `isCancelled` method is called each time through a `while` loop, allowing for a quick exit before work begins and again at regular intervals.

Listing 2-4 Responding to a cancellation request

```
- (void)main {
    @try {
        BOOL isDone = NO;

        while (![self isCancelled] && !isDone) {
            // Do some work and set isDone to YES when finished
        }
    }
    @catch(...) {
        // Do not rethrow exceptions.
    }
}
```

Although the preceding example contains no cleanup code, your own code should be sure to free up any resources that were allocated by your custom code.

Configuring Operations for Concurrent Execution

Operation objects execute in a synchronous manner by default—that is, they perform their task in the thread that calls their `start` method. Because operation queues provide threads for nonconcurrent operations, though, most operations still run asynchronously. However, if you plan to execute operations manually and still want them to run asynchronously, you must take the appropriate actions to ensure that they do. You do this by defining your operation object as a concurrent operation.

Table 2-2 lists the methods you typically override to implement a concurrent operation.

Table 2-2 Methods to override for concurrent operations

Method	Description
`start`	(Required) All concurrent operations must override this method and replace the default behavior with their own custom implementation. To execute an operation manually, you call its `start` method. Therefore, your implementation of this method is the starting point for your operation and is where you set up the thread or other execution environment in which to execute your task. Your implementation must not call `super` at any time.
`main`	(Optional) This method is typically used to implement the task associated with the operation object. Although you could perform the task in the `start` method, implementing the task using this method can result in a cleaner separation of your setup and task code.
`isExecuting` `isFinished`	(Required) Concurrent operations are responsible for setting up their execution environment and reporting the status of that environment to outside clients. Therefore, a concurrent operation must maintain some state information to know when it is executing its task and when it has finished that task. It must then report that state using these methods. Your implementations of these methods must be safe to call from other threads simultaneously. You must also generate the appropriate KVO notifications for the expected key paths when changing the values reported by these methods.
`isConcurrent`	(Required) To identify an operation as a concurrent operation, override this method and return `YES`.

The rest of this section shows a sample implementation of the `MyOperation` class, which demonstrates the fundamental code needed to implement a concurrent operation. The `MyOperation` class simply executes its own `main` method on a separate thread that it creates. The actual work that the `main` method performs is irrelevant. The point of the sample is to demonstrate the infrastructure you need to provide when defining a concurrent operation.

Listing 2-5 shows the interface and part of the implementation of the `MyOperation` class. The implementations of the `isConcurrent`, `isExecuting`, and `isFinished` methods for the `MyOperation` class are relatively straightforward. The `isConcurrent` method should simply return `YES` to indicate that this is a concurrent operation. The `isExecuting` and `isFinished` methods simply return values stored in instance variables of the class itself.

Listing 2-5 Defining a concurrent operation

```
@interface MyOperation : NSOperation {
    BOOL        executing;
    BOOL        finished;
}
- (void)completeOperation;
@end

@implementation MyOperation
```

```
- (id)init {
    self = [super init];
    if (self) {
        executing = NO;
        finished = NO;
    }
    return self;
}

- (BOOL)isConcurrent {
    return YES;
}

- (BOOL)isExecuting {
    return executing;
}

- (BOOL)isFinished {
    return finished;
}
@end
```

Listing 2-6 shows the `start` method of `MyOperation`. The implementation of this method is minimal so as to demonstrate the tasks you absolutely must perform. In this case, the method simply starts up a new thread and configures it to call the `main` method. The method also updates the `executing` member variable and generates KVO notifications for the `isExecuting` key path to reflect the change in that value. With its work done, this method then simply returns, leaving the newly detached thread to perform the actual task.

Listing 2-6 The start method

```
- (void)start {
   // Always check for cancellation before launching the task.
   if ([self isCancelled])
   {
      // Must move the operation to the finished state if it is canceled.
      [self willChangeValueForKey:@"isFinished"];
      finished = YES;
      [self didChangeValueForKey:@"isFinished"];
      return;
   }

   // If the operation is not canceled, begin executing the task.
   [self willChangeValueForKey:@"isExecuting"];
   [NSThread detachNewThreadSelector:@selector(main) toTarget:self withObject:nil];
   executing = YES;
   [self didChangeValueForKey:@"isExecuting"];
}
```

Listing 2-7 shows the remaining implementation for the `MyOperation` class. As was seen in Listing 2-6, the `main` method is the entry point for a new thread. It performs the work associated with the operation object and calls the custom `completeOperation` method when that work is finally done. The `completeOperation` method then generates the needed KVO notifications for both the `isExecuting` and `isFinished` key paths to reflect the change in state of the operation.

Listing 2-7 Updating an operation at completion time

```
- (void)main {
    @try {

        // Do the main work of the operation here.

        [self completeOperation];
    }
    @catch(...) {
       // Do not rethrow exceptions.
    }
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];

    executing = NO;
    finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}
```

Even if an operation is canceled, you should always notify KVO observers that your operation is now finished with its work. When an operation object is dependent on the completion of other operation objects, it monitors the `isFinished` key path for those objects. Only when all objects report that they are finished does the dependent operation signal that it is ready to run. Failing to generate a finish notification can therefore prevent the execution of other operations in your application.

Maintaining KVO Compliance

The `NSOperation` class is key-value observing (KVO) compliant for the following key paths:

- `isCancelled`
- `isConcurrent`
- `isExecuting`
- `isFinished`
- `isReady`
- `dependencies`
- `queuePriority`
- `completionBlock`

If you override the `start` method or do any significant customization of an `NSOperation` object other than override `main`, you must ensure that your custom object remains KVO compliant for these

key paths. When overriding the `start` method, the key paths you should be most concerned with are `isExecuting` and `isFinished`. These are the key paths most commonly affected by reimplementing that method.

If you want to implement support for dependencies on something besides other operation objects, you can also override the `isReady` method and force it to return `NO` until your custom dependencies were satisfied. (If you implement custom dependencies, be sure to call `super` from your `isReady` method if you still support the default dependency management system provided by the `NSOperation` class.) When the readiness status of your operation object changes, generate KVO notifications for the `isReady` key path to report those changes. Unless you override the `addDependency:` or `removeDependency:` methods, you should not need to worry about generating KVO notifications for the `dependencies` key path.

Although you could generate KVO notifications for other key paths of `NSOperation`, it is unlikely you would ever need to do so. If you need to cancel an operation, you can simply call the existing `cancel` method to do so. Similarly, there should be little need for you to modify the queue priority information in an operation object. Finally, unless your operation is capable of changing its concurrency status dynamically, you do not need to provide KVO notifications for the `isConcurrent` key path.

For more information on key-value observing and how to support it in your custom objects, see *Key-Value Observing Programming Guide*.

Customizing the Execution Behavior of an Operation Object

The configuration of operation objects occurs after you have created them but before you add them to a queue. The types of configurations described in this section can be applied to all operation objects, regardless of whether you subclassed `NSOperation` yourself or used an existing subclass.

Configuring Interoperation Dependencies

Dependencies are a way for you to serialize the execution of distinct operation objects. An operation that is dependent on other operations cannot begin executing until all of the operations on which it depends have finished executing. Thus, you can use dependencies to create simple one-to-one dependencies between two operation objects or to build complex object dependency graphs.

To establish dependencies between two operation objects, you use the `addDependency:` method of `NSOperation`. This method creates a one-way dependency from the current operation object to the target operation you specify as a parameter. This dependency means that the current object cannot begin executing until the target object finishes executing. Dependencies are also not limited to operations in the same queue. Operation objects manage their own dependencies and so it is perfectly acceptable to create dependencies between operations and add them all to different queues. One thing that is not acceptable, however, is to create circular dependencies between operations. Doing so is a programmer error that will prevent the affected operations from ever running.

When all of an operation's dependencies have themselves finished executing, an operation object normally becomes ready to execute. (If you customize the behavior of the `isReady` method, the readiness of the operation is determined by the criteria you set.) If the operation object is in a queue, the queue may start executing that operation at any time. If you plan to execute the operation manually, it is up to you to call the operation's `start` method.

> **Important:** You should always configure dependencies before running your operations or adding them to an operation queue. Dependencies added afterward may not prevent a given operation object from running.

Dependencies rely on each operation object sending out appropriate KVO notifications whenever the status of the object changes. If you customize the behavior of your operation objects, you may need to generate appropriate KVO notifications from your custom code in order to avoid causing issues

with dependencies. For more information on KVO notifications and operation objects, see Maintaining KVO Compliance. For additional information on configuring dependencies, see *NSOperation Class Reference*.

Changing an Operation's Execution Priority

For operations added to a queue, execution order is determined first by the readiness of the queued operations and then by their relative priority. Readiness is determined by an operation's dependencies on other operations, but the priority level is an attribute of the operation object itself. By default, all new operation objects have a "normal" priority, but you can increase or decrease that priority as needed by calling the object's `setQueuePriority:` method.

Priority levels apply only to operations in the same operation queue. If your application has multiple operation queues, each prioritizes its own operations independently of any other queues. Thus, it is still possible for low-priority operations to execute before high-priority operations in a different queue.

Priority levels are not a substitute for dependencies. Priorities determine the order in which an operation queue starts executing only those operations that are currently ready. For example, if a queue contains both a high-priority and low-priority operation and both operations are ready, the queue executes the high-priority operation first. However, if the high-priority operation is not ready but the low-priority operation is, the queue executes the low-priority operation first. If you want to prevent one operation from starting until another operation has finished, you must use dependencies (as described in Configuring Interoperation Dependencies) instead.

Changing the Underlying Thread Priority

In OS X v10.6 and later, it is possible to configure the execution priority of an operation's underlying thread. Thread policies in the system are themselves managed by the kernel, but in general higher-priority threads are given more opportunities to run than lower-priority threads. In an operation object, you specify the thread priority as a floating-point value in the range 0.0 to 1.0, with 0.0 being the lowest priority and 1.0 being the highest priority. If you do not specify an explicit thread priority, the operation runs with the default thread priority of 0.5.

To set an operation's thread priority, you must call the `setThreadPriority:` method of your operation object before adding it to a queue (or executing it manually). When it comes time to execute the operation, the default `start` method uses the value you specified to modify the priority of the current thread. This new priority remains in effect for the duration of your operation's `main` method only. All other code (including your operation's completion block) is run with the default thread priority. If you create a concurrent operation, and therefore override the `start` method, you must configure the thread priority yourself.

Setting Up a Completion Block

In OS X v10.6 and later, an operation can execute a completion block when its main task finishes executing. You can use a completion block to perform any work that you do not consider part of the main task. For example, you might use this block to notify interested clients that the operation itself has completed. A concurrent operation object might use this block to generate its final KVO notifications.

To set a completion block, use the `setCompletionBlock:` method of `NSOperation`. The block you pass to this method should have no arguments and no return value.

Tips for Implementing Operation Objects

Although operation objects are fairly easy to implement, there are several things you should be aware of as you are writing your code. The following sections describe factors that you should take into account when writing the code for your operation objects.

Managing Memory in Operation Objects

The following sections describe key elements of good memory management in an operation object. For general information about memory management in Objective-C programs, see *Advanced Memory Management Programming Guide*.

Avoid Per-Thread Storage

Although most operations execute on a thread, in the case of nonconcurrent operations, that thread is usually provided by an operation queue. If an operation queue provides a thread for you, you should consider that thread to be owned by the queue and not to be touched by your operation. Specifically, you should never associate any data with a thread that you do not create yourself or manage. The threads managed by an operation queue come and go depending on the needs of the system and your application. Therefore, passing data between operations using per-thread storage is unreliable and likely to fail.

In the case of operation objects, there should be no reason for you to use per-thread storage in any case. When you initialize an operation object, you should provide the object with everything it needs to do its job. Therefore, the operation object itself provides the contextual storage you need. All incoming and outgoing data should be stored there until it can be integrated back into your application or is no longer required.

Keep References to Your Operation Object As Needed

Just because operation objects run asynchronously, you should not assume that you can create them and forget about them. They are still just objects and it is up to you to manage any references to them that your code needs. This is especially important if you need to retrieve result data from an operation after it is finished.

The reason you should always keep your own references to operations is that you may not get the chance to ask a queue for the object later. Queues make every effort to dispatch and execute operations as quickly as possible. In many cases, queues start executing operations almost immediately after they are added. By the time your own code goes back to the queue to get a reference to the operation, that operation could already be finished and removed from the queue.

Handling Errors and Exceptions

Because operations are essentially discrete entities inside your application, they are responsible for handling any errors or exceptions that arise. In OS X v10.6 and later, the default `start` method provided by the `NSOperation` class does not catch exceptions. (In OS X v10.5, the start method does catch and suppress exceptions.) Your own code should always catch and suppress exceptions directly. It should also check error codes and notify the appropriate parts of your application as needed. And if you replace the `start` method, you must similarly catch any exceptions in your custom implementation to prevent them from leaving the scope of the underlying thread.

Among the types of error situations you should be prepared to handle are the following:

- Check and handle UNIX `errno`-style error codes.
- Check explicit error codes returned by methods and functions.
- Catch exceptions thrown by your own code or by other system frameworks.
- Catch exceptions thrown by the `NSOperation` class itself, which throws exceptions in the following situations:
  - When the operation is not ready to execute but its `start` method is called
  - When the operation is executing or finished (possibly because it was canceled) and its `start` method is called again
  - When you try to add a completion block to an operation that is already executing or finished
  - When you try to retrieve the result of an `NSInvocationOperation` object that was canceled

If your custom code does encounter an exception or error, you should take whatever steps are needed to propagate that error to the rest of your application. The `NSOperation` class does not provide explicit methods for passing along error result codes or exceptions to other parts of your application. Therefore, if such information is important to your application, you must provide the necessary code.

Determining an Appropriate Scope for Operation Objects

Although it is possible to add an arbitrarily large number of operations to an operation queue, doing so is often impractical. Like any object, instances of the `NSOperation` class consume memory and have real costs associated with their execution. If each of your operation objects does only a small amount of work, and you create tens of thousands of them, you may find that you are spending more time dispatching operations than doing real work. And if your application is already memory-constrained, you might find that just having tens of thousands of operation objects in memory might degrade performance even further.

The key to using operations efficiently is to find an appropriate balance between the amount of work you need to do and to keep the computer busy. Try to make sure that your operations do a reasonable amount of work. For example, if your application creates 100 operation objects to perform the same task on 100 different values, consider creating 10 operation objects to process 10 values each instead.

You should also avoid adding large numbers of operations to a queue all at once, or avoid continuously adding operation objects to a queue faster than they can be processed. Rather than flood a queue with operation objects, create those objects in batches. As one batch finishes executing, use a completion block to tell your application to create a new batch. When you have a lot of work to do, you want to keep the queues filled with enough operations so that the computer stays busy, but you do not want to create so many operations at once that your application runs out of memory.

Of course, the number of operation objects you create, and the amount of work you perform in each, is variable and entirely dependent on your application. You should always use tools such as Instruments to help you find an appropriate balance between efficiency and speed. For an overview of Instruments and the other performance tools you can use to gather metrics for your code, see *Performance Overview*.

Executing Operations

Ultimately, your application needs to execute operations in order to do the associated work. In this section, you learn several ways to execute operations as well as how you can manipulate the execution of your operations at runtime.

Adding Operations to an Operation Queue

By far, the easiest way to execute operations is to use an operation queue, which is an instance of the `NSOperationQueue` class. Your application is responsible for creating and maintaining any operation queues it intends to use. An application can have any number of queues, but there are practical limits to how many operations may be executing at a given point in time. Operation queues work with the system to restrict the number of concurrent operations to a value that is appropriate for the available cores and system load. Therefore, creating additional queues does not mean that you can execute additional operations.

To create a queue, you allocate it in your application as you would any other object:

```
NSOperationQueue* aQueue = [[NSOperationQueue alloc] init];
```

To add operations to a queue, you use the `addOperation:` method. In OS X v10.6 and later, you can add groups of operations using the `addOperations:waitUntilFinished:` method, or you can add block objects directly to a queue (without a corresponding operation object) using the `addOperationWithBlock:` method. Each of these methods queues up an operation (or operations) and notifies the queue that it should begin processing them. In most cases, operations are executed shortly after being added to a queue, but the operation queue may delay execution of queued operations for any of several reasons. Specifically, execution may be delayed if queued operations are dependent on other operations that have not yet completed. Execution may also be delayed if the operation queue itself is suspended or is already executing its maximum number of concurrent operations. The following examples show the basic syntax for adding operations to a queue.

```
[aQueue addOperation:anOp]; // Add a single operation
[aQueue addOperations:anArrayOfOps waitUntilFinished:NO]; // Add multiple operations
[aQueue addOperationWithBlock:^{
   /* Do something. */
}];
```

Important:

You should make all necessary configuration and modifications to an operation object before adding it to a queue, because once added, the operation may be run at any time, which may be too late for a change to have the intended effect.

Although the `NSOperationQueue` class is designed for the concurrent execution of operations, it is possible to force a single queue to run only one operation at a time. The `setMaxConcurrentOperationCount:` method lets you configure the maximum number of concurrent operations for an operation queue object. Passing a value of 1 to this method causes the queue to execute only one operation at a time. Although only one operation at a time may execute, the order of execution is still based on other factors, such as the readiness of each operation and its assigned priority. Thus, a serialized operation queue does not offer quite the same behavior as a serial dispatch queue in Grand Central Dispatch does. If the execution order of your operation objects is important to you, you should use dependencies to establish that order before adding your operations to a queue. For information about configuring dependencies, see Configuring Interoperation Dependencies.

For information about using operation queues, see *NSOperationQueue Class Reference*. For more information about serial dispatch queues, see Creating Serial Dispatch Queues.

Executing Operations Manually

Although operation queues are the most convenient way to run operation objects, it is also possible to execute operations without a queue. If you choose to execute operations manually, however, there are some precautions you should take in your code. In particular, the operation must be ready to run and you must always start it using its `start` method.

An operation is not considered able to run until its `isReady` method returns `YES`. The `isReady` method is integrated into the dependency management system of the `NSOperation` class to provide the status of the operation's dependencies. Only when its dependencies are cleared is an operation free to begin executing.

When executing an operation manually, you should always use the `start` method to begin execution. You use this method, instead of `main` or some other method, because the `start` method performs several safety checks before it actually runs your custom code. In particular, the default `start` method generates the KVO notifications that operations require to process their dependencies correctly. This method also correctly avoids executing your operation if it has already been canceled and throws an exception if your operation is not actually ready to run.

If your application defines concurrent operation objects, you should also consider calling the `isConcurrent` method of operations prior to launching them. In cases where this method returns `NO`, your local code can decide whether to execute the operation synchronously in the current thread or create a separate thread first. However, implementing this kind of checking is entirely up to you.

Listing 2-8 shows a simple example of the kind of checks you should perform before executing operations manually. If the method returns `NO`, you could schedule a timer and call the method again later. You would then keep rescheduling the timer until the method returns `YES`, which could occur because the operation was canceled.

Listing 2-8 Executing an operation object manually

```
- (BOOL)performOperation:(NSOperation*)anOp
{
   BOOL        ranIt = NO;

   if ([anOp isReady] && ![anOp isCancelled])
   {
      if (![anOp isConcurrent])
         [anOp start];
      else
         [NSThread detachNewThreadSelector:@selector(start)
                   toTarget:anOp withObject:nil];
      ranIt = YES;
   }
   else if ([anOp isCancelled])
   {
      // If it was canceled before it was started,
      //  move the operation to the finished state.
      [self willChangeValueForKey:@"isFinished"];
      [self willChangeValueForKey:@"isExecuting"];
      executing = NO;
      finished = YES;
      [self didChangeValueForKey:@"isExecuting"];
      [self didChangeValueForKey:@"isFinished"];

      // Set ranIt to YES to prevent the operation from
      // being passed to this method again in the future.
      ranIt = YES;
   }
   return ranIt;
}
```

Canceling Operations

Once added to an operation queue, an operation object is effectively owned by the queue and cannot be removed. The only way to dequeue an operation is to cancel it. You can cancel a single individual operation object by calling its `cancel` method or you can cancel all of the operation objects in a queue by calling the `cancelAllOperations` method of the queue object.

You should cancel operations only when you are sure you no longer need them. Issuing a cancel command puts the operation object into the “canceled” state, which prevents it from ever being run. Because a canceled operation is still considered to be “finished”, objects that are dependent on it receive the appropriate KVO notifications to clear that dependency. Thus, it is more common to cancel all queued operations in response to some significant event, like the application quitting or the user specifically requesting the cancellation, rather than cancel operations selectively.

Waiting for Operations to Finish

For the best performance, you should design your operations to be as asynchronous as possible, leaving your application free to do additional work while the operation executes. If the code that creates an operation also processes the results of that operation, you can use the `waitUntilFinished` method of `NSOperation` to block that code until the operation finishes. In general, though, it is best to avoid calling this method if you can help it. Blocking the current thread may be a convenient solution, but it does introduce more serialization into your code and limits the overall amount of concurrency.

> **Important:** You should never wait for an operation from your application's main thread. You should only do so from a secondary thread or from another operation. Blocking your main thread prevents your application from responding to user events and could make your application appear unresponsive.

In addition to waiting for a single operation to finish, you can also wait on all of the operations in a queue by calling the `waitUntilAllOperationsAreFinished` method of `NSOperationQueue`. When waiting for an entire queue to finish, be aware that your application's other threads can still add operations to the queue, thus prolonging the wait.

Suspending and Resuming Queues

If you want to issue a temporary halt to the execution of operations, you can suspend the corresponding operation queue using the `setSuspended:` method. Suspending a queue does not cause already executing operations to pause in the middle of their tasks. It simply prevents new operations from being scheduled for execution. You might suspend a queue in response to a user request to pause any ongoing work, because the expectation is that the user might eventually want to resume that work.
