

---

# NSOperation Class Reference

Data Management: Event Handling



2009-04-20



Apple Inc.  
© 2009 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, Cocoa, Mac, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

**Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY**

**DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.**

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.**

**Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.**

# Contents

## **NSOperation Class Reference 7**

---

Overview	7
Operation Dependencies	8
KVO-Compliant Properties	8
Multicore Considerations	8
Concurrent Versus Non-Concurrent Operations	9
Subclassing Notes	9
Tasks	12
Initialization	12
Executing the Operation	12
Canceling Operations	13
Getting the Operation Status	13
Managing Dependencies	13
Prioritizing Operations in an Operation Queue	13
Managing the Execution Priority	13
Waiting for Completion	14
Instance Methods	14
addDependency:	14
cancel	14
completionBlock	15
dependencies	15
init	16
isCancelled	16
isConcurrent	17
isExecuting	17
isFinished	18
isReady	18
main	19
queuePriority	19
removeDependency:	20
setCompletionBlock:	20
setQueuePriority:	21
setThreadPriority:	21
start	22
threadPriority	23
waitUntilFinished	23
Constants	23
NSOperationQueuePriority	23
Operation Priorities	24

**Document Revision History 25**

---

**Index 27**

---

# Tables

## NSOperation Class Reference 7

---

Table 1	Key paths for operation object states	11
---------	---------------------------------------	----



# NSOperation Class Reference

---

<b>Inherits from</b>	NSObject
<b>Conforms to</b>	NSObject (NSObject)
<b>Framework</b>	/System/Library/Frameworks/Foundation.framework
<b>Availability</b>	Available in Mac OS X v10.5 and later.
<b>Companion guide</b>	Concurrency Programming Guide
<b>Declared in</b>	NSOperation.h
<b>Related sample code</b>	From A View to A Movie NSOperationSample ZipBrowser

## Overview

The `NSOperation` class is an abstract class you use to encapsulate the code and data associated with a single task. Because it is abstract, you do not use this class directly but instead subclass or use one of the system-defined subclasses (`NSInvocationOperation` or `NSBlockOperation`) to perform the actual task. Despite being abstract, the base implementation of `NSOperation` does include significant logic to coordinate the safe execution of your task. The presence of this built-in logic allows you to focus on the actual implementation of your task, rather than on the glue code needed to ensure it works correctly with other system objects.

An operation object is a single-shot object—that is, it executes its task once and cannot be used to execute it again. You typically execute operations by adding them to an operation queue (an instance of the `NSOperationQueue` class). An operation queue executes its operations either directly, by running them on secondary threads, or indirectly using the `libdispatch` library (also known as Grand Central Dispatch). For more information about how queues execute operations, see *NSOperationQueue Class Reference*.

If you do not want to use an operation queue, you can execute an operation yourself by calling its `start` method directly from your code. Executing operations manually does put more of a burden on your code, because starting an operation that is not in the ready state triggers an exception. The `isReady` method reports on the operation's readiness.

## Operation Dependencies

---

Dependencies are a convenient way to execute operations in a specific order. You can add and remove dependencies for an operation using the `addDependency:` and `removeDependency:` methods. By default, an operation object that has dependencies is not considered ready until all of its dependent operation objects have finished executing. Once the last dependent operation finishes, however, the operation object becomes ready and able to execute.

The dependencies supported by `NSOperation` make no distinction about whether a dependent operation finished successfully or unsuccessfully. (In other words, canceling an operation similarly marks it as finished.) It is up to you to determine whether an operation with dependencies should proceed in cases where its dependent operations were cancelled or did not complete their task successfully. This may require you to incorporate some additional error tracking capabilities into your operation objects.

## KVO-Compliant Properties

---

The `NSOperation` class is key-value coding (KVC) and key-value observing (KVO) compliant for several of its properties. As needed, you can observe these properties to control other parts of your application. The properties you can observe include the following:

- `isCancelled` - read-only property
- `isConcurrent` - read-only property
- `isExecuting` - read-only property
- `isFinished` - read-only property
- `isReady` - read-only property
- `dependencies` - read-only property
- `queuePriority` - readable and writable property
- `completionBlock` - readable and writable property

Although you can attach observers to these properties, you should not use Cocoa bindings to bind them to elements of your application's user interface. Code associated with your user interface typically must execute only in your application's main thread. Because an operation may execute in any thread, KVO notifications associated with that operation may similarly occur in any thread.

If you provide custom implementations for any of the preceding properties, your implementations must maintain KVC and KVO compliance. If you define additional properties for your `NSOperation` objects, it is recommended that you make those properties KVC and KVO compliant as well. For information on how to support key-value coding, see *Key-Value Coding Programming Guide*. For information on how to support key-value observing, see *Key-Value Observing Programming Guide*.

## Multicore Considerations

---

The `NSOperation` class is itself multicore aware. It is therefore safe to call the methods of an `NSOperation` object from multiple threads without creating additional locks to synchronize access to the object. This behavior is necessary because an operation typically runs in a separate thread from the one that created and is monitoring it.

When you subclass `NSOperation`, you must make sure that any overridden methods remain safe to call from multiple threads. If you implement custom methods in your subclass, such as custom data accessors, you must also make sure those methods are thread-safe. Thus, access to any data variables in the operation must be synchronized to prevent potential data corruption. For more information about synchronization, see *Threading Programming Guide*.

## Concurrent Versus Non-Concurrent Operations

---

If you plan on executing an operation object manually, instead of adding it to a queue, you can design your operation to execute in a concurrent or non-concurrent manner. Operation objects are non-concurrent by default. In a non-concurrent operation, the operation's task is performed synchronously—that is, the operation object does not create a separate thread on which to run the task. Thus, when you call the `start` method of a non-concurrent operation directly from your code, the operation executes immediately in the current thread. By the time the `start` method of such an object returns control to the caller, the task itself is complete.

In contrast to a non-concurrent operation, which runs synchronously, a concurrent operation runs asynchronously. In other words, when you call the `start` method of a concurrent operation, that method could return before the corresponding task is completed. This might happen because the operation object created a new thread to execute the task or because the operation called an asynchronous function. It does not actually matter if the operation is ongoing when control returns to the caller, only that it could be ongoing.

If you always plan to use queues to execute your operations, it is simpler to define them as non-concurrent. If you execute operations manually, though, you might want to define your operation objects as concurrent to ensure that they always execute asynchronously. Defining a concurrent operation requires more work, because you have to monitor the ongoing state of your task and report changes in that state using KVO notifications. But defining concurrent operations can be useful in cases where you want to ensure that a manually executed operation does not block the calling thread.

For information on how to define both concurrent and non-concurrent operations, see the subclassing notes.

**Note:** In Mac OS X v10.6, operation queues ignore the value returned by `isConcurrent` and always call the `start` method of your operation from a separate thread. In Mac OS X v10.5, however, operation queues create a thread only if `isConcurrent` returns `NO`. In general, if you are always using operations with an operation queue, there is no reason to make them concurrent.

## Subclassing Notes

---

The `NSOperation` class provides the basic logic to track the execution state of your operation but otherwise must be subclassed to do any real work. How you create your subclass depends on whether your operation is designed to execute concurrently or non-concurrently.

### Methods to Override

---

For non-concurrent operations, you typically override only one method:

- `main`

Into this method, you place the code needed to perform the given task. Of course, you should also define a custom initialization method to make it easier to create instances of your custom class. You might also want to define getter and setter methods to access the data from the operation. However, if you do define custom getter and setter methods, you must make sure those methods can be called safely from multiple threads.

If you are creating a concurrent operation, you need to override the following methods at a minimum:

- `start`
- `isConcurrent`
- `isExecuting`
- `isFinished`

In a concurrent operation, your `start` method is responsible for starting the operation in an asynchronous manner. Whether you spawn a thread or call an asynchronous function, you do it from this method. Upon starting the operation, your `start` method should also update the execution state of the operation as reported by the `isExecuting` method. You do this by sending out KVO notifications for the `isExecuting` key path, which lets interested clients know that the operation is now running. Your `isExecuting` method must also return the status in a thread-safe manner.

Upon completion or cancellation of its task, your concurrent operation object must generate KVO notifications for both the `isExecuting` and `isFinished` key paths to mark the final change of state for your operation. (In the case of cancellation, it is still important to update the `isFinished` key path, even if the operation did not completely finish its task. Queued operations must report that they are finished before they can be removed from a queue.) In addition to generating KVO notifications, your overrides of the `isExecuting` and `isFinished` methods should also continue to return accurate values based on the state of your operation.

For additional information and guidance on how to define concurrent operations, see *Concurrency Programming Guide*.

**Important:** At no time in your `start` method should you ever call `super`. When you define a concurrent operation, you take it upon yourself to provide the same behavior that the default `start` method provides, which includes starting the task and generating the appropriate KVO notifications. Your `start` method should also check to see if the operation itself was cancelled before actually starting the task. For more information about cancellation semantics, see [“Responding to the Cancel Command”](#) (page 11).

Even for concurrent operations, there should be little need to override methods other than those described above. However, if you customize the dependency features of operations, you might have to override additional methods and provide additional KVO notifications. In the case of dependencies, this would likely only require providing notifications for the `isReady` key path. Because the `dependencies` property is used to manage the list of dependent operations, changes to it are already handled by the default `NSOperation` class.

## Maintaining Operation Object States

---

Operation objects maintain state information internally to determine when it is safe to execute and also to notify external clients of the progression through the operation’s life cycle. Your custom subclasses must maintain this state information to ensure the correct execution of operations in your code. Table 1 lists the key paths associated with an operation’s states and how you should manage that key path in any custom subclasses.

**Table 1** Key paths for operation object states

Key Path	Description
<code>isReady</code>	<p>The <code>isReady</code> key path lets clients know when an operation is ready to execute. The <code>isReady</code> method returns <code>YES</code> to indicate that the operation is ready to execute now or <code>NO</code> if there are still unfinished operations on which it is dependent.</p> <p>In most cases, you do not have to manage the state of this key path yourself. If the readiness of your operations is determined by factors other than dependent operations, however—such as by some external condition in your program—you can provide your own implementation of the <code>isReady</code> method and track your operation’s readiness yourself. It is often simpler though just to create operation objects only when your external state allows it.</p> <p>In Mac OS X v10.6 and later, if you cancel an operation while it is waiting on the completion of one or more dependent operations, those dependencies are thereafter ignored and the value of this property is updated to reflect that it is now ready to run. This behavior gives an operation queue the chance to flush cancelled operations out of its queue more quickly.</p>
<code>isExecuting</code>	<p>The <code>isExecuting</code> key path lets clients know whether the operation is actively working on its assigned task. The <code>isExecuting</code> method must return <code>YES</code> if it is working on its task or <code>NO</code> if it is not.</p> <p>If you replace the <code>start</code> method of your operation object, you must also replace the <code>isExecuting</code> method and generate KVO notifications when the execution state of your operation changes.</p>
<code>isFinished</code>	<p>The <code>isFinished</code> key path lets clients know that an operation finished its task successfully or was cancelled and is exiting. An operation object does not clear a dependency until the value at the <code>isFinished</code> key path changes to <code>YES</code>. Similarly, an operation queue does not dequeue an operation until the <code>isFinished</code> method returns <code>YES</code>. Thus, marking operations as finished is critical to keeping queues from backing up with in-progress or cancelled operations.</p> <p>If you replace the <code>start</code> method or your operation object, you must also replace the <code>isFinished</code> method and generate KVO notifications when the operation finishes executing or is cancelled.</p>
<code>isCancelled</code>	<p>The <code>isCancelled</code> key path lets clients know that the cancellation of an operation was requested. Support for cancellation is voluntary but encouraged and your own code should not have to send KVO notifications for this key path. The handling of cancellation notices in an operation is described in more detail in <a href="#">“Responding to the Cancel Command”</a> (page 11).</p>

## Responding to the Cancel Command

---

Once you add an operation to a queue, the operation is out of your hands. The queue takes over and handles the scheduling of that task. However, if you decide later that you do not want to execute the operation after all—because the user pressed a cancel button in a progress panel or quit the application, for example—you can cancel the operation to prevent it from consuming CPU time needlessly. You do this by calling the `cancel` method of the operation object itself or by calling the `cancelAllOperations` method of the `NSOperationQueue` class.

Cancelling an operation does not immediately force it to stop what it is doing. Although respecting the value returned by the `isCancelled` is expected of all operations, your code must explicitly check the value returned by this method and abort as needed. The default implementation of `NSOperation` does include checks for cancellation. For example, if you cancel an operation before its `start` method is called, the `start` method exits without starting the task.

**Note:** In Mac OS X v10.6, the behavior of the `cancel` method varies depending on whether the operation is currently in an operation queue. For unqueued operations, this method marks the operation as finished immediately, generating the appropriate KVO notifications. For queued operations, it simply marks the operation as ready to execute and lets the queue call its `start` method, which subsequently exits and results in the clearing of the operation from the queue.

You should always support cancellation semantics in any custom code you write. In particular, your main task code should periodically check the value of the `isCancelled` method. If the method ever returns `YES`, your operation object should clean up and exit as quickly as possible. If you implement a custom `start` method, that method should include early checks for cancellation and behave appropriately. Your custom `start` method must be prepared to handle this type of early cancellation.

In addition to simply exiting when an operation is cancelled, it is also important that you move a cancelled operation to the appropriate final state. Specifically, if you manage the values for the `isFinished` and `isExecuting` properties yourself (perhaps because you are implementing a concurrent operation), you must update those variables accordingly. Specifically, you must change the value returned by `isFinished` to `YES` and the value returned by `isExecuting` to `NO`. You must make these changes even if the operation was cancelled before it started executing.

## Tasks

### Initialization

- [init](#) (page 16)  
Returns an initialized `NSOperation` object.

### Executing the Operation

- [start](#) (page 22)  
Begins the execution of the operation.
- [main](#) (page 19)  
Performs the receiver's non-concurrent task.
- [completionBlock](#) (page 15)  
Returns the block to execute when the operation's main task is complete.
- [setCompletionBlock:](#) (page 20)  
Sets the block to execute when the operation has finished executing.

## Canceling Operations

- `cancel` (page 14)  
Advises the operation object that it should stop executing its task.

## Getting the Operation Status

- `isCancelled` (page 16)  
Returns a Boolean value indicating whether the operation has been cancelled.
- `isExecuting` (page 17)  
Returns a Boolean value indicating whether the operation is currently executing.
- `isFinished` (page 18)  
Returns a Boolean value indicating whether the operation is done executing.
- `isConcurrent` (page 17)  
Returns a Boolean value indicating whether the operation runs asynchronously.
- `isReady` (page 18)  
Returns a Boolean value indicating whether the receiver's operation can be performed now.

## Managing Dependencies

- `addDependency:` (page 14)  
Makes the receiver dependent on the completion of the specified operation.
- `removeDependency:` (page 20)  
Removes the receiver's dependence on the specified operation.
- `dependencies` (page 15)  
Returns a new array object containing the operations on which the receiver is dependent.

## Prioritizing Operations in an Operation Queue

- `queuePriority` (page 19)  
Returns the priority of the operation in an operation queue.
- `setQueuePriority:` (page 21)  
Sets the priority of the operation when used in an operation queue.

## Managing the Execution Priority

- `threadPriority` (page 23)  
Returns the thread priority to use when executing the operation.
- `setThreadPriority:` (page 21)  
Sets the thread priority to use when executing the operation.

## Waiting for Completion

- [waitUntilFinished](#) (page 23)

Blocks execution of the current thread until the receiver finishes.

## Instance Methods

### addDependency:

Makes the receiver dependent on the completion of the specified operation.

- (void)addDependency:(NSOperation \*)*operation*

#### Parameters

*operation*

The operation on which the receiver should depend. The same dependency should not be added more than once to the receiver, and the results of doing so are undefined.

#### Discussion

The receiver is not considered ready to execute until all of its dependent operations have finished executing. If the receiver is already executing its task, adding dependencies has no practical effect. This method may change the `isReady` and `dependencies` properties of the receiver.

It is a programmer error to create any circular dependencies among a set of operations. Doing so can cause a deadlock among the operations and may freeze your program.

#### Availability

Available in Mac OS X v10.5 and later.

#### See Also

- [removeDependency:](#) (page 20)
- [dependencies](#) (page 15)

#### Declared In

NSOperation.h

### cancel

Advises the operation object that it should stop executing its task.

- (void)cancel

#### Discussion

This method does not force your operation code to stop. Instead, it updates the object's internal flags to reflect the change in state. If the operation has already finished executing, this method has no effect. Canceling an operation that is currently in an operation queue, but not yet executing, makes it possible to remove the operation from the queue sooner than usual.

In Mac OS X v10.6 and later, if an operation is in a queue but waiting on unfinished dependent operations, those operations are subsequently ignored. Because it is already cancelled, this behavior allows the operation queue to call the operation's `start` method sooner and clear the object out of the queue. If you cancel an operation that is not in a queue, this method immediately marks the object as finished. In each case, marking the object as ready or finished results in the generation of the appropriate KVO notifications.

In versions of Mac OS X prior to 10.6, an operation object remains in the queue until all of its dependencies are removed through the normal processes. Thus, the operation must wait until all of its dependent operations finish executing or are themselves cancelled and have their `start` method called.

For more information on what you must do in your operation objects to support cancellation, see [“Responding to the Cancel Command”](#) (page 11).

#### Availability

Available in Mac OS X v10.5 and later.

#### See Also

- [isCancelled](#) (page 16)

#### Declared In

`NSOperation.h`

## completionBlock

Returns the block to execute when the operation's main task is complete.

```
- (void (^)(void))completionBlock
```

#### Return Value

The block to execute after the operation's main task is completed. This block takes no parameters and has no return value.

#### Discussion

Operation objects monitor the `isFinished` key path and execute this block when the value at that key path changes to YES. As a result, this block is called regardless of whether the operation completed successfully or was cancelled.

#### Availability

Available in Mac OS X v10.6 and later.

#### See Also

- [setCompletionBlock:](#) (page 20)

#### Declared In

`NSOperation.h`

## dependencies

Returns a new array object containing the operations on which the receiver is dependent.

```
- (NSArray *)dependencies
```

**Return Value**

A new array object containing the `NSOperation` objects.

**Discussion**

The receiver is not considered ready to execute until all of its dependent operations finish executing.

Operations are not removed from this dependency list as they finish executing. You can therefore use this list to track all dependent operations, including those that have already finished executing. The only way to remove an operation from this list is to use the `removeDependency:` method.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- [addDependency:](#) (page 14)
- [removeDependency:](#) (page 20)

**Declared In**

`NSOperation.h`

**init**

Returns an initialized `NSOperation` object.

- (id)init

**Return Value**

The initialized `NSOperation` object.

**Discussion**

Your custom subclasses must call this method. The default implementation initializes the object's instance variables and prepares the it for use.

**Availability**

Available in Mac OS X v10.5 and later.

**Related Sample Code**

From A View to A Movie

`NSOperationSample`

`ZipBrowser`

**Declared In**

`NSOperation.h`

**isCancelled**

Returns a Boolean value indicating whether the operation has been cancelled.

- (BOOL)isCancelled

**Return Value**

YES if the operation was explicitly cancelled by an invocation of the receiver's `cancel` method; otherwise, NO. This method may return YES even if the operation is currently executing.

**Discussion**

Canceling an operation does not actively stop the receiver’s code from executing. An operation object is responsible for calling this method periodically and stopping itself if the method returns YES.

You should always call this method before doing any work towards accomplishing the operation’s task, which typically means calling it at the beginning of your custom `main` method. It is possible for an operation to be cancelled before it begins executing or at any time while it is executing. Therefore, calling this method at the beginning of your `main` method (and periodically throughout that method) lets you exit as quickly as possible when an operation is cancelled.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- [cancel](#) (page 14)

**Related Sample Code**

From A View to A Movie

NSOperationSample

**Declared In**

NSOperation.h

**isConcurrent**

Returns a Boolean value indicating whether the operation runs asynchronously.

- (BOOL)isConcurrent

**Return Value**

YES if the operation runs asynchronously with respect to the current thread or NO if the operation runs synchronously on whatever thread started it. This method returns NO by default.

**Discussion**

If you are implementing a concurrent operation, you must override this method and return YES from your implementation. For more information about the differences between concurrent and non-concurrent operations, see “[Concurrent Versus Non-Concurrent Operations](#)” (page 9).

In Mac OS X v10.6 and later, operation queues ignore the value returned by this method and always start operations on a separate thread.

**Availability**

Available in Mac OS X v10.5 and later.

**Declared In**

NSOperation.h

**isExecuting**

Returns a Boolean value indicating whether the operation is currently executing.

- (BOOL)isExecuting

**Return Value**

YES if the operation is executing; otherwise, NO if the operation has not been started or is already finished.

**Discussion**

If you are implementing a concurrent operation, you should override this method to return the execution state of your operation. If you do override it, be sure to generate KVO notifications for the `isExecuting` key path whenever the execution state of your operation object changes. For more information about manually generating KVO notifications, see *Key-Value Observing Programming Guide*.

**Availability**

Available in Mac OS X v10.5 and later.

**Declared In**

`NSOperation.h`

**isFinished**

Returns a Boolean value indicating whether the operation is done executing.

- (BOOL)isFinished

**Return Value**

YES if the operation is no longer executing; otherwise, NO.

**Discussion**

If you are implementing a concurrent operation, you should override this method and return a Boolean to indicate whether your operation is currently finished. If you do override it, be sure to generate appropriate KVO notifications for the `isFinished` key path when the completion state of your operation object changes. For more information about manually generating KVO notifications, see *Key-Value Observing Programming Guide*.

**Availability**

Available in Mac OS X v10.5 and later.

**Declared In**

`NSOperation.h`

**isReady**

Returns a Boolean value indicating whether the receiver's operation can be performed now.

- (BOOL)isReady

**Return Value**

YES if the operation can be performed now; otherwise, NO.

**Discussion**

Operations may not be ready due to dependencies on other operations or because of external conditions that might prevent needed data from being ready. The `NSOperation` class manages dependencies on other operations and reports the readiness of the receiver based on those dependencies.

If you want to use custom conditions to determine the readiness of your operation object, you can override this method and return a value that accurately reflects the readiness of the receiver. If you do so, your custom implementation should invoke `super` and incorporate its return value into the readiness state of the object. Your custom implementation must also generate appropriate KVO notifications for the `isReady` key path.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- [dependencies](#) (page 15)

**Declared In**

`NSOperation.h`

**main**

Performs the receiver's non-concurrent task.

- (void)main

**Discussion**

The default implementation of this method does nothing. You should override this method to perform the desired task. In your implementation, do not invoke `super`.

If you are implementing a concurrent operation, you are not required to override this method but may do so if you plan to call it from your custom `start` method.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- [start](#) (page 22)

**Declared In**

`NSOperation.h`

**queuePriority**

Returns the priority of the operation in an operation queue.

- (NSOperationQueuePriority)queuePriority

**Return Value**

The relative priority of the operation. The returned value always corresponds to one of the predefined constants. (For a list of valid values, see ["Operation Priorities"](#) (page 24).) If no priority is explicitly set, this method returns `NSOperationQueuePriorityNormal`.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- [setQueuePriority:](#) (page 21)

**Declared In**

NSOperation.h

**removeDependency:**

Removes the receiver's dependence on the specified operation.

```
- (void)removeDependency:(NSOperation *)operation
```

**Parameters***operation*

The dependent operation to be removed from the receiver.

**Discussion**

This method may change the `isReady` and `dependencies` properties of the receiver.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- [addDependency:](#) (page 14)
- [dependencies](#) (page 15)

**Declared In**

NSOperation.h

**setCompletionBlock:**

Sets the block to execute when the operation has finished executing.

```
- (void)setCompletionBlock:(void (^)(void))block
```

**Parameters***block*

The block to be executed when the operation finishes. This method creates a copy of the specified block. The block itself should take no parameters and have no return value.

**Discussion**

The exact execution context for your completion block is not guaranteed but is typically a secondary thread. Therefore, you should not use this block to do any work that requires a very specific execution context. Instead, you should shunt that work to your application's main thread or to the specific thread that is capable of doing it. For example, if you have a custom thread for coordinating the completion of the operation, you could use the completion block to ping that thread.

A finished operation may finish either because it was cancelled or because it successfully completed its task. You should take that fact into account when writing your block code. Similarly, you should not make any assumptions about the successful completion of dependent operations, which may themselves have been cancelled.

**Availability**

Available in Mac OS X v10.6 and later.

**Declared In**

NSOperation.h

## setQueuePriority:

Sets the priority of the operation when used in an operation queue.

```
- (void)setQueuePriority:(NSOperationQueuePriority)priority
```

### Parameters

*priority*

The relative priority of the operation. For a list of valid values, see [“Operation Priorities”](#) (page 24).

### Discussion

You should use priority values only as needed to classify the relative priority of non-dependent operations. Priority values should not be used to implement dependency management among different operation objects. If you need to establish dependencies between operations, use the `addDependency:` method instead.

If you attempt to specify a priority value that does not match one of the defined constants, this method automatically adjusts the value you specify towards the `NSOperationQueuePriorityNormal` priority, stopping at the first valid constant value. For example, if you specified the value `-10`, this method would adjust that value to match the `NSOperationQueuePriorityVeryLow` constant. Similarly, if you specified `+10`, this method would adjust the value to match the `NSOperationQueuePriorityVeryHigh` constant.

### Availability

Available in Mac OS X v10.5 and later.

### See Also

- [queuePriority](#) (page 19)
- [addDependency:](#) (page 14)

### Related Sample Code

NSOperationSample

### Declared In

NSOperation.h

## setThreadPriority:

Sets the thread priority to use when executing the operation.

```
- (void)setThreadPriority:(double)priority
```

### Parameters

*priority*

The new thread priority, specified as a floating-point number in the range 0.0 to 1.0, where 1.0 is the highest priority.

### Discussion

The value you specify is mapped to the operating system’s priority values. The specified thread priority is applied to the thread only while the operation’s `main` method is executing. It is not applied while the operation’s completion block is executing. For a concurrent operation in which you create your own thread, you must set the thread priority yourself in your custom `start` method and reset the original priority when the operation is finished.

### Availability

Available in Mac OS X v10.6 and later.

**See Also**

+ [setThreadPriority:](#) (NSThread)

**Declared In**

NSOperation.h

**start**

Begins the execution of the operation.

```
- (void)start
```

**Discussion**

The default implementation of this method updates the execution state of the operation and calls the receiver's main method. This method also performs several checks to ensure that the operation can actually run. For example, if the receiver was cancelled or is already finished, this method simply returns without calling main. (In Mac OS X v10.5, this method throws an exception if the operation is already finished.) If the operation is currently executing or is not ready to execute, this method throws an `NSInvalidArgumentException` exception. In Mac OS X v10.5, this method catches and ignores any exceptions thrown by your main method automatically. In Mac OS X v10.6 and later, exceptions are allowed to propagate beyond this method. You should never allow exceptions to propagate out of your main method.

**Note:** An operation is not considered ready to execute if it is still dependent on other operations that have not yet finished.

If you are implementing a concurrent operation, you must override this method and use it to initiate your operation. Your custom implementation must not call `super` at any time. In addition to configuring the execution environment for your task, your implementation of this method must also track the state of the operation and provide appropriate state transitions. When the operation executes and subsequently finishes its work, it should generate KVO notifications for the `isExecuting` and `isFinished` key paths respectively. For more information about manually generating KVO notifications, see *Key-Value Observing Programming Guide*.

You can call this method explicitly if you want to execute your operations manually. However, it is a programmer error to call this method on an operation object that is already in an operation queue or to queue the operation after calling this method. Once you add an operation object to a queue, the queue assumes all responsibility for it.

**Availability**

Available in Mac OS X v10.5 and later.

**See Also**

- [main](#) (page 19)
- [isReady](#) (page 18)
- [dependencies](#) (page 15)

**Declared In**

NSOperation.h

## threadPriority

Returns the thread priority to use when executing the operation.

```
- (double)threadPriority
```

### Return Value

A floating-point number in the range 0.0 to 1.0, where 1.0 is the highest priority. The default thread priority is 0.5.

### Availability

Available in Mac OS X v10.6 and later.

### See Also

```
+ threadPriority
```

### Declared In

```
NSOperation.h
```

## waitUntilFinished

Blocks execution of the current thread until the receiver finishes.

```
- (void)waitUntilFinished
```

### Discussion

The receiver should never call this method on itself and should avoid calling it on any operations submitted to the same operation queue as itself. Doing so can cause the operation to deadlock. It is generally safe to call this method on an operation that is in a different operation queue, although it is still possible to create deadlocks if each operation waits on the other.

A typical use for this method would be to call it from the code that created the operation in the first place. After submitting the operation to a queue, you would call this method to wait until that operation finished executing.

### Availability

Available in Mac OS X v10.6 and later.

### Declared In

```
NSOperation.h
```

## Constants

### NSOperationQueuePriority

Describes the priority of an operation relative to other operations in an operation queue.

```
typedef NSInteger NSOperationQueuePriority;
```

### Discussion

For a list of related constants, see [“Operation Priorities”](#) (page 24).

**Availability**

Available in Mac OS X v10.5 and later.

**Declared In**

NSOperation.h

**Operation Priorities**

These constants let you prioritize the order in which operations execute.

```
enum {
    NSOperationQueuePriorityVeryLow = -8,
    NSOperationQueuePriorityLow = -4,
    NSOperationQueuePriorityNormal = 0,
    NSOperationQueuePriorityHigh = 4,
    NSOperationQueuePriorityVeryHigh = 8
};
```

**Constants**

NSOperationQueuePriorityVeryLow

Operations receive very low priority for execution.

Available in Mac OS X v10.5 and later.

Declared in NSOperation.h.

NSOperationQueuePriorityLow

Operations receive low priority for execution.

Available in Mac OS X v10.5 and later.

Declared in NSOperation.h.

NSOperationQueuePriorityNormal

Operations receive the normal priority for execution.

Available in Mac OS X v10.5 and later.

Declared in NSOperation.h.

NSOperationQueuePriorityHigh

Operations receive high priority for execution.

Available in Mac OS X v10.5 and later.

Declared in NSOperation.h.

NSOperationQueuePriorityVeryHigh

Operations receive very high priority for execution.

Available in Mac OS X v10.5 and later.

Declared in NSOperation.h.

**Discussion**

You can use these constants to specify the relative ordering of operations that are waiting to be started in an operation queue. You should always use these constants (and not the defined value) for determining priority.

**Declared In**

NSOperation.h

# Document Revision History

---

This table describes the changes to *NSOperation Class Reference*.

Date	Notes
2009-04-20	Updated for Mac OS X v10.6.
2008-11-19	Updated the guidelines related to KVO compliance.
2008-10-15	Clarified cancellation semantics for concurrent operations.
2007-04-30	New document describing methods for managing encapsulated tasks.

**REVISION HISTORY**

Document Revision History

# Index

---

## A

---

`addDependency`: [instance method 14](#)

## C

---

`cancel` [instance method 14](#)  
`completionBlock` [instance method 15](#)

## D

---

`dependencies` [instance method 15](#)

## I

---

`init` [instance method 16](#)  
`isCancelled` [instance method 16](#)  
`isConcurrent` [instance method 17](#)  
`isExecuting` [instance method 17](#)  
`isFinished` [instance method 18](#)  
`isReady` [instance method 18](#)

## M

---

`main` [instance method 19](#)

## N

---

`NSOperationQueuePriority` [data type 23](#)  
`NSOperationQueuePriorityHigh` [constant 24](#)  
`NSOperationQueuePriorityLow` [constant 24](#)  
`NSOperationQueuePriorityNormal` [constant 24](#)  
`NSOperationQueuePriorityVeryHigh` [constant 24](#)

`NSOperationQueuePriorityVeryLow` [constant 24](#)

## O

---

[Operation Priorities 24](#)

## Q

---

`queuePriority` [instance method 19](#)

## R

---

`removeDependency`: [instance method 20](#)

## S

---

`setCompletionBlock`: [instance method 20](#)  
`setQueuePriority`: [instance method 21](#)  
`setThreadPriority`: [instance method 21](#)  
`start` [instance method 22](#)

## T

---

`threadPriority` [instance method 23](#)

## W

---

`waitUntilFinished` [instance method 23](#)