Class

Operation

An abstract class that represents the code and data associated with a single task.

iOS 2.0+ | iPadOS 2.0+ | Mac Catalyst 13.1+ | macOS 10.5+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
class Operation
```

Overview

Because the `Operation` class is an abstract class, you do not use it directly but instead subclass or use one of the system-defined subclasses (`NSInvocationOperation` or `BlockOperation`) to perform the actual task. Despite being abstract, the base implementation of `Operation` 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 `OperationQueue` 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 `OperationQueue`.

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` property 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. To observe the properties, use the following key paths:

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

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.

Asynchronous Versus Synchronous 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 synchronous or asynchronous manner. Operation objects are synchronous by default. In a synchronous operation, the operation object does not create a separate thread on which to run its task. When you call the `start()` method of a synchronous 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.

When you call the `start()` method of an asynchronous operation, that method may return before the corresponding task is completed. An asynchronous operation object is responsible for scheduling its task on a separate thread. The operation could do that by starting a new thread directly, by calling an asynchronous method, or by submitting a block to a dispatch queue for execution. 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 synchronous. If you execute operations manually, though, you might want to define your operation objects as asynchronous. Defining an asynchronous 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 asynchronous operations is useful in cases where you want to ensure that a manually executed operation does not block the calling thread.

When you add an operation to an operation queue, the queue ignores the value of the `isAsynchronous` property and always calls the `start()` method from a separate thread. Therefore, if you always run operations by adding them to an operation queue, there is no reason to make them asynchronous.

For information on how to define both synchronous and asynchronous operations, see the subclassing notes.

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 and properties at a minimum:

- `start()`
- `isAsynchronous`
- `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` property. 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` property must also provide 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` properties should also continue to report 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.

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 contains 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 maintains this state information to ensure the correct execution of operations in your code. The key paths associated with an operation’s states are:

`isReady`

The `isReady` key path lets clients know when an operation is ready to execute. The `isReady` property contains the value `true` when the operation is ready to execute now or `false` if there are still unfinished operations on which it is dependent.

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 `isReady` property and track your operation’s readiness yourself. It is often simpler though just to create operation objects only when your external state allows it.

In macOS 10.6 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.

`isExecuting`

The `isExecuting` key path lets clients know whether the operation is actively working on its assigned task. The `isExecuting` property must report the value `true` if the operation is working on its task or `false` if it is not.

If you replace the `start()` method of your operation object, you must also replace the `isExecuting` property and generate KVO notifications when the execution state of your operation changes.

`isFinished`

> The `isFinished` 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 `isFinished` key path changes to `true`. Similarly, an operation queue does not dequeue an operation until the `isFinished` property contains the value `true`. Thus, marking operations as finished is critical to keeping queues from backing up with in-progress or cancelled operations.

If you replace the `start()` method or your operation object, you must also replace the `isFinished` property and generate KVO notifications when the operation finishes executing or is cancelled.

`isCancelled`

> The `isCancelled` 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 Responding to the Cancel Command.

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 `OperationQueue` class.

Canceling an operation does not immediately force it to stop what it is doing. Although respecting the value in the `isCancelled` property is expected of all operations, your code must explicitly check the value in this property and abort as needed. The default implementation of `NSOperation` includes 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 macOS 10.6 and later, if you call the `cancel()` method on an operation that is in an operation queue and has unfinished dependent operations, those dependent operations are subsequently ignored. Because the operation is already cancelled, this behavior allows the queue to call the operation's `start()` method to remove the operation from the queue without calling its `main()` method. If you call the `cancel()` method on an operation that is not in a queue, the operation is immediately marked as being cancelled. In each case, marking the operation as ready or finished results in the generation of the appropriate KVO notifications.

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` property. If the property reports the value `true`, 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 properties accordingly. Specifically, you must change the value returned by `isFinished` to `true` and the value returned by `isExecuting` to `false`. You must make these changes even if the operation was cancelled before it started executing.

Topics

Executing the Operation

`func start()`

Begins the execution of the operation.

`func main()`

Performs the receiver's non-concurrent task.

`var completionBlock: (() -> Void)?`

The block to execute after the operation's main task is completed.

Canceling Operations

`func cancel()`

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

Getting the Operation Status

`var isCancelled: Bool`

A Boolean value indicating whether the operation has been cancelled

`var isExecuting: Bool`

A Boolean value indicating whether the operation is currently executing.

`var isFinished: Bool`

A Boolean value indicating whether the operation has finished executing its task.

`var isConcurrent: Bool`

A Boolean value indicating whether the operation executes its task asynchronously.

`var isAsynchronous: Bool`

A Boolean value indicating whether the operation executes its task asynchronously.

`var isReady: Bool`

A Boolean value indicating whether the operation can be performed now.

`var name: String?`

The name of the operation.

Managing Dependencies

`func addDependency(Operation)`

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

`func removeDependency(Operation)`

Removes the receiver's dependence on the specified operation.

`var dependencies: [Operation]`

An array of the operation objects that must finish executing before the current object can begin executing.

Configuring the Execution Priority

`var qualityOfService: QualityOfService`

The relative amount of importance for granting system resources to the operation.

~~`var threadPriority: Double`~~

The thread priority to use when executing the operation

Deprecated

`var queuePriority: Operation.QueuePriority`

The execution priority of the operation in an operation queue.

Waiting on an Operation Object

`func waitUntilFinished()`

Blocks execution of the current thread until the operation object finishes its task.

Constants

`enum QueuePriority`

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

`enum QualityOfService`

Constants that indicate the nature and importance of work to the system.

Relationships

Inherits From

`NSObject`

Inherited By

`BlockOperation`

Conforms To

`CVarArg`
`CustomDebugStringConvertible`

```
CustomStringConvertible
Equatable
Hashable
NSObjectProtocol
Sendable
SendableMetatype
```

See Also

Operations

`class OperationQueue`

A queue that regulates the execution of operations.

`class BlockOperation`

An operation that manages the concurrent execution of one or more blocks.