# Notification Programming Topics for Cocoa

**Cocoa > Events & Other Input**

**2007-05-03**

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



Simultaneously published in the United States and Canada.

# Contents

# Introduction to Notification Programming Topics

This document describes how to use the architecture supplied by Foundation to pass around information about the occurrence of events.

You should read this document to learn about notifications, notification centers, and notification queues.

## Organization of This Document

This document contains the following articles:

- "Notifications" (page 7) describes what a notification is.
- "Notification Centers" (page 9) describes notification centers.
- "Notification Queues" (page 11) describes notification queues.
- "Registering for a Notification" (page 15) describes how to register for a notification.
- "Posting a Notification" (page 19) describes how to post a notification.
- "Delivering Notifications To Particular Threads" (page 21) shows an example of how to handle notifications that should be processed on a particular thread.

# Notifications

A notification encapsulates information about an event, such as a window gaining focus or a network connection closing. Objects that need to know about an event (for example, a file that needs to know when its window is about to be closed) register with the notification center that it wants to be notified when that event happens. When the event does happen, a notification is posted to the notification center, which immediately broadcasts the notification to all registered objects. Optionally, a notification is queued in a notification queue, which posts notifications to a notification center after it delays specified notifications and coalesces notifications that are similar according to some specified criteria you specify.

**Note:** Frameworks such as Foundation and Application Kit make extensive use of notifications to allow objects to react to events they are interested in. The notifications sent by each class are described in the class's reference documentation, under the "Notifications" section.

## Notifications and Their Rationale

The standard way to pass information between objects is message passing—one object invokes the method of another object. However, message passing requires that the object sending the message know who the receiver is and what messages it responds to. At times, this tight coupling of two objects is undesirable—most notably because it would join together two otherwise independent subsystems. For these cases, a broadcast model is introduced: An object posts a notification, which is dispatched to the appropriate observers through an `NSNotificationCenter` object, or simply notification center.

An `NSNotification` object (referred to as a notification) contains a name, an object, and an optional dictionary. The name is a tag identifying the notification. The object is any object that the poster of the notification wants to send to observers of that notification (typically, it is the object that posted the notification). The dictionary stores other related objects if any.

Any object may post a notification. Other objects can register themselves with the notification center as observers to receive notifications when they are posted. The notification center takes care of broadcasting notifications to the registered observers, if any. The object posting the notification, the object included in the notification, and the observer of the notification may all be different objects or the same object. Objects that post notifications need not know anything about the observers. On the other hand, observers need to know at least the notification name and keys to the dictionary if provided.

## Notification and Delegation

Using the notification system is similar to using delegates, but it has these advantages:

- Any number of objects may receive the notification, not just the delegate object. This precludes returning a value.

- An object may receive any message you like from the notification center, not just the predefined delegate methods.
- The object posting the notification does not even have to know the observer exists.

# Notification Centers

A notification center manages the sending and receiving of notifications. It notifies all observers of notifications meeting specific criteria. The notification information is encapsulated in `NSNotification` objects. Client objects register themselves with the notification center as observers of specific notifications posted by other objects. When an event occurs, an object posts an appropriate notification to the notification center. (See "Posting a Notification" (page 19) for more on posting notifications.) The notification center dispatches a message to each registered observer, passing the notification as the sole argument. It is possible for the posting object and the observing object to be the same.

Cocoa includes two types of notification centers:

- The `NSNotificationCenter` class manages notifications within a single task.
- The `NSDistributedNotificationCenter` class manages notifications across multiple tasks on a single computer.

## NSNotificationCenter

Each task has a default notification center that you access with the `NSNotificationCenter +defaultCenter` method. This notification center handles notifications within a single task. For communication between tasks on the same machine, use a distributed notification center (see "NSDistributedNotificationCenter" (page 9)).

A notification center delivers notifications to observers synchronously. In other words, when posting a notification, control does not return to the poster until all observers have received and processed the notification. To send notifications asynchronously use a notification queue, which is described in "Notification Queues" (page 11).

In a multithreaded application, notifications are always delivered in the thread in which the notification was posted, which may not be the same thread in which an observer registered itself.

## NSDistributedNotificationCenter

Each task has a default distributed notification center that you access with the `NSDistributedNotificationCenter +defaultCenter` method. This distributed notification center handles notifications that can be sent between tasks on a single machine. For communication between tasks on different machines, use distributed objects (see Distributed Objects).

Posting a distributed notification is an expensive operation. The notification gets sent to a systemwide server that then distributes it to all the tasks that have objects registered for distributed notifications. The latency between posting the notification and the notification's arrival in another task is unbounded. In fact, if too many notifications are being posted and the server's queue fills up, notifications can be dropped.

Distributed notifications are delivered via a task's run loop. A task must be running a run loop in one of the "common" modes, such as `NSDefaultRunLoopMode`, to receive a distributed notification. If the receiving task is multithreaded, do not depend on the notification arriving on the main thread. The notification is usually delivered to the main thread's run loop, but other threads could also receive the notification.

Whereas a regular notification center allows any object to be observed, a distributed notification center is restricted to observing a string object. Because the posting object and the observer may be in different tasks, notifications cannot contain pointers to arbitrary objects. Therefore, a distributed notification center requires notifications to use a string as the notification object. Notification matching is done based on this string, rather than an object pointer.

# Notification Queues

`NSNotificationQueue` objects (or simply, notification queues) act as buffers for notification centers (instances of `NSNotificationCenter`). The `NSNotificationQueue` class contributes two important features to the Foundation Kit's notification mechanism: the coalescing of notifications and asynchronous posting.

## Notification Queue Basics

Using the `NSNotificationCenter`'s `postNotification:` method and its variants, you can post a notification to a notification center. However, the invocation of the method is synchronous: before the posting object can resume its thread of execution, it must wait until the notification center dispatches the notification to all observers and returns. A notification queue, on the other hand, maintains notifications (instances of `NSNotification`) generally in a First In First Out (FIFO) order. When a notification rises to the front of the queue, the queue posts it to the notification center, which in turn dispatches the notification to all objects registered as observers.

Every thread has a default notification queue, which is associated with the default notification center for the task. You can create your own notification queues and have multiple queues per center and thread.

## Posting Notifications Asynchronously

With `NSNotificationQueue`'s `enqueueNotification:postingStyle:` and `enqueueNotification:postingStyle:coalesceMask:forModes:` methods, you can post a notification asynchronously to the current thread by putting it in a queue. These methods immediately return to the invoking object after putting the notification in the queue.

**Note:** When the thread where a notification is enqueued terminates before the notification queue posts the notification to its notification center, the notification goes unposted. See "Delivering Notifications To Particular Threads" (page 21) to learn how to post a notification to a different thread.

The notification queue is emptied and its notifications posted based on the posting style and run loop mode specified in the enqueuing method. The mode argument specifies the run loop mode in which the queue will be emptied. For example, if you specify `NSModalPanelRunLoopMode`, the notifications will be posted only when the run loop is in this mode. If the run loop is not currently in this mode, the notifications wait until the next time that mode is entered. See "Input Modes" for more information on run loop modes.

Posting to a notification queue can occur in one of three different styles: `NSPostASAP`, `NSPostWhenIdle`, and `NSPostNow`. These styles are described in the following sections.

## Posting As Soon As Possible

Any notification queued with the `NSPostASAP` style is posted to the notification center when the current iteration of the run loop completes, assuming the current run loop mode matches the requested mode. (If the requested and current modes are different, the notification is posted when the requested mode is entered.) Because the run loop can make multiple callouts during each iteration, the notification may or may not get delivered as soon as the current callout exits and control returns to the run loop. Other callouts may take place first, such as a timer or source firing or other asynchronous notifications being delivered.

You typically use the `NSPostASAP` posting style for an expensive resource, such as the display server. When many clients draw on the window buffer during a callout from the run loop, it is expensive to flush the buffer to the display server after every draw operation. In this situation, each `draw...` method enqueues some notification such as "FlushTheServer" with coalescing on name and object specified and with a posting style of `NSPostASAP`. As a result, only one of those notifications is dispatched at the end of the run loop and the window buffer is flushed only once.

## Posting When Idle

A notification queued with the `NSPostWhenIdle` style is posted only when the run loop is in a wait state. In this state, there's nothing in the run loop's input channels, be it timers or other asynchronous events. A typical example of queuing with the `NSPostWhenIdle` style occurs when the user types text, and the program displays the size of the text in bytes somewhere. It would be very expensive (and not very useful) to update the text field size after each character the user types, especially if the user types quickly. In this case, the program queues a notification, such as "ChangeTheDisplayedSize," with coalescing turned on and a posting style of `NSPostWhenIdle` after each character typed. When the user stops typing, the single "ChangeTheDisplayedSize" notification in the queue (due to coalescing) is posted when the run loop enters its wait state and the display is updated. Note that a run loop that is about to exit (which occurs when all of the input channels have expired) is not in a wait state and thus will not post a notification.

## Posting Immediately

A notification queued with `NSPostNow` is posted immediately after coalescing to the notification center. You queue a notification with `NSPostNow` (or post one with `postNotification:`) when you do not require asynchronous calling behavior. For many programming situations, synchronous behavior is not only allowable but desirable: You want the notification center to return after dispatching so you can be sure that observing objects have received and processed the notification. Of course, you should use `enqueueNotification...` with `NSPostNow` rather than use `postNotification:` when there are similar notifications in the queue that you want to remove through coalescing.

# Coalescing Notifications

In some situations, you may want to post a notification if a given event occurs at least once, but you want to post no more than one notification even if the event occurs multiple times. For example, in an application that receives data in discrete packets, upon receipt of a packet you may wish to post a notification to signify that the data needs to be processed. If multiple packets arrive within a given time period, however, you do

not want to post multiple notifications. Moreover, the object that posts these notifications may not have any way of knowing whether more packets are coming or not, whether the posting method is called in a loop or not.

In some situations it may be possible to simply set a Boolean flag (whether an instance variable of an object or a global variable) to denote that an event has occurred and to suppress posting of further notifications until the flag is cleared. If this is not possible, however, in this situation you cannot directly use `NSNotificationCenter` since its behavior is synchronous—notifications are posted before returning, thus there is no opportunity for "ignoring" duplicate notifications; moreover, an `NSNotificationCenter` instance has no way of knowing whether more notifications are coming or not.

Rather than posting a notification to a notification center, therefore, you can add the notification to an `NSNotificationQueue` instance specifying an appropriate option for **coalescing**. Coalescing is a process that removes from a queue notifications that are similar in some way to a notification that was queued earlier. You indicate the criteria for similarity by specifying one or more of the following constants in the third argument of the `enqueueNotification:postingStyle:coalesceMask:forModes:` method.

| | |
|---|---|
| `NSNotificationNoCoalescing` | Do not coalesce notifications in the queue. |
| `NSNotificationCoalescingOnName` | Coalesce notifications with the same name. |
| `NSNotificationCoalescingOnSender` | Coalesce notifications with the same object. |

You can perform a bitwise-OR operation with the `NSNotificationCoalescingOnName` and `NSNotificationCoalescingOnSender` constants to specify coalescing using both the notification name and notification object. The following example illustrates how you might use a queue to ensure that, within a given event loop cycle, all notifications named `MyNotificationName` are coalesced into a single notification.

```
// MyNotificationName defined globally
NSString *MyNotificationName = @"MyNotification";

id object = /* the object associated with the notification *;/
NSNotification *myNotification =
        [NSNotification notificationWithName:MyNotificationName object:object]
[[NSNotificationQueue defaultQueue]
        enqueueNotification:myNotification
        postingStyle:NSPostWhenIdle
        coalesceMask:NSNotificationCoalescingOnName
        forModes:nil];
```

# Registering for a Notification

You can register for notifications from within your own application or other applications. See "Registering for Local Notifications" (page 15) for the former and "Registering for Distributed Notifications" (page 16) for the latter. To unregister for a notification, which must be done when your object is deallocated, see "Unregistering an Observer" (page 17).

## Registering for Local Notifications

An object registers itself to receive a notification by invoking the notification center method `addObserver:selector:name:object:` (or the `addObserver` method in Java), specifying the message the notification center should send to the observer, the name of the notification it wants to receive, and about which object. However, the observer need not specify both the name and the object. If it specifies only the object, it will receive all notifications containing that object. If the object specifies only a notification name, it will receive that notification every time it's posted, regardless of the object associated with it.

It is possible for an observer to register to receive more than one message for the same notification. In such a case, the observer will receive all messages it is registered to receive for the notification, but the order in which it receives them cannot be determined.

If an observer later decides it no longer needs to receive notifications (for example, if it is being deallocated), it can remove itself from the notification center's list of observers with the methods `removeObserver:` or `removeObserver:name:object:` (or the `removeObserver` method in Java).

Normally, you register objects with the task's default notification center. You obtain the default object using the `defaultCenter` class method.

As an example of using the notification center to receive notifications, suppose your program can perform a number of conversions on text (for instance, RTF to ASCII). You have defined a class of objects that perform those conversions, `Converter`. `Converter` objects might be added or removed during program execution. Your program has a client object that wants to be notified when converters are added or removed, allowing the application to reflect the available services in a pop-up menu. The client object would register itself as an observer by sending the following messages to the notification center:

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(objectAddedToConverterList:)
    name:@"ConverterAdded" object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(objectRemovedFromConverterList:)
    name:@"ConverterRemoved" object:nil];
```

By passing `nil` as the object to observe, the client object (`self`) is notified when any object posts a `"ConverterAdded"` or `"ConverterRemoved"` notification.

When a user installs or removes a Converter, the Converter posts either a `"ConverterAdded"` or `"ConverterRemoved"` notification to the notification center. The notification center notifies all observers who are interested in the notification by invoking the method they specified in the selector argument of `addObserver:selector:name:object:`. In the case of our example observer, the selector is either `objectAddedToConverterList:` or `objectRemovedFromConverterList:`. Assume the Converter class has an instance method `converterName` that returns the name of the Converter object. Then the `objectAddedToConverterList:` method might have the following implementation:

```
- (void)objectAddedToConverterList:(NSNotification *)notification
{
    Converter *addedConverter = [notification object];

    // Add this to our popup (it will only be added if not there)...
    [myPopUpButton addItem:[addedConverter converterName]];
}
```

The Converters don't need to know anything about the pop-up list or any other aspect of the user interface to your program.

# Registering for Distributed Notifications

An object registers itself to receive a notification by sending the `addObserver:selector:name:object:suspensionBehavior:` method to an `NSDistributedNotificationCenter` object, specifying the message the notification should send, the name of the notification it wants to receive, the identifying string to match (the *`object`* argument), and the behavior to follow if notification delivery is suspended.

Because the posting object and the observer may be in different tasks, notifications can't contain pointers to arbitrary objects. Therefore, the `NSDistributedNotificationCenter` class requires notifications to use an `NSString` object as the *`object`* argument. Notification matching is done based on this string, rather than an object pointer. You should check the documentation of the object posting the notification to see what it uses as its identifying string.

When a task is no longer interested in receiving notifications immediately, it may suspend notification delivery. This is often done when the application is hidden, or is put into the background. (The `NSApplication` object automatically suspends delivery when the application is not active.) The *`suspensionBehavior`* argument in the `addObserver` method identifies how arriving notifications should be handled while delivery is suspended. There are four different types of suspension behavior, each useful in different circumstances.

| Suspension Behavior | Description |
| --- | --- |
| `NSNotification-SuspensionBehaviorDrop` | The server does not queue any notifications with this name and object until it receives the `setSuspended:NO` message. |
| `NSNotification-SuspensionBehavior-Coalesce` | The server queues only the last notification of the specified name and object; earlier notifications are dropped. In cover methods for which suspension behavior is not an explicit argument, `NSNotification-SuspensionBehaviorCoalesce` is the default. |

| Suspension Behavior | Description |
| --- | --- |
| `NSNotification-SuspensionBehaviorHold` | The server holds all matching notifications until the queue has been filled (queue size determined by the server) at which point the server may flush queued notifications. |
| `NSNotification-SuspensionBehavior-DeliverImmediately` | The server delivers notifications matching this registration irrespective of whether it has received the `setSuspended:YES` message. When a notification with this suspension behavior is matched, it has the effect of first flushing any queued notifications. The effect is as if the server received `setSuspended:NO` while the application is suspended, followed by the notification in question being delivered, followed by a transition back to the previous suspended or unsuspended state. |

You suspend notifications by sending `setSuspended:YES` to the distributed notification center. While notifications are suspended, the notification server handles notifications destined for the task that suspended notification delivery according to the suspension behavior specified by the observers when they registered to receive notifications. When the task unsuspends notification delivery, all queued notifications are delivered immediately. In applications using Application Kit, the `NSApplication` object automatically suspends notification delivery when the application is not active.

Note that a notification destined for an observer that registered with `NSNotificationSuspensionBehaviorDeliverImmediately`, automatically flushes the queue as it is delivered, causing all queued notifications to be delivered at that time as well.

The suspended state can be overridden by the poster of a notification. If the notification is urgent, such as a warning of a server being shut down, the poster can force the notification to be delivered immediately to all observers by posting the notification with the `NSDistributedNotificationCenter postNotificationName:object:userInfo:deliverImmediately:` method with the *`deliverImmediately`* argument `YES`.

# Unregistering an Observer

If you deallocate an object that is observing notifications, you need to tell the notification center to stop sending it notifications. Otherwise, the next notification gets sent to a nonexistent object and the program crashes. You can send the following message to completely remove an object as an observer of local notifications, regardless of how many objects and notifications for which it registered itself:

```
[[NSNotificationCenter defaultCenter] removeObserver:self];
```

For observers of distributed notifications send:

```
[[NSDistributedNotificationCenter defaultCenter] removeObserver:self];
```

Use the more specific `removeObserver...` methods that specify the notification name and observed object to selectively unregister an object for particular notifications.

# Posting a Notification

You can post notifications within your own application or make them available to other applications. See "Posting Local Notifications" (page 19) for the former and "Posting Distributed Notifications" (page 20) for the latter.

## Posting Local Notifications

You can create a notification object with the `NSNotification` class Java constructor or the Objective-C methods `notificationWithName:object:` or `notificationWithName:object:userInfo:`. You then post the notification object to a notification center using the `postNotification:` instance method. `NSNotification` objects are immutable objects, so once created, they cannot be modified.

However, you normally don't create your own notifications directly. The methods `postNotificationName:object:` and `postNotificationName:object:userInfo:` of the `NSNotificationCenter` class allow you to conveniently post a notification without creating it first. In Java, you use the `postNotification` method with the notification properties as arguments.

In each case, you usually post the notification to the task's default notification center. You obtain the default object using the `defaultCenter` class method.

As an example of using the notification center to post a notification, consider the example from "Registering for Local Notifications" (page 15). You have a program that can perform a number of conversions on text (for instance, RTF to ASCII). The conversions are handled by a class of objects (`Converter`) that can be added or removed during program execution. Your program may have other objects that want to be notified when converters are added or removed, but the `Converter` objects do not need to know who these objects are or what they do. You thus declare two notifications, `"ConverterAdded"` and `"ConverterRemoved"`, which you post when the given event occurs.

When a user installs or removes a converter, it sends one of the following messages to the notification center:

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"ConverterAdded" object:self];
```

or

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"ConverterRemoved" object:self];
```

The notification center then identifies which objects (if any) are interested in these notifications and notifies them.

If there are other objects of interest to the observer (besides the notification name and observed object), place them in the notification's optional dictionary or use `postNotificationName:object:userInfo:`.

# Posting Distributed Notifications

Posting distributed notifications is much the same as for posting local notifications. You can create an `NSNotification` object manually and post with `postNotification:` or use one of the `NSDistributedNotificationCenter` convenience methods. The only differences are that the notification object must be a string object and the optional user-info dictionary can contain only property list objects, such as `NSString` and `NSNumber` (see "Property Lists" for details on property lists).

An observer of a given notification may be in a suspended state and not processing notifications immediately. If an object posting a notification wants to ensure that all observers receive the notification immediately (for example, if the notification is a warning that a server is about to shut down), it can invoke `postNotificationName:object:userInfo:deliverImmediately:` with *`deliverImmediately:YES`*. The notification center delivers the notification as if the observers had registered with `NSNotificationSuspensionBehaviorDeliverImmediately` (further described in "Registering for Distributed Notifications" (page 16)). Delivery is not guaranteed, however. For example, the task receiving the notifications may be too busy to process and accept queued notifications. In this case, the notification is dropped.

# Delivering Notifications To Particular Threads

Regular notification centers deliver notifications on the thread in which the notification was posted. Distributed notification centers deliver notifications are delivered on the main thread (prior to Mac OS X v10.3, they were delivered on an undefined thread). At times, you may require notifications to be delivered on a particular thread that is determined by you instead of the notification center. For example, if an object running in a background thread is listening for notifications from the user interface, such as a window closing, you would like to receive the notifications in the background thread instead of the main thread. In these cases, you must capture the notifications as they are delivered on the default thread and redirect them to the appropriate thread.

One way to redirect notifications is to use a custom notification queue (not an `NSNotificationQueue` object) to hold any notifications that are received on incorrect threads and then process them on the correct thread. This technique works as follows. You register for a notification normally. When a notification arrives, you test whether the current thread is the thread that should handle the notification. If it is the wrong thread, you store the notification in a queue and then send a signal to the correct thread, indicating that a notification needs processing. The other thread receives the signal, removes the notification from the queue, and processes the notification.

To implement this technique, your observer object needs to have instance variables for the following values: a mutable array to hold the notifications, a communication port for signaling the correct thread (a Mach port), a lock to prevent multithreading conflicts with the notification array, and a value that identifies the correct thread (an `NSThread` object). You also need methods to setup the variables, to process the notifications, and to receive the Mach messages. Here are the necessary definitions to add to the class of your observer object.

```
@interface MyThreadedClass: NSObject
{
    /* Threaded notification support */
    NSMutableArray *notifications;
    NSThread *notificationThread;
    NSLock *notificationLock;
    NSMachPort *notificationPort;
}

- (void) setUpThreadingSupport;
- (void) handleMachMessage:(void *) msg;
- (void) processNotification:(NSNotification *) notification;
@end
```

Before registering for any notifications, you need to initialize the instance variables. The following method initializes the queue and lock objects, retains a reference to the current thread object, and creates a Mach communication port, which it adds to the current thread's run loop.

```
- (void) setUpThreadingSupport {
    if ( notifications ) return;

    notifications      = [[NSMutableArray alloc] init];
    notificationLock   = [[NSLock alloc] init];
    notificationThread = [[NSThread currentThread] retain];
```

```
    notificationPort = [[NSMachPort alloc] init];
    [notificationPort setDelegate:self];
    [[NSRunLoop currentRunLoop] addPort:notificationPort
            forMode:(NSString *) kCFRunLoopCommonModes];
}
```

After this method runs, any messages sent to `notificationPort` are received in the run loop of the thread that first ran this method. If the receiving thread's run loop is not running when the Mach message arrives, the kernel holds on to the message until the next time the run loop is entered. The receiving thread's run loop sends the incoming messages to the `handleMachMessage:` method of the port's delegate.

In this implementation, no information is contained in the messages sent to `notificationPort`. Instead, the information passed between threads is contained in the notification array. When a Mach message arrives, the `handleMachMessage:` method ignores the contents of the message and just checks the `notifications` array for any notifications that need processing. The notifications are removed from the array and forwarded to the real notification processing method. Because port messages may get dropped if too many are sent simultaneously, the `handleMachMessage:` method iterates over the array until it is empty. The method must acquire a lock when accessing the notification array to prevent conflicts between one thread adding notifications and another removing notifications from the array.

```
- (void) handleMachMessage:(void *) msg {
    [notificationLock lock];
    while ( [notifications count] ) {
       NSNotification *notification = [[notifications objectAtIndex:0] retain];
        [notifications removeObjectAtIndex:0];
        [notificationLock unlock];
        [self processNotification:notification];
        [notification release];
        [notificationLock lock];
    };
    [notificationLock unlock];
}
```

When a notification is delivered to your object, the method that receives the notification must identify whether it is running in the correct thread or not. If it is the correct thread, the notification is processed normally. If it is the wrong thread, the notification is added to the queue and the notification port signaled.

```
- (void) processNotification:(NSNotification *) notification {
    if( [NSThread currentThread] != notificationThread ) {
        // Forward the notification to the correct thread
        [notificationLock lock];
        [notifications addObject:notification];
        [notificationLock unlock];
        [notificationPort sendBeforeDate:[NSDate date]
                components:nil
                from:nil
                reserved:0];
    }
    else {
        // Process the notification here;
    }
}
```

Finally, to register for a notification that you want delivered on the current thread, regardless of the thread in which it may be posted, you must initialize your object's notification instance variables by invoking `setUpThreadingSupport` and then register for the notification normally, specifying the special notification processing method as the selector.

```
[self setupThreadingSupport];
[[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(processNotification:)
        name:@"NotificationName"
        object:nil];
```

This implementation is limited in several aspects. First, all threaded notifications processed by this object must pass through the same method (`processNotification:`). Second, each object must provide its own implementation and communication port. A better, but more complex, implementation would generalize the behavior into either a subclass of `NSNotificationCenter` or a separate class that would have one notification queue for each thread and be able to deliver notifications to multiple observer objects and methods.

# Document Revision History

This table describes the changes to *Notification Programming Topics for Cocoa*.

| Date | Notes |
| --- | --- |
| 2007-05-03 | Noted that distributed notification centers deliver notifications on the main thread. |
| 2007-01-08 | Corrected minor typographical errors. |
| 2006-11-07 | Enhanced description of coalescing notifications. |
| 2006-04-04 | Clarified on which thread an enqueued notification is posted. Changed title from "Notifications." |
| | Specified that notifications are posted on the same thread where they are enqueued in "Posting Notifications Asynchronously" (page 11). |
| | Indicated that notifications are used throughout the Cocoa frameworks and pointed out where such notifications are described in "Notifications" (page 7). |
| 2002-11-12 | Revision history was added to existing topic. It will be used to record changes to the content of the topic. |