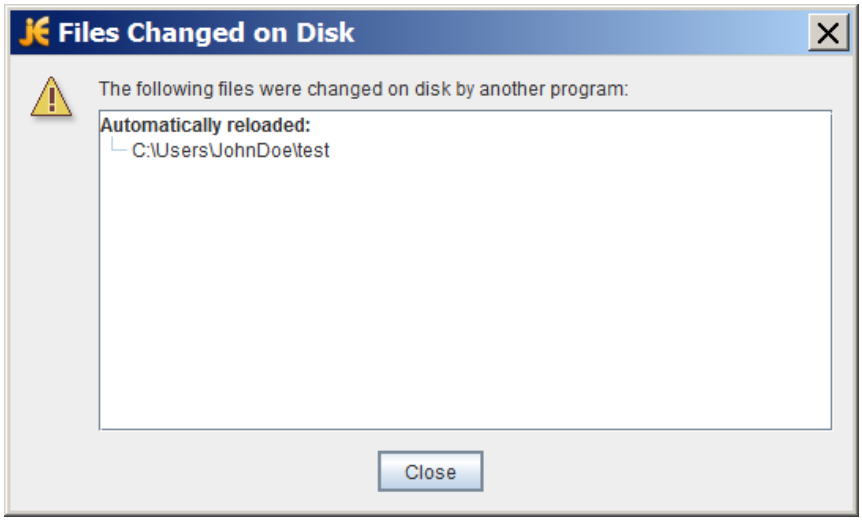
Trail: Essential Classes
Lesson: Basic I/O
Section: File I/O (Featuring NIO.2)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Watching a Directory for Changes

Have you ever found yourself editing a file, using an IDE or another editor, and a dialog box appears to inform you that one of the open files has changed on the file system and needs to be reloaded? Or perhaps, like the NetBeans IDE, the application just quietly updates the file without notifying you. The following sample dialog box shows how this notification looks with the free editor, jEdit:



jEdit Dialog Box Showing That a Modified File Is Detected

To implement this functionality, called *file change notification*, a program must be able to detect what is happening to the relevant directory on the file system. One way to do so is to poll the file system looking for changes, but this approach is inefficient. It does not scale to applications that have hundreds of open files or directories to monitor.

The `java.nio.file` package provides a file change notification API, called the Watch Service API. This API enables you to register a directory (or directories) with the watch service. When registering, you tell the service which types of events you are interested in: file creation, file deletion, or file modification. When the service detects an event of interest, it is forwarded to the registered process. The registered process has a thread (or a pool of threads) dedicated to watching for any events it has registered for. When an event comes in, it is handled as needed.

This section covers the following:

Watch Service Overview

The `WatchService` API is fairly low level, allowing you to customize it. You can use it as is, or you can choose to create a high-level API on top of this mechanism so that it is suited to your particular needs.

Here are the basic steps required to implement a watch service:

- Create a `WatchService` "watcher" for the file system.
- For each directory that you want monitored, register it with the watcher. When registering a directory, you specify the type of events for which you want notification. You receive a `WatchKey` instance for each directory that you register.
- Implement an infinite loop to wait for incoming events. When an event occurs, the key is signaled and placed into the watcher's queue.

- Retrieve the key from the watcher's queue. You can obtain the file name from the key.
- Retrieve each pending event for the key (there might be multiple events) and process as needed.
- Reset the key, and resume waiting for events.
- Close the service: The watch service exits when either the thread exits or when it is closed (by invoking its `closed` method).

`WatchKeys` are thread-safe and can be used with the `java.nio.concurrent` package. You can dedicate a thread pool to this effort.

Try It Out

Because this API is more advanced, try it out before proceeding. Save the `WatchDir` example to your computer, and compile it. Create a `test` directory that will be passed to the example. `WatchDir` uses a single thread to process all events, so it blocks keyboard input while waiting for events. Either run the program in a separate window, or in the background, as follows:

```
java WatchDir test &
```

Play with creating, deleting, and editing files in the `test` directory. When any of these events occurs, a message is printed to the console. When you have finished, delete the `test` directory and `WatchDir` exits. Or, if you prefer, you can manually kill the process.

You can also watch an entire file tree by specifying the `-r` option. When you specify `-r`, `WatchDir` walks the file tree, registering each directory with the watch service.

Creating a Watch Service and Registering for Events

The first step is to create a new `WatchService` by using the `newWatchService` method in the `FileSystem` class, as follows:

```
WatchService watcher = FileSystems.getDefault().newWatchService();
```

Next, register one or more objects with the watch service. Any object that implements the `Watchable` interface can be registered. The `Path` class implements the `Watchable` interface, so each directory to be monitored is registered as a `Path` object.

As with any `Watchable`, the `Path` class implements two `register` methods. This page uses the two-argument version, `register(WatchService, WatchEvent.Kind<?>...)`. (The three-argument version takes a `WatchEvent.Modifier`, which is not currently implemented.)

When registering an object with the watch service, you specify the types of events that you want to monitor. The supported `StandardWatchEventKinds` event types follow:

- `ENTRY_CREATE` – A directory entry is created.
- `ENTRY_DELETE` – A directory entry is deleted.
- `ENTRY_MODIFY` – A directory entry is modified.
- `OVERFLOW` – Indicates that events might have been lost or discarded. You do not have to register for the `OVERFLOW` event to receive it.

The following code snippet shows how to register a `Path` instance for all three event types:

```
import static java.nio.file.StandardWatchEventKinds.*;

Path dir = ...;
try {
    WatchKey key = dir.register(watcher,
                           ENTRY_CREATE,
                           ENTRY_DELETE,
                           ENTRY_MODIFY);
} catch (IOException x) {
    System.err.println(x);
}
```

Processing Events

The order of events in an event processing loop follow:

1. Get a watch key. Three methods are provided:
    - `poll` – Returns a queued key, if available. Returns immediately with a `null` value, if unavailable.
    - `poll(long, TimeUnit)` – Returns a queued key, if one is available. If a queued key is not immediately available, the program waits until the specified time. The `TimeUnit` argument determines whether the specified time is nanoseconds, milliseconds, or some other unit of time.
    - `take` – Returns a queued key. If no queued key is available, this method waits.
2. Process the pending events for the key. You fetch the `List` of `WatchEvents`from the `pollEvents` method.
3. Retrieve the type of event by using the `kind` method. No matter what events the key has registered for, it is possible to receive an `OVERFLOW` event. You can choose to handle the overflow or ignore it, but you should test for it.
4. Retrieve the file name associated with the event. The file name is stored as the context of the event, so the `context` method is used to retrieve it.
5. After the events for the key have been processed, you need to put the key back into a `ready` state by invoking `reset`. If this method returns `false`, the key is no longer valid and the loop can exit. This step is very **important**. If you fail to invoke `reset`, this key will not receive any

further events.

A watch key has a state. At any given time, its state might be one of the following:

- `Ready` indicates that the key is ready to accept events. When first created, a key is in the ready state.
- `Signaled` indicates that one or more events are queued. Once the key has been signaled, it is no longer in the ready state until the `reset` method is invoked.
- `Invalid` indicates that the key is no longer active. This state happens when one of the following events occurs:
  - The process explicitly cancels the key by using the `cancel` method.
  - The directory becomes inaccessible.
  - The watch service is closed.

Here is an example of an event processing loop. It is taken from the `Email` example, which watches a directory, waiting for new files to appear. When a new file becomes available, it is examined to determine if it is a `text/plain` file by using the `probeContentType(Path)` method. The intention is that `text/plain` files will be emailed to an alias, but that implementation detail is left to the reader.

The methods specific to the watch service API are shown in bold:

```
for (;;) {

    // wait for key to be signaled
    WatchKey key;
    try {
        key = watcher.take();
    } catch (InterruptedException x) {
        return;
    }

    for (WatchEvent<?> event: key.pollEvents()) {
        WatchEvent.Kind<?> kind = event.kind();

        // This key is registered only
        // for ENTRY_CREATE events,
        // but an OVERFLOW event can
        // occur regardless if events
        // are lost or discarded.
        if (kind == OVERFLOW) {
            continue;
        }

        // The filename is the
        // context of the event.
        WatchEvent<Path> ev = (WatchEvent<Path>)event;
        Path filename = ev.context();

        // Verify that the new
        //  file is a text file.
        try {
            // Resolve the filename against the directory.
            // If the filename is "test" and the directory is "foo",
            // the resolved name is "test/foo".
            Path child = dir.resolve(filename);
            if (!Files.probeContentType(child).equals("text/plain")) {
                System.err.format("New file '%s'" +
                    " is not a plain text file.%n", filename);
                continue;
            }
        } catch (IOException x) {
            System.err.println(x);
            continue;
        }

        // Email the file to the
        //  specified email alias.
        System.out.format("Emailing file %s%n", filename);
        //Details left to reader....
    }

    // Reset the key -- this step is critical if you want to
    // receive further watch events.  If the key is no longer valid,
    // the directory is inaccessible so exit the loop.
    boolean valid = key.reset();
    if (!valid) {
        break;
```

```
        }
    }
```

Retrieving the File Name

The file name is retrieved from the event context. The `Email` example retrieves the file name with this code:

```
WatchEvent<Path> ev = (WatchEvent<Path>)event;
Path filename = ev.context();
```

When you compile the `Email` example, it generates the following error:

```
Note: Email.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This error is a result of the line of code that casts the `WatchEvent<T>` to a `WatchEvent<Path>`. The `WatchDir` example avoids this error by creating a utility `cast` method that suppresses the unchecked warning, as follows:

```
@SuppressWarnings("unchecked")
static <T> WatchEvent<T> cast(WatchEvent<?> event) {
    return (WatchEvent<Path>)event;
}
```

If you are unfamiliar with the `@SuppressWarnings` syntax, see Annotations.

When to Use and Not Use This API

The Watch Service API is designed for applications that need to be notified about file change events. It is well suited for any application, like an editor or IDE, that potentially has many open files and needs to ensure that the files are synchronized with the file system. It is also well suited for an application server that watches a directory, perhaps waiting for `.jsp` or `.jar` files to drop, in order to deploy them.

This API is *not* designed for indexing a hard drive. Most file system implementations have native support for file change notification. The Watch Service API takes advantage of this support where available. However, when a file system does not support this mechanism, the Watch Service will poll the file system, waiting for events.

Previous page: Finding Files
Next page: Other Useful Methods