How To Use the POIFS APIs

by Marc Johnson

1. How To Use the POIFS APIs

This document describes how to use the POIFS APIs to read, write, and modify files that employ a POIFS-compatible data structure to organize their content.

1.1. Revision History

- 02.10.2002 - completely rewritten from original documents on sourceforge

1.2. Target Audience

This document is intended for Java developers who need to use the POIFS APIs to read, write, or modify files that employ a POIFS-compatible data structure to organize their content. It is not necessary for developers to understand the POIFS data structures, and an explanation of those data structures is beyond the scope of this document. It is expected that the members of the target audience will understand the rudiments of a hierarchical file system, and familiarity with the event pattern employed by Java APIs such as AWT would be helpful.

1.3. Glossary

This document attempts to be consistent in its terminology, which is defined here:

Term	*Definition*
Directory	A special file that may contain other directories and documents.
DirectoryEntry	Representation of a directory within another directory.
Document	A file containing data, such as word processing data or a spreadsheet workbook.
DocumentEntry	Representation of a document within a directory.

Entry	Representation of a file in a directory.
File	A named entity, managed and contained by the file system.
File System	The POIFS data structures, plus the contained directories and documents, which are maintained in a hierarchical directory structure.
Root Directory	The directory at the base of a file system. All file systems have a root directory. The POIFS APIs will not allow the root directory to be removed or renamed, but it can be accessed for the purpose of reading its contents or adding files (directories and documents) to it.

2. Reading a File System

This section covers reading a file system. There are two ways to read a file system; these techniques are sketched out in the following table, and then explained in greater depth in the sections following the table.

Technique	*Advantages*	*Disadvantages*
Conventional Reading	Simpler API similar to reading a conventional file system. Can read documents in any order.	All files are resident in memory, whether your application needs them or not.
Event-Driven Reading	Reduced footprint -- only the documents you care about are processed. Improved performance -- no time is wasted reading the documents you're not interested in.	More complicated API. Need to know in advance which documents you want to read. No control over the order in which the documents are read. No way to go back and get additional documents except to re-read the file system, which may not be possible, e.g., if the file system is being read from an input stream that lacks random access support.

2.1. Conventional Reading

In this technique for reading, the entire file system is loaded into memory, and the entire directory tree can be walked by an application, reading specific documents at the

application's leisure.

2.1.1. Preparation

Before an application can read a file from the file system, the file system needs to be loaded into memory. This is done by using the `org.apache.poi.poifs.filesystem.POIFSFileSystem` class. Once the file system has been loaded into memory, the application may need the root directory. The following code fragment will accomplish this preparation stage:

```
// need an open InputStream; for a file-based system, this would be appropriate:
// InputStream stream = new FileInputStream(fileName);
POIFSFileSystem fs;
try
{
    fs = new POIFSFileSystem(inputStream);
}
catch (IOException e)
{
    // an I/O error occurred, or the InputStream did not provide a compatible
    // POIFS data structure
}
DirectoryEntry root = fs.getRoot();
```

Assuming no exception was thrown, the file system can then be read.

Note: loading the file system can take noticeable time, particularly for large file systems.

2.1.2. Reading the Directory Tree

Once the file system has been loaded into memory and the root directory has been obtained, the root directory can be read. The following code fragment shows how to read the entries in an `org.apache.poi.poifs.filesystem.DirectoryEntry` instance:

```
// dir is an instance of DirectoryEntry ...
for (Iterator iter = dir.getEntries(); iter.hasNext(); )
{
    Entry entry = (Entry)iter.next();
    System.out.println("found entry: " + entry.getName());
    if (entry instanceof DirectoryEntry)
    {
        // .. recurse into this directory
    }
    else if (entry instanceof DocumentEntry)
    {
        // entry is a document, which you can read
    }
    else
```

```
    {
        // currently, either an Entry is a DirectoryEntry or a DocumentEntry,
        // but in the future, there may be other entry subinterfaces. The
        // internal data structure certainly allows for a lot more entry types.
    }
}
```

2.1.3. Reading a Specific Document

There are a couple of ways to read a document, depending on whether the document resides in the root directory or in another directory. Either way, you will obtain an `org.apache.poi.poifs.filesystem.DocumentInputStream` instance.

2.1.3.1. DocumentInputStream

The DocumentInputStream class is a simple implementation of InputStream that makes a few guarantees worth noting:

- `available()` always returns the number of bytes in the document from your current position in the document.
- `markSupported()` returns `true`.
- `mark(int limit)` ignores the limit parameter; basically the method marks the current position in the document.
- `reset()` takes you back to the position when `mark()` was last called, or to the beginning of the document if `mark()` has not been called.
- `skip(long n)` will take you to your current position + n (but not past the end of the document).

The behavior of `available` means you can read in a document in a single read call like this:

```
byte[] content = new byte[ stream.available() ];
stream.read(content);
stream.close();
```

The combination of `mark`, `reset`, and `skip` provide the basic mechanisms needed for random access of the document contents.

2.1.3.2. Reading a Document From the Root Directory

If the document resides in the root directory, you can obtain a `DocumentInputStream` like this:

```
// load file system
try
{
```

```
    DocumentInputStream stream = filesystem.createDocumentInputStream(documentName);
    // process data from stream
}
catch (IOException e)
{
    // no such document, or the Entry represented by documentName is not a
    // DocumentEntry
}
```

2.1.3.3. Reading a Document From an Arbitrary Directory

A more generic technique for reading a document is to obtain an `org.apache.poi.poifs.filesystem.DirectoryEntry` instance for the directory containing the desired document (recall that you can use `getRoot()` to obtain the root directory from its file system). From that DirectoryEntry, you can then obtain a `DocumentInputStream` like this:

```
DocumentEntry document = (DocumentEntry)directory.getEntry(documentName);
DocumentInputStream stream = new DocumentInputStream(document);
```

2.2. Event-Driven Reading

The event-driven API for reading documents is a little more complicated and requires that your application know, in advance, which files it wants to read. The benefit of using this API is that each document is in memory just long enough for your application to read it, and documents that you never read at all are not in memory at all. When you're finished reading the documents you wanted, the file system has no data structures associated with it at all and can be discarded.

2.2.1. Preparation

The preparation phase involves creating an instance of `org.apache.poi.poifs.eventfilesystem.POIFSReader` and to then register one or more `org.apache.poi.poifs.eventfilesystem.POIFSReaderListener` instances with the `POIFSReader`.

```
POIFSReader reader = new POIFSReader();
// register for everything
reader.registerListener(myOmnivorousListener);
// register for selective files
reader.registerListener(myPickyListener, "foo");
reader.registerListener(myPickyListener, "bar");
// register for selective files
reader.registerListener(myOtherPickyListener, new POIFSDocumentPath(),
     "fubar");
```

```
reader.registerListener(myOtherPickyListener, new POIFSDocumentPath(
    new String[] { "usr", "bin" ), "fubar");
```

2.2.2. POIFSReaderListener

`org.apache.poi.poifs.eventfilesystem.POIFSReaderListener` is an interface used to register for documents. When a matching document is read by the `org.apache.poi.poifs.eventfilesystem.POIFSReader`, the `POIFSReaderListener` instance receives an `org.apache.poi.poifs.eventfilesystem.POIFSReaderEvent` instance, which contains an open `DocumentInputStream` and information about the document.

A `POIFSReaderListener` instance can register for individual documents, or it can register for all documents; once it has registered for all documents, subsequent (and previous!) registration requests for individual documents are ignored. There is no way to unregister a `POIFSReaderListener`.

Thus, it is possible to register a single `POIFSReaderListener` for multiple documents - one, some, or all documents. It is guaranteed that a single `POIFSReaderListener` will receive exactly one notification per registered document. There is no guarantee as to the order in which it will receive notification of its documents, as future implementations of `POIFSReader` are free to change the algorithm for walking the file system's directory structure.

It is also permitted to register more than one `POIFSReaderListener` for the same document. There is no guarantee of ordering for notification of `POIFSReaderListener` instances that have registered for the same document when `POIFSReader` processes that document.

It is guaranteed that all notifications occur in the same thread. A future enhancement may be made to provide multi-threaded notifications, but such an enhancement would very probably be made in a new reader class, a `ThreadedPOIFSReader` perhaps.

The following table describes the three ways to register a `POIFSReaderListener` for a document or set of documents:

Method Signature	*What it does*
registerListener(POIFSReaderListener *listener*)	registers *listener* for all documents.
registerListener(POIFSReaderListener *listener*, String *name*)	registers *listener* for a document with the specified *name* in the root directory.
registerListener(POIFSReaderListener *listener*, POIFSDocumentPath *path*, String *name*)	registers *listener* for a document with the specified *name* in the directory described by *path*

2.2.3. POIFSDocumentPath

The `org.apache.poi.poifs.filesystem.POIFSDocumentPath` class is used to describe a directory in a POIFS file system. Since there are no reserved characters in the name of a file in a POIFS file system, a more traditional string-based solution for describing a directory, with special characters delimiting the components of the directory name, is not feasible. The constructors for the class are used as follows:

Constructor example	*Directory described*
new POIFSDocumentPath()	The root directory.
new POIFSDocumentPath(null)	The root directory.
new POIFSDocumentPath(new String[0])	The root directory.
new POIFSDocumentPath(new String[ ] { "foo", "bar"})	in Unix terminology, "/foo/bar".
new POIFSDocumentPath(new POIFSDocumentPath(new String[] { "foo" }), new String[ ] { "fu", "bar"})	in Unix terminology, "/foo/fu/bar".

2.2.4. Processing POIFSReaderEvent Events

Processing `org.apache.poi.poifs.eventfilesystem.POIFSReaderEvent` events is relatively easy. After all of the `POIFSReaderListener` instances have been registered with `POIFSReader`, the `POIFSReader.read(InputStream stream)` method is called.

Assuming that there are no problems with the data, as the `POIFSReader` processes the documents in the specified `InputStream`'s data, it calls registered `POIFSReaderListener` instances' `processPOIFSReaderEvent` method with a `POIFSReaderEvent` instance.

The `POIFSReaderEvent` instance contains information to identify the document (a `POIFSDocumentPath` object to identify the directory that the document is in, and the document name), and an open `DocumentInputStream` instance from which to read the document.

3. Writing a File System

Writing a file system is very much like reading a file system in that there are multiple ways to do so. You can load an existing file system into memory and modify it (removing files,

renaming files) and/or add new files to it, and write it, or you can start with a new, empty file system:

```
POIFSFileSystem fs = new POIFSFileSystem();
```

3.1. The Naming of Names

There are two restrictions on the names of files in a file system that must be considered when creating files:

1. The name of the file must not exceed 31 characters. If it does, the POIFS API will silently truncate the name to fit.
2. The name of the file must be unique within its containing directory. This seems pretty obvious, but if it isn't spelled out, there'll be hell to pay, to be sure. Uniqueness, of course, is determined *after* the name has been truncated, if the original name was too long to begin with.

3.2. Creating a Document

A document can be created by acquiring a `DirectoryEntry` and calling one of the two `createDocument` methods:

Method Signature	*Advantages*	*Disadvantages*
CreateDocument(String name, InputStream stream)	Simple API.	Increased memory footprint (document is in memory until file system is written).
CreateDocument(String name, int size, POIFSWriterListener writer)	Decreased memory footprint (only very small documents are held in memory, and then only for a short time).	More complex API. Determining document size in advance may be difficult. Lose control over when document is to be written.

Unlike reading, you don't have to choose between the in-memory and event-driven writing models; both can co-exist in the same file system.

Writing is initiated when the `POIFSFileSystem` instance's `writeFilesystem()` method is called with an `OutputStream` to write to.

The event-driven model is quite similar to the event-driven model for reading, in that the file system calls your `org.apache.poi.poifs.filesystem.POIFSWriterListener` when it's time to write your document, just as the `POIFSReader` calls your `POIFSReaderListener` when it's time to read your document. Internally, when `writeFilesystem()` is called,

the final POIFS data structures are created and are written to the specified `OutputStream`. When the file system needs to write a document out that was created with the event-driven model, it calls the `POIFSWriterListener` back, calling its `processPOIFSWriterEvent()` method, passing an `org.apache.poi.poifs.filesystem.POIFSWriterEvent` instance. This object contains the `POIFSDocumentPath` and name of the document, its size, and an open `org.apache.poi.poifs.filesystem.DocumentOutputStream` to which to write. A `DocumentOutputStream` is a wrapper over the `OutputStream` that was provided to the `POIFSFileSystem` to write to, and has the responsibility of making sure that the document your application writes fits within the size you specified for it.

3.3. Creating a Directory

Creating a directory is similar to creating a document, except that there's only one way to do so:

```
DirectoryEntry createdDir = existingDir.createDirectory(name);
```

3.4. Using POIFSFileSystem Directly To Create a Document Or Directory

As with reading documents, it is possible to create a new document or directory in the root directory by using convenience methods of POIFSFileSystem.

DirectoryEntry Method Signature	POIFSFileSystem Method Signature
createDocument(String name, InputStream stream)	createDocument(InputStream stream, String name)
createDocument(String name, int size, POIFSWriterListener writer)	createDocument(String name, int size, POIFSWriterListener writer)
createDirectory(String name)	createDirectory(String name)

4. Modifying a File System

It is possible to modify an existing POIFS file system, whether it's one your application has loaded into memory, or one which you are creating on the fly.

4.1. Removing a Document

Removing a document is simple: you get the `Entry` corresponding to the document and call its `delete()` method. This is a boolean method, but should always return `true`, indicating that the operation succeeded.

4.2. Removing a Directory

Removing a directory is also simple: you get the `Entry` corresponding to the directory and call its `delete()` method. This is a boolean method, but, unlike deleting a document, may not always return `true`, indicating that the operation succeeded. Here are the reasons why the operation may fail:

- The directory still has files in it (to check, call `isEmpty()` on its DirectoryEntry; is the return value `false`?)
- The directory is the root directory. You cannot remove the root directory.

4.3. Renaming a File

Regardless of whether the file is a directory or a document, it can be renamed, with one exception - the root directory has a special name that is expected by the components of a major software vendor's office suite, and the POIFS API will not let that name be changed. Renaming is done by acquiring the file's corresponding `Entry` instance and calling its `renameTo` method, passing in the new name.

Like `delete`, `renameTo` returns `true` if the operation succeeded, otherwise `false`. Reasons for failure include these:

- The new name is the same as another file in the same directory. And don't forget - if the new name is longer than 31 characters, it *will* be silently truncated. In its original length, the new name may have been unique, but truncated to 31 characters, it may not be unique any longer.
- You tried to rename the root directory.