HPSF HOW-TO

by Rainer Klute

1. How To Use the HPSF API

This HOW-TO is organized in four sections. You should read them sequentially because the later sections build upon the earlier ones.

1. The first section explains how to **read the most important standard properties** of a Microsoft Office document. Standard properties are things like title, author, creation date etc. It is quite likely that you will find here what you need and don't have to read the other sections.
2. The second section goes a small step further and focusses on **reading additional standard properties**. It also talks about **exceptions** that may be thrown when dealing with HPSF and shows how you can **read properties of embedded objects**.
3. The third section explains how to **write standard properties**. HPSF provides some high-level classes and methods which make writing of standard properties easy. They are based on the low-level writing functions explained in the fifth section.
4. The fourth section tells how to **read non-standard properties**. Non-standard properties are application-specific triples consisting of an ID, a type, and a value.
5. The fifth section tells you how to **write property set streams** using HPSF's low-level methods. You have to understand the fourth section before you should think about low-level writing properties. Check the Javadoc API documentation to find out about the details!

> **Note:**
>
> **Please note:** HPSF's writing functionality is **not** present in POI releases up to and including 2.5. In order to write properties you have to download a 3.0.x POI release, or retrieve the POI development version from the Subversion repository.

1.1. Reading Standard Properties

> **Note:**
>
> This section explains how to read the most important standard properties of a Microsoft Office document. Standard properties are things like title, author, creation date etc. This section introduces the **summary information stream** which is used to keep these properties. Chances are that you will find here what you need and don't have to read the other sections.

If all you are interested in is getting the textual content of all the document properties, such as for full text indexing, then take a look at `org.apache.poi.hpsf.extractor.HPSFPropertiesExtractor`. However, if you want full access to the properties, please read on!

The first thing you should understand is that a Microsoft Office file is not one large bunch of bytes but has an internal filesystem structure with files and directories. You can access these files and directories using the POI filesystem (POIFS) provides. A file or document in a POI filesystem is also called a **stream** - The properties of, say, an Excel document are stored apart of the actual spreadsheet data in separate streams. The good new is that this separation makes the properties independent of the concrete Microsoft Office file. In the following text we will always say "POI filesystem" instead of "Microsoft Office file" because a POI filesystem is not necessarily created by or for a Microsoft Office application, because it is shorter, and because we want to avoid the name of That Redmond Company.

The following example shows how to read the "title" property. Reading other properties is similar. Consider the API documentation of the class `org.apache.poi.hpsf.SummaryInformation` to learn which methods are available.

The standard properties this section focusses on can be found in a document called *\005SummaryInformation* located in the root of the POI filesystem. The notation *\005* in the document's name means the character with a decimal value of 5. In order to read the "title" property, an application has to perform the following steps:

1. Open the document *\005SummaryInformation* located in the root of the POI filesystem.
2. Create an instance of the class `SummaryInformation` from that document.
3. Call the `SummaryInformation` instance's `getTitle()` method.

Sounds easy, doesn't it? Here are the steps in detail.

1.1.1. Open the document \005SummaryInformation in the root of the POI filesystem

An application that wants to open a document in a POI filesystem (POIFS) proceeds as shown by the following code fragment. The full source code of the sample application is available in the *examples* section of the POI source tree as *ReadTitle.java*.

```
import java.io.*;
import org.apache.poi.hpsf.*;
import org.apache.poi.poifs.eventfilesystem.*;

// ...

public static void main(String[] args)
    throws IOException
```

```
{
    final String filename = args[0];
    POIFSReader r = new POIFSReader();
    r.registerListener(new MyPOIFSReaderListener(),
                       "\005SummaryInformation");
    r.read(new FileInputStream(filename));
}
```

The first interesting statement is

```
POIFSReader r = new POIFSReader();
```

It creates a `org.apache.poi.poifs.eventfilesystem.POIFSReader` instance which we shall need to read the POI filesystem. Before the application actually opens the POI filesystem we have to tell the `POIFSReader` which documents we are interested in. In this case the application should do something with the document *\005SummaryInformation*.

```
r.registerListener(new MyPOIFSReaderListener(),
                   "\005SummaryInformation");
```

This method call registers a `org.apache.poi.poifs.eventfilesystem.POIFSReaderListener` with the `POIFSReader`. The `POIFSReaderListener` interface specifies the method `processPOIFSReaderEvent()` which processes a document. The class `MyPOIFSReaderListener` implements the `POIFSReaderListener` and thus the `processPOIFSReaderEvent()` method. The eventing POI filesystem calls this method when it finds the *\005SummaryInformation* document. In the sample application `MyPOIFSReaderListener` is a static class in the *ReadTitle.java* source file.

Now everything is prepared and reading the POI filesystem can start:

```
r.read(new FileInputStream(filename));
```

The following source code fragment shows the `MyPOIFSReaderListener` class and how it retrieves the title.

```
static class MyPOIFSReaderListener implements POIFSReaderListener
{
    public void processPOIFSReaderEvent(POIFSReaderEvent event)
    {
        SummaryInformation si = null;
        try
        {
            si = (SummaryInformation)
                 PropertySetFactory.create(event.getStream());
        }
        catch (Exception ex)
        {
            throw new RuntimeException
                ("Property set stream \"" +
                 event.getPath() + event.getName() + "\": " + ex);
        }
```

```
        final String title = si.getTitle();
        if (title != null)
            System.out.println("Title: \"" + title + "\"");
        else
            System.out.println("Document has no title.");
    }
}
```

The line

```
SummaryInformation si = null;
```

declares a `SummaryInformation` variable and initializes it with `null`. We need an instance of this class to access the title. The instance is created in a `try` block:

```
si = (SummaryInformation)
                PropertySetFactory.create(event.getStream());
```

The expression `event.getStream()` returns the input stream containing the bytes of the property set stream named *\005SummaryInformation*. This stream is passed into the `create` method of the factory class `org.apache.poi.hpsf.PropertySetFactory` which returns a `org.apache.poi.hpsf.PropertySet` instance. It is more or less safe to cast this result to `SummaryInformation`, a convenience class with methods like `getTitle()`, `getAuthor()` etc.

The `PropertySetFactory.create()` method may throw all sorts of exceptions. We'll deal with them in the next sections. For now we just catch all exceptions and throw a `RuntimeException` containing the message text of the origin exception.

If all goes well, the sample application retrieves the title and prints it to the standard output. As you can see you must be prepared for the case that the POI filesystem does not have a title.

```
final String title = si.getTitle();
if (title != null)
    System.out.println("Title: \"" + title + "\"");
else
    System.out.println("Document has no title.");
```

Please note that a POI filesystem does not necessarily contain the *\005SummaryInformation* stream. The documents created by the Microsoft Office suite have one, as far as I know. However, an Excel spreadsheet exported from StarOffice 5.2 won't have a *\005SummaryInformation* stream. In this case the applications won't throw an exception but simply does not call the `processPOIFSReaderEvent` method. You have been warned!

1.2. Additional Standard Properties, Exceptions And Embedded Objects

Note:

This section focusses on reading additional standard properties which are kept in the **document summary information** stream. It also talks about exceptions that may be thrown when dealing with HPSF and shows how you can read properties of

embedded objects.

A couple of **additional standard properties** are not contained in the *\005SummaryInformation* stream explained above. Examples for such properties are a document's category or the number of multimedia clips in a PowerPoint presentation. Microsoft has invented an additional stream named *\005DocumentSummaryInformation* to hold these properties. With two minor exceptions you can proceed exactly as described above to read the properties stored in *\005DocumentSummaryInformation*:

- Instead of *\005SummaryInformation* use *\005DocumentSummaryInformation* as the stream's name.
- Replace all occurrences of the class `SummaryInformation` by `DocumentSummaryInformation`.

And of course you cannot call `getTitle()` because `DocumentSummaryInformation` has different query methods, e.g. `getCategory`. See the Javadoc API documentation for the details.

In the previous section the application simply caught all **exceptions** and was in no way interested in any details. However, a real application will likely want to know what went wrong and act appropriately. Besides any I/O exceptions there are three HPSF resp. POI specific exceptions you should know about:

`NoPropertySetStreamException`:
This exception is thrown if the application tries to create a `PropertySet` instance from a stream that is not a property set stream. (`SummaryInformation` and `DocumentSummaryInformation` are subclasses of `PropertySet`.) A faulty property set stream counts as not being a property set stream at all. An application should be prepared to deal with this case even if it opens streams named *\005SummaryInformation* or *\005DocumentSummaryInformation*. These are just names. A stream's name by itself does not ensure that the stream contains the expected contents and that this contents is correct.

`UnexpectedPropertySetTypeException`
This exception is thrown if a certain type of property set is expected somewhere (e.g. a `SummaryInformation` or `DocumentSummaryInformation`) but the provided property set is not of that type.

`MarkUnsupportedException`
This exception is thrown if an input stream that is to be parsed into a property set does not support the `InputStream.mark(int)` operation. The POI filesystem uses the `DocumentInputStream` class which does support this operation, so you are safe here. However, if you read a property set stream from another kind

of input stream things may be different.

Many Microsoft Office documents contain **embedded objects**, for example an Excel sheet within a Word document. Embedded objects may have property sets of their own. An application can open these property set streams as described above. The only difference is that they are not located in the POI filesystem's root but in a **nested directory** instead. Just register a `POIFSReaderListener` for the property set streams you are interested in. For example, the *POIBrowser* application in the contrib section tries to open each and every document in a POI filesystem as a property set stream. If this operation was successful it displays the properties.

1.3. Writing Standard Properties

Note:

This section explains how to **write standard properties**. HPSF provides some high-level classes and methods which make writing of standard properties easy. They are based on the low-level writing functions explained in another section.

As explained above, standard properties are located in the summary information and document summary information streams of typical POI filesystems. You have already learned about the classes `SummaryInformation` and `DocumentSummaryInformation` and their `get...()` methods for reading standard properties. These classes also provide `set...()` methods for writing properties.

After setting properties in `SummaryInformation` or `DocumentSummaryInformation` you have to write them to a disk file. The following sample program shows how you can

1. read a disk file into a POI filesystem,
2. read the document summary information from the POI filesystem,
3. set a property to a new value,
4. write the modified document summary information back to the POI filesystem, and
5. write the POI filesystem to a disk file.

The complete source code of this program is available as *ModifyDocumentSummaryInformation.java* in the *examples* section of the POI source tree.

Note:

Dealing with the summary information stream is analogous to handling the document summary information and therefore does not need to be explained here in detailed. See the HPSF API documentation to learn about the `set...()` methods of the class `SummaryInformation`.

The first step is to read the POI filesystem into memory:

```
InputStream is = new FileInputStream(poiFilesystem);
```

```
POIFSFileSystem poifs = new POIFSFileSystem(is);
is.close();
```

The code snippet above assumes that the variable `poiFilesystem` holds the name of a disk file. It reads the file from an input stream and creates a `POIFSFileSystem` object in memory. After having read the file, the input stream should be closed as shown.

In order to read the document summary information stream the application must open the element *\005DocumentSummaryInformation* in the POI filesystem's root directory. However, the POI filesystem does not necessarily contain a document summary information stream, and the application should be able to deal with that situation. The following code does so by creating a new `DocumentSummaryInformation` if there is none in the POI filesystem:

```
DirectoryEntry dir = poifs.getRoot();
DocumentSummaryInformation dsi;
try
{
    DocumentEntry dsiEntry = (DocumentEntry)
        dir.getEntry(DocumentSummaryInformation.DEFAULT_STREAM_NAME);
    DocumentInputStream dis = new DocumentInputStream(dsiEntry);
    PropertySet ps = new PropertySet(dis);
    dis.close();
    dsi = new DocumentSummaryInformation(ps);
}
catch (FileNotFoundException ex)
{
    /* There is no document summary information. We have to create a
     * new one. */
    dsi = PropertySetFactory.newDocumentSummaryInformation();
}
```

In the source code above the statement

```
DirectoryEntry dir = poifs.getRoot();
```

gets hold of the POI filesystem's root directory as a `DirectoryEntry`. The `getEntry()` method of this class is used to access a file or directory entry in a directory. However, if the file to be opened does not exist, a `FileNotFoundException` will be thrown. Therefore opening the document summary information entry should be done in a `try` block:

```
    DocumentEntry dsiEntry = (DocumentEntry)
        dir.getEntry(DocumentSummaryInformation.DEFAULT_STREAM_NAME);
```

`DocumentSummaryInformation.DEFAULT_STREAM_NAME` represents the string "\005DocumentSummaryInformation", i.e. the standard name of a document summary information stream. If this stream exists, the `getEntry()` method returns a `DocumentEntry`. To read the `DocumentEntry`'s contents, create a `DocumentInputStream`:

```
     DocumentInputStream dis = new DocumentInputStream(dsiEntry);
```

Up to this point we have used POI's POIFS component. Now HPSF enters the stage. A

property set is created from the input stream's data:

```
    PropertySet ps = new PropertySet(dis);
    dis.close();
    dsi = new DocumentSummaryInformation(ps);
```

If the data really constitutes a property set, a `PropertySet` object is created. Otherwise a `NoPropertySetStreamException` is thrown. After having read the data from the input stream the latter should be closed.

Since we know - or at least hope - that the stream named "\005DocumentSummaryInformation" is not just any property set but really contains the document summary information, we try to create a new `DocumentSummaryInformation` from the property set. If the stream is not document summary information stream the sample application fails with a `UnexpectedPropertySetTypeException`.

If the POI document does not contain a document summary information stream, we can create a new one in the `catch` clause. The `PropertySetFactory`'s method `newDocumentSummaryInformation()` establishes a new and empty `DocumentSummaryInformation` instance:

```
    dsi = PropertySetFactory.newDocumentSummaryInformation();
```

Whether we read the document summary information from the POI filesystem or created it from scratch, in either case we now have a `DocumentSummaryInformation` instance we can write to. Writing is quite simple, as the following line of code shows:

```
dsi.setCategory("POI example");
```

This statement sets the "category" property to "POI example". Any former "category" value will be lost. If there hasn't been a "category" property yet, a new one will be created.

`DocumentSummaryInformation` of course has methods to set the other standard properties, too - look into the API documentation to see all of them.

Once all properties are set as needed, they should be stored into the file on disk. The first step is to write the `DocumentSummaryInformation` into the POI filesystem:

```
dsi.write(dir, DocumentSummaryInformation.DEFAULT_STREAM_NAME);
```

The `DocumentSummaryInformation`'s `write()` method takes two parameters: The first is the `DirectoryEntry` in the POI filesystem, the second is the name of the stream to create in the directory. If this stream already exists, it will be overwritten.

Note:

If you not only modified the document summary information but also the summary information you have to write both of them to the POI filesystem.

Still the POI filesystem is a data structure in memory only and must be written to a disk file to make it permanent. The following lines write back the POI filesystem to the file it was read from before. Please note that in production-quality code you should never write directly to the origin file, because in case of an error everything would be lost. Here it is done this way to keep the example short.

```
OutputStream out = new FileOutputStream(poiFilesystem);
poifs.writeFilesystem(out);
out.close();
```

1.3.1. User-Defined Properties

If you compare the source code excerpts above with the file containing the full source code, you will notice that I left out some following lines of code. The are dealing with the special topic of custom properties.

```
DocumentSummaryInformation dsi = ...
...
CustomProperties customProperties = dsi.getCustomProperties();
if (customProperties == null)
    customProperties = new CustomProperties();

/* Insert some custom properties into the container. */
customProperties.put("Key 1", "Value 1");
customProperties.put("Schlüssel 2", "Wert 2");
customProperties.put("Sample Number", new Integer(12345));
customProperties.put("Sample Boolean", new Boolean(true));
customProperties.put("Sample Date", new Date());

/* Read a custom property. */
Object value = customProperties.get("Sample Number");

/* Write the custom properties back to the document summary
 * information. */
dsi.setCustomProperties(customProperties);
```

Custom properties are properties the user can define himself. Using for example Microsoft Word he can define these extra properties and give each of them a **name**, a **type** and a **value**. The custom properties are stored in the document information summary along with the standard properties.

The source code example shows how to retrieve the custom properties as a whole from a `DocumentSummaryInformation` instance using the `getCustomProperties()` method. The result is a `CustomProperties` instance or `null` if no user-defined properties exist.

Since `CustomProperties` implements the `Map` interface you can read and write properties with the usual `Map` methods. However, `CustomProperties` poses some restrictions on the types of keys and values.

- The **key** is a string.
- The **value** is one of `String`, `Boolean`, `Long`, `Integer`, `Short`, or `java.util.Date`.

The `CustomProperties` class has been designed for easy access using just keys and values. The underlying Microsoft-specific custom properties data structure is more complicated. However, it does not provide noteworthy additional benefits. It is possible to have multiple properties with the same name or properties without a name at all. When reading custom properties from a document summary information stream, the `CustomProperties` class ignores properties without a name and keeps only the "last" (whatever that means) of those properties having the same name. You can find out whether a `CustomProperties` instance dropped any properties with the `isPure()` method.

You can read and write the full spectrum of custom properties with HPSF's low-level methods. They are explained in the next section.

1.4. Reading Non-Standard Properties

Note:
This section tells how to read non-standard properties. Non-standard properties are application-specific ID/type/value triples.

1.4.1. Overview

Now comes the real hardcode stuff. As mentioned above, `SummaryInformation` and `DocumentSummaryInformation` are just special cases of the general concept of a property set. This concept says that a **property set** consists of properties and that each **property** is an entity with an **ID**, a **type**, and a **value**.

Okay, that was still rather easy. However, to make things more complicated, Microsoft in its infinite wisdom decided that a property set shalt be broken into one or more **sections**. Each section holds a bunch of properties. But since that's still not complicated enough, a section may have an optional **dictionary** that maps property IDs to **property names** - we'll explain later what that means.

The procedure to get to the properties is the following:

1. Use the **`PropertySetFactory`** class to create a `PropertySet` object from a property set stream. If you don't know whether an input stream is a property set stream, just try to call `PropertySetFactory.create(java.io.InputStream)`: You'll either get a `PropertySet` instance returned or an exception is thrown.
2. Call the `PropertySet`'s method `getSections()` to get the sections contained in the property set. Each section is an instance of the `Section` class.

3. Each section has a format ID. The format ID of the first section in a property set determines the property set's type. For example, the first (and only) section of the summary information property set has a format ID of `F29F85E0-4FF9-1068-AB-91-08-00-2B-27-B3-D9`. You can get the format ID with `Section.getFormatID()`.
4. The properties contained in a `Section` can be retrieved with `Section.getProperties()`. The result is an array of `Property` instances.
5. A property has a name, a type, and a value. The `Property` class has methods to retrieve them.

1.4.2. A Sample Application

Let's have a look at a sample Java application that dumps all property set streams contained in a POI file system. The full source code of this program can be found as *ReadCustomPropertySets.java* in the *examples* area of the POI source code tree. Here are the key sections:

```
import java.io.*;
import java.util.*;
import org.apache.poi.hpsf.*;
import org.apache.poi.poifs.eventfilesystem.*;
import org.apache.poi.util.HexDump;
```

The most important package the application needs is `org.apache.poi.hpsf.*`. This package contains the HPSF classes. Most classes named below are from the HPSF package. Of course we also need the POIFS event file system's classes and `java.io.*` since we are dealing with POI I/O. From the `java.util` package we use the `List` and `Iterator` class. The class `org.apache.poi.util.HexDump` provides a methods to dump byte arrays as nicely formatted strings.

```
public static void main(String[] args)
    throws IOException
{
    final String filename = args[0];
    POIFSReader r = new POIFSReader();

    /* Register a listener for *all* documents. */
    r.registerListener(new MyPOIFSReaderListener());
    r.read(new FileInputStream(filename));
}
```

The `POIFSReader` is set up in a way that the listener `MyPOIFSReaderListener` is called on every file in the POI file system.

1.4.3. The Property Set

The listener class tries to create a `PropertySet` from each stream using the `PropertySetFactory.create()` method:

```
static class MyPOIFSReaderListener implements POIFSReaderListener
{
    public void processPOIFSReaderEvent(POIFSReaderEvent event)
    {
        PropertySet ps = null;
        try
        {
            ps = PropertySetFactory.create(event.getStream());
        }
        catch (NoPropertySetStreamException ex)
        {
            out("No property set stream: \"" + event.getPath() +
                event.getName() + "\"");
            return;
        }
        catch (Exception ex)
        {
            throw new RuntimeException
                ("Property set stream \"" +
                 event.getPath() + event.getName() + "\": " + ex);
        }

        /* Print the name of the property set stream: */
        out("Property set stream \"" + event.getPath() +
            event.getName() + "\":");
```

Creating the `PropertySet` is done in a `try` block, because not each stream in the POI file system contains a property set. If it is some other file, the `PropertySetFactory.create()` throws a `NoPropertySetStreamException`, which is caught and logged. Then the program continues with the next stream. However, all other types of exceptions cause the program to terminate by throwing a runtime exception. If all went well, we can print the name of the property set stream.

1.4.4. The Sections

The next step is to print the number of sections followed by the sections themselves:

```
/* Print the number of sections: */
final long sectionCount = ps.getSectionCount();
out("   No. of sections: " + sectionCount);

/* Print the list of sections: */
List sections = ps.getSections();
int nr = 0;
for (Iterator i = sections.iterator(); i.hasNext();)
{
    /* Print a single section: */
    Section sec = (Section) i.next();

    // See below for the complete loop body.
}
```

The `PropertySet`'s method `getSectionCount()` returns the number of sections.

To retrieve the sections, use the `getSections()` method. This method returns a `java.util.List` containing instances of the `Section` class in their proper order.

The sample code shows a loop that retrieves the `Section` objects one by one and prints some information about each one. Here is the complete body of the loop:

```
/* Print a single section: */
Section sec = (Section) i.next();
out("   Section " + nr++ + ":");
String s = hex(sec.getFormatID().getBytes());
s = s.substring(0, s.length() - 1);
out("      Format ID: " + s);

/* Print the number of properties in this section. */
int propertyCount = sec.getPropertyCount();
out("      No. of properties: " + propertyCount);

/* Print the properties: */
Property[] properties = sec.getProperties();
for (int i2 = 0; i2 < properties.length; i2++)
{
    /* Print a single property: */
    Property p = properties[i2];
    int id = p.getID();
    long type = p.getType();
    Object value = p.getValue();
    out("      Property ID: " + id + ", type: " + type +
        ", value: " + value);
}
```

1.4.5. The Section's Format ID

The first method called on the `Section` instance is `getFormatID()`. As explained above, the format ID of the first section in a property set determines the type of the property set. Its type is `ClassID` which is essentially a sequence of 16 bytes. A real application using its own type of a custom property set should have defined a unique format ID and, when reading a property set stream, should check the format ID is equal to that unique format ID. The sample program just prints the format ID it finds in a section:

```
String s = hex(sec.getFormatID().getBytes());
s = s.substring(0, s.length() - 1);
out("      Format ID: " + s);
```

As you can see, the `getFormatID()` method returns a `ClassID` object. An array containing the bytes can be retrieved with `ClassID.getBytes()`. In order to get a nicely formatted printout, the sample program uses the `hex()` helper method which in turn uses the POI utility class `HexDump` in the `org.apache.poi.util` package. Another helper method is `out()` which just saves typing `System.out.println()`.

1.4.6. The Properties

Before getting the properties, it is possible to find out how many properties are available in the section via the `Section.getPropertyCount()`. The sample application uses this method to print the number of properties to the standard output:

```
int propertyCount = sec.getPropertyCount();
out("      No. of properties: " + propertyCount);
```

Now its time to get to the properties themselves. You can retrieve a section's properties with the method `Section.getProperties()`:

```
Property[] properties = sec.getProperties();
```

As you can see the result is an array of `Property` objects. This class has three methods to retrieve a property's ID, its type, and its value. The following code snippet shows how to call them:

```
for (int i2 = 0; i2 < properties.length; i2++)
{
    /* Print a single property: */
    Property p = properties[i2];
    int id = p.getID();
    long type = p.getType();
    Object value = p.getValue();
    out("      Property ID: " + id + ", type: " + type +
        ", value: " + value);
}
```

1.4.7. Sample Output

The output of the sample program might look like the following. It shows the summary information and the document summary information property sets of a Microsoft Word document. However, unlike the first and second section of this HOW-TO the application does not have any code which is specific to the `SummaryInformation` and `DocumentSummaryInformation` classes.

```
Property set stream "/SummaryInformation":
   No. of sections: 1
   Section 0:
      Format ID: 00000000 F2 9F 85 E0 4F F9 10 68 AB 91 08 00 2B 27 B3 D9 ....O..h....+
      No. of properties: 17
      Property ID: 1, type: 2, value: 1252
      Property ID: 2, type: 30, value: Titel
      Property ID: 3, type: 30, value: Thema
      Property ID: 4, type: 30, value: Rainer Klute (Autor)
      Property ID: 5, type: 30, value: Test (Stichwörter)
      Property ID: 6, type: 30, value: This is a document for testing HPSF
      Property ID: 7, type: 30, value: Normal.dot
      Property ID: 8, type: 30, value: Unknown User
      Property ID: 9, type: 30, value: 3
      Property ID: 18, type: 30, value: Microsoft Word 9.0
```

```
      Property ID: 12, type: 64, value: Mon Jan 01 00:59:25 CET 1601
      Property ID: 13, type: 64, value: Thu Jul 18 16:22:00 CEST 2002
      Property ID: 14, type: 3, value: 1
      Property ID: 15, type: 3, value: 20
      Property ID: 16, type: 3, value: 93
      Property ID: 19, type: 3, value: 0
      Property ID: 17, type: 71, value: [B@13582d
Property set stream "/DocumentSummaryInformation":
   No. of sections: 2
   Section 0:
      Format ID: 00000000 D5 CD D5 02 2E 9C 10 1B 93 97 08 00 2B 2C F9 AE ............+
      No. of properties: 14
      Property ID: 1, type: 2, value: 1252
      Property ID: 2, type: 30, value: Test
      Property ID: 14, type: 30, value: Rainer Klute (Manager)
      Property ID: 15, type: 30, value: Rainer Klute IT-Consulting GmbH
      Property ID: 5, type: 3, value: 3
      Property ID: 6, type: 3, value: 2
      Property ID: 17, type: 3, value: 111
      Property ID: 23, type: 3, value: 592636
      Property ID: 11, type: 11, value: false
      Property ID: 16, type: 11, value: false
      Property ID: 19, type: 11, value: false
      Property ID: 22, type: 11, value: false
      Property ID: 13, type: 4126, value: [B@56a499
      Property ID: 12, type: 4108, value: [B@506411
   Section 1:
      Format ID: 00000000 D5 CD D5 05 2E 9C 10 1B 93 97 08 00 2B 2C F9 AE ............+
      No. of properties: 7
      Property ID: 0, type: 0, value: {6=Test-JaNein, 5=Test-Zahl, 4=Test-Datum, 3=Test
      Property ID: 1, type: 2, value: 1252
      Property ID: 2, type: 65, value: [B@c9ba38
      Property ID: 3, type: 30, value: This is some text.
      Property ID: 4, type: 64, value: Wed Jul 17 00:00:00 CEST 2002
      Property ID: 5, type: 3, value: 27
      Property ID: 6, type: 11, value: true
No property set stream: "/WordDocument"
No property set stream: "/CompObj"
No property set stream: "/1Table"
```

There are some interesting items to note:

- The first property set (summary information) consists of a single section, the second property set (document summary information) consists of two sections.
- Each section type (identified by its format ID) has its own domain of property ID. For example, in the second property set the properties with ID 2 have different meanings in the two section. By the way, the format IDs of these sections are **not** equal, but you have to look hard to find the difference.
- The properties are not in any particular order in the section, although they slightly tend to be sorted by their IDs.

1.4.8. Property IDs

Properties in the same section are distinguished by their IDs. This is similar to variables in a programming language like Java, which are distinguished by their names. But unlike variable names, property IDs are simple integral numbers. There is another similarity, however. Just like a Java variable has a certain scope (e.g. a member variables in a class), a property ID also has its scope of validity: the section.

Two property IDs in sections with different section format IDs don't have the same meaning even though their IDs might be equal. For example, ID 4 in the first (and only) section of a summary information property set denotes the document's author, while ID 4 in the first section of the document summary information property set means the document's byte count. The sample output above does not show a property with an ID of 4 in the first section of the document summary information property set. That means that the document does not have a byte count. However, there is a property with an ID of 4 in the *second* section: This is a user-defined property ID - we'll get to that topic in a minute.

So, how can you find out what the meaning of a certain property ID in the summary information and the document summary information property set is? The standard property sets as such don't have any hints about the **meanings of their property IDs**. For example, the summary information property set does not tell you that the property ID 4 stands for the document's author. This is external knowledge. Microsoft defined standard meanings for some of the property IDs in the summary information and the document summary information property sets. As a help to the Java and POI programmer, the class `PropertyIDMap` in the `org.apache.poi.hpsf.wellknown` package defines constants for the "well-known" property IDs. For example, there is the definition

```
public final static int PID_AUTHOR = 4;
```

These definitions allow you to use symbolic names instead of numbers.

In order to provide support for the other way, too, - i.e. to map property IDs to property names - the class `PropertyIDMap` defines two static methods: `getSummaryInformationProperties()` and `getDocumentSummaryInformationProperties()`. Both return `java.util.Map` objects which map property IDs to strings. Such a string gives a hint about the property's meaning. For example, `PropertyIDMap.getSummaryInformationProperties().get(4)` returns the string "PID_AUTHOR". An application could use this string as a key to a localized string which is displayed to the user, e.g. "Author" in English or "Verfasser" in German. HPSF might provide such language-dependend ("localized") mappings in a later release.

Usually you won't have to deal with those two maps. Instead you should call the `Section.getPIDString(int)` method. It returns the string associated with the specified property ID in the context of the `Section` object.

Above you learned that property IDs have a meaning in the scope of a section only. However, there are two exceptions to the rule: The property IDs 0 and 1 have a fixed meaning in **all** sections:

Property ID	Meaning
0	The property's value is a **dictionary**, i.e. a mapping from property IDs to strings.
1	The property's value is the number of a **codepage**, i.e. a mapping from character codes to characters. All strings in the section containing this property must be interpreted using this codepage. Typical property values are 1252 (8-bit "western" characters, ISO-8859-1), 1200 (16-bit Unicode characters, UFT-16), or 65001 (8-bit Unicode characters, UFT-8).

1.4.9. Property types

A property is nothing without its value. It is stored in a property set stream as a sequence of bytes. You must know the property's **type** in order to properly interpret those bytes and reasonably handle the value. A property's type is one of the so-called Microsoft-defined "**variant types**". When you call `Property.getType()` you'll get a `long` value which denoting the property's variant type. The class `Variant` in the `org.apache.poi.hpsf` package holds most of those `long` values as named constants. For example, the constant `VT_I4 = 3` means a signed integer value of four bytes. Examples of other types are `VT_LPSTR = 30` meaning a null-terminated string of 8-bit characters, `VT_LPWSTR = 31` which means a null-terminated Unicode string, or `VT_BOOL = 11` denoting a boolean value.

In most cases you won't need a property's type because HPSF does all the work for you.

1.4.10. Property values

When an application wants to retrieve a property's value and calls `Property.getValue()`, HPSF has to interpret the bytes making out the value according to the property's type. The type determines how many bytes the value consists of and what to do with them. For example, if the type is `VT_I4`, HPSF knows that the value is four bytes long and that these bytes comprise a signed integer value in the little-endian format. This is quite different from e.g. a type of `VT_LPWSTR`. In this case HPSF has to scan the value bytes for a Unicode null character and collect everything from the beginning to that null character as a Unicode string.

The good new is that HPSF does another job for you, too: It maps the variant type to an adequate Java type.

Variant type:	Java type:
VT_I2	java.lang.Integer
VT_I4	java.lang.Long
VT_FILETIME	java.util.Date
VT_LPSTR	java.lang.String
VT_LPWSTR	java.lang.String
VT_CF	byte[]
VT_BOOL	java.lang.Boolean

The bad news is that there are still a couple of variant types HPSF does not yet support. If it encounters one of these types it returns the property's value as a byte array and leaves it to be interpreted by the application.

An application retrieves a property's value by calling the `Property.getValue()` method. This method's return type is the abstract `Object` class. The `getValue()` method looks up the property's variant type, reads the property's value bytes, creates an instance of an adequate Java type, assigns it the property's value and returns it. Primitive types like `int` or `long` will be returned as the corresponding class, e.g. `Integer` or `Long`.

1.4.11. Dictionaries

The property with ID 0 has a very special meaning: It is a **dictionary** mapping property IDs to property names. We have seen already that the meanings of standard properties in the summary information and the document summary information property sets have been defined by Microsoft. The advantage is that the labels of properties like "Author" or "Title" don't have to be stored in the property set. However, a user can define custom fields in, say, Microsoft Word. For each field the user has to specify a name, a type, and a value.

The names of the custom-defined fields (i.e. the property names) are stored in the document summary information second section's **dictionary**. The dictionary is a map which associates property IDs with property names.

The method `Section.getPIDString(int)` not only returns with the well-known property names of the summary information and document summary information property sets, but with self-defined properties, too. It should also work with self-defined properties in self-defined sections.

1.4.12. Codepage support

The property with ID 1 holds the number of the codepage which was used to encode the strings in this section. If this property is not available in a section, the platform's default character encoding will be used. This works fine as long as the document being read has been written on a platform with the same default character encoding. However, if you receive a document from another region of the world and the codepage is undefined, you are in trouble.

HPSF's codepage support is only as good as the character encoding support of the Java Virtual Machine (JVM) the application runs on. If HPSF encounters a codepage number it assumes that the JVM has a character encoding with a corresponding name. For example, if the codepage is 1252, HPSF uses the character encoding "cp1252" to read or write strings. If the JVM does not have that character encoding installed or if the codepage number is illegal, an UnsupportedEncodingException will be thrown. This works quite well with Java 2 Standard Edition (J2SE) versions since 1.4. However, under J2SE 1.3 or lower you are out of luck. You should install a newer J2SE version to process codepages with HPSF.

There are some exceptions to the rule saying that a character encoding's name is derived from the codepage number by prepending the string "cp" to it. In these cases the codepage number is mapped to a well-known character encoding name. Here are a few examples:

Codepage 932
is mapped to the character encoding "SJIS".
Codepage 1200
is mapped to the character encoding "UTF-16".
Codepage 65001
is mapped to the character encoding "UTF-8".

More of these mappings between codepage and character encoding name are hard-coded in the classes `org.apache.poi.hpsf.Constants` and `org.apache.poi.hpsf.VariantSupport`. Probably there will be a need to add more mappings. The HPSF author will appreciate any hints.

1.5. Writing Properties

Note:
This section describes how to write properties.

1.5.1. Overview of Writing Properties

Writing properties is possible at a high level and at a low level:

- Most users will want to create or change entries in the summary information or document summary information streams.
- On the low level, there are no convenience classes or methods. You have to deal with things like property IDs and variant types to write properties. Therefore you should have read section 3 to understand the description of the low-level writing functions.

HPSF's writing capabilities come with the classes `MutablePropertySet`, `MutableSection`, `MutableProperty`, and some helper classes. The "mutable" classes extend their respective superclasses `PropertySet`, `Section`, and `Property` and provide "set" and "write" methods, following the Decorator pattern.

1.5.2. Low-Level Writing: An Overview

When you are going to write a property set stream your application has to perform the following steps:

1. Create a `MutablePropertySet` instance.
2. Get hold of a `MutableSection`. You can either retrieve the one that is always present in a new `MutablePropertySet`, or you have to create a new `MutableSection` and add it to the `MutablePropertySet`.
3. Set any `Section` fields as you like.
4. Create as many `MutableProperty` objects as you need. Set each property's ID, type, and value. Add the `MutableProperty` objects to the `MutableSection`.
5. Create further `MutableSection`s if you need them.
6. Eventually retrieve the property set as a byte stream using `MutablePropertySet.toInputStream()` and write it to a POIFS document.

1.5.3. Low-level Writing Functions In Details

Writing properties is introduced by an artificial but simple example: a program creating a new document (aka POI file system) which contains only a single document: a summary information property set stream. The latter will hold the document's title only. This is artificial in that it does not contain any Word, Excel or other kind of useful application document data. A document containing just a property set is without any practical use. However, it is perfectly fine for an example because it make it very simple and easy to understand, and you will get used to writing properties in real applications quickly.

The application expects the name of the POI file system to be written on the command line. The title property it writes is "Sample title".

Here's the application's source code. You can also find it in the "examples" section of the POI source code distribution. Explanations are following below.

```
package org.apache.poi.hpsf.examples;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

import org.apache.poi.hpsf.MutableProperty;
import org.apache.poi.hpsf.MutablePropertySet;
import org.apache.poi.hpsf.MutableSection;
import org.apache.poi.hpsf.SummaryInformation;
import org.apache.poi.hpsf.Variant;
import org.apache.poi.hpsf.WritingNotSupportedException;
import org.apache.poi.hpsf.wellknown.PropertyIDMap;
import org.apache.poi.hpsf.wellknown.SectionIDMap;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

/**
 * <p>This class is a simple sample application showing how to create a property
 * set and write it to disk.</p>
 *
 * @author Rainer Klute
 * @since 2003-09-12
 */
public class WriteTitle
{
    /**
     * <p>Runs the example program.</p>
     *
     * @param args Command-line arguments. The first and only command-line
     * argument is the name of the POI file system to create.
     * @throws IOException if any I/O exception occurs.
     * @throws WritingNotSupportedException if HPSF does not (yet) support
     * writing a certain property type.
     */
    public static void main(final String[] args)
    throws WritingNotSupportedException, IOException
    {
        /* Check whether we have exactly one command-line argument. */
        if (args.length != 1)
        {
            System.err.println("Usage: " + WriteTitle.class.getName() +
                               "destinationPOIFS");
            System.exit(1);
        }

        final String fileName = args[0];

        /* Create a mutable property set. Initially it contains a single section
         * with no properties. */
        final MutablePropertySet mps = new MutablePropertySet();

        /* Retrieve the section the property set already contains. */
        final MutableSection ms = (MutableSection) mps.getSections().get(0);
```

```
        /* Turn the property set into a summary information property. This is
         * done by setting the format ID of its first section to
         * SectionIDMap.SUMMARY_INFORMATION_ID. */
        ms.setFormatID(SectionIDMap.SUMMARY_INFORMATION_ID);

        /* Create an empty property. */
        final MutableProperty p = new MutableProperty();

        /* Fill the property with appropriate settings so that it specifies the
         * document's title. */
        p.setID(PropertyIDMap.PID_TITLE);
        p.setType(Variant.VT_LPWSTR);
        p.setValue("Sample title");

        /* Place the property into the section. */
        ms.setProperty(p);

        /* Create the POI file system the property set is to be written to. */
        final POIFSFileSystem poiFs = new POIFSFileSystem();

        /* For writing the property set into a POI file system it has to be
         * handed over to the POIFS.createDocument() method as an input stream
         * which produces the bytes making out the property set stream. */
        final InputStream is = mps.toInputStream();

        /* Create the summary information property set in the POI file
         * system. It is given the default name most (if not all) summary
         * information property sets have. */
        poiFs.createDocument(is, SummaryInformation.DEFAULT_STREAM_NAME);

        /* Write the whole POI file system to a disk file. */
        poiFs.writeFilesystem(new FileOutputStream(fileName));
    }

}
```

The application first checks that there is exactly one single argument on the command line: the name of the file to write. If this single argument is present, the application stores it in the `fileName` variable. It will be used in the end when the POI file system is written to a disk file.

```
if (args.length != 1)
{
    System.err.println("Usage: " + WriteTitle.class.getName() +
                       "destinationPOIFS");
    System.exit(1);
}
final String fileName = args[0];
```

Let's create a property set now. We cannot use the `PropertySet` class, because it is read-only. It does not have a constructor creating an empty property set, and it does not have any methods to modify its contents, i.e. to write sections containing properties into it.

The class to use is `MutablePropertySet`. It is a subclass of `PropertySet`. The sample application calls its no-args constructor in order to establish an empty property set:

```
final MutablePropertySet mps = new MutablePropertySet();
```

As said, we have an empty property set now. Later we will put some contents into it.

By the way, the `MutablePropertySet` class has another constructor taking a `PropertySet` as parameter. It creates a mutable deep copy of the property set given to it.

The `MutablePropertySet` created by the no-args constructor is not really empty: It contains a single section without properties. We can either retrieve that section and fill it with properties or we can replace it by another section. We can also add further sections to the property set. The sample application decides to retrieve the section being already there:

```
final MutableSection ms = (MutableSection) mps.getSections().get(0);
```

The `getSections()` method returns the property set's sections as a list, i.e. an instance of `java.util.List`. Calling `get(0)` returns the list's first (or zeroth, if you prefer) element. The `Section` returned is a `MutableSection`: a subclass of `Section` you can modify.

The alternative to retrieving the `MutableSection` being already there would have been to create an new `MutableSection` like this:

```
MutableSection s = new MutableSection();
```

There is also a constructor which takes a `Section` as parameter and creates a mutable deep copy of it.

The `MutableSection` the sample application retrieved from the `MutablePropertySet` is still empty. It contains no properties and does not have a format ID. As you have read above the format ID of the first section in a property set determines the property set's type. Since our property set should become a SummaryInformation property set we have to set the format ID of its first (and only) section to `F29F85E0-4FF9-1068-AB-91-08-00-2B-27-B3-D9`. However, you won't have to remember that ID: HPSF has it defined as the well-known constant `SectionIDMap.SUMMARY_INFORMATION_ID`. The sample application writes it to the section using the `setFormatID(byte[])` method:

```
ms.setFormatID(SectionIDMap.SUMMARY_INFORMATION_ID);
```

Now it is time to create a property. As you might expect there is a subclass of `Property` called `MutableProperty` with a no-args constructor:

```
final MutableProperty p = new MutableProperty();
```

A `MutableProperty` object must have an ID, a type, and a value (see above for details). The class provides methods to set these attributes:

```
p.setID(PropertyIDMap.PID_TITLE);
```

```
p.setType(Variant.VT_LPWSTR);
p.setValue("Sample title");
```

The `MutableProperty` class has a constructor which you can use to pass in all three attributes in a single call. See the Javadoc API documentation for details!

The sample property set is complete now. We have a `MutablePropertySet` containing a `MutableSection` containing a `MutableProperty`. Of course we could have added more sections to the property set and more properties to the sections but we wanted to keep things simple.

The property set has to be written to a POI file system. The following statement creates it.

```
final POIFSFileSystem poiFs = new POIFSFileSystem();
```

Writing the property set includes the step of converting it into a sequence of bytes. The `MutablePropertySet` class has the method `toInputStream()` for this purpose. It returns the bytes making out the property set stream as an `InputStream`:

```
final InputStream is = mps.toInputStream();
```

If you'd read from this input stream you'd receive all the property set's bytes. However, it is very likely that you'll never do that. Instead you'll pass the input stream to the `POIFSFileSystem.createDocument()` method, like this:

```
poiFs.createDocument(is, SummaryInformation.DEFAULT_STREAM_NAME);
```

Besides the `InputStream` `createDocument()` takes a second parameter: the name of the document to be created. For a SummaryInformation property set stream the default name is available as the constant `SummaryInformation.DEFAULT_STREAM_NAME`.

The last step is to write the POI file system to a disk file:

```
poiFs.writeFilesystem(new FileOutputStream(fileName));
```

1.6. Further Reading

There are still some aspects of HSPF left which are not covered by this HOW-TO. You should dig into the Javadoc API documentation to learn further details. Since you've struggled through this document up to this point, you are well prepared.