

# PJX

---

Java Class Library for PDF Software Development

Nassib Nassar *Etymon Systems, Inc.*

---

This document describes PJX Version 1.4.0. Updates and information are available from the Etymon web site at <http://www.etymon.com/>.

*Copyright © 1998–2006 by Etymon Systems, Inc.*

*ETYMON is a registered trademark and service mark of Etymon Systems, Inc. All other trademarks are the property of their respective owners.*

*Etymon Systems, Inc. disclaims all warranties, either express or implied, including but not limited to implied warranties of merchantability, fitness for a particular purpose, and non-infringement of third-party rights, and all other remedies for breach of the above warranty. Etymon Systems, Inc. assumes no liability with respect to the accuracy, adequacy, quality, and reliability of this publication.*

*The PDF data structures, operators, and specification are  
Copyright © 1985–2006 by Adobe Systems Incorporated.*

# Table of Contents

<b>1</b>	<b>Tutorial .....</b>	<b>1</b>
1.1	Introduction .....	1
1.2	Reading a document .....	1
1.3	Modifying a document .....	2
1.4	Writing a document .....	3
1.5	Appending documents .....	3
1.6	Examining objects recursively .....	4



# 1 Tutorial

THIS document serves as an introduction to PJX, a Java class library for PDF software development. It is designed as a supplement to the API (`javadoc`) reference documentation for PJX. This document is intended to be used in conjunction with, and assumes some familiarity with, the PDF specification published by Adobe Systems Incorporated (*PDF Reference*, 3rd ed., ISBN 0-201-75839-3). It also assumes familiarity with the Java programming language. **This is a preliminary document and a work in progress.**

The purpose of PJX is to enable you to be a PDF programmer. It does not hide the details of PDF unless you want it to; but it tries to make them much easier and more pleasant to work with. It provides a set of fundamental tools that can be used to develop almost any PDF application. On top of these tools are layered classes that encapsulate common PDF functions. This provides access to PDF documents at multiple levels, from basic encoding/decoding up to the application layer. You can use PJX as a foundation for building PDF capabilities into existing Java software or for creating new PDF applications.

## 1.1 Introduction

This chapter gives an overview of how the PJX classes fit together in order to perform simple operations on PDF documents.

The core of PJX is the `PdfManager` class in `com.etymon.pjx` which coordinates a set of modifications to a PDF document. `PdfManager` operates on PDF documents via two other classes, `PdfReader` and `PdfWriter`. Additional utility classes in `com.etymon.pjx.util` operate on PDF documents via `PdfManager`; they serve to consolidate common PDF functions and access to related sets of PDF objects.

## 1.2 Reading a document

The `PdfReader` class provides low-level access to an existing PDF document. Its constructor accepts a class that implements the `PdfInput` interface. Two such classes are provided, `PdfInputBuffer` and `PdfInputFile`. Using `PdfInputBuffer` causes the PDF document to reside in a buffer in memory for the sake of efficiency; the document can originate in the file system or it can be already in memory. `PdfInputFile` accesses a PDF document in the file system and only reads portions of it into memory if they are needed.

Following are some examples of constructing a `PdfReader` instance:

```
PdfReader r = new PdfReader(new PdfInputBuffer(new File("test.pdf")));
ByteBuffer bb = ... ;
PdfReader r = new PdfReader(new PdfInputBuffer(bb, "Test document"));
PdfReader r = new PdfReader(new PdfInputFile(new File("test.pdf")));
```

Once the `PdfReader` instance has been constructed, we can associate a `PdfManager` instance with it:

```
PdfManager m = new PdfManager(r);
```

`PdfManager` keeps track of changes made to the document and allows the resultant document to be written to a `PdfWriter`.

It is not necessary to retain a reference to the `PdfReader` instance, although it is sometimes useful to do so. For example, if an existing document needs to be reused, such as in order to be modified in different ways by multiple instances of `PdfManager`, only one instance of `PdfReader` should be created for that document.

### 1.3 Modifying a document

`PdfManager` provides fundamental methods for accessing and modifying the set of PDF objects in a document. One possible starting point for examining a PDF document is to use `PdfManager` to access the document's trailer dictionary. The trailer dictionary contains pointers to useful objects in the document. We can get the trailer dictionary as follows:

```
PdfDictionary td = m.getTrailerDictionary();
```

PJX represents a PDF dictionary using a Java `Map` instance, which can be retrieved via the `PdfDictionary` method, `getMap()`:

```
Map tdMap = td.getMap();
```

The PDF specification tells us that the trailer dictionary contains several keys including `Info`, which is mapped to an indirect reference to the document information dictionary. We can retrieve the reference from `tdMap`:

```
PdfReference infoRef = (PdfReference)tdMap.get(new PdfName("Info"));
```

`PdfManager` provides a method to resolve an indirect reference, `getObjectIndirect(PdfObject)`:

```
PdfDictionary info = (PdfDictionary)m.getObjectIndirect(infoRef);
```

In general it is a good idea to call `getObjectIndirect(PdfObject)` on any retrieved PDF object, since virtually any PDF object can be referenced indirectly, and this method does not mind being called with direct objects, which it simply returns unchanged.

An alternative way to resolve an indirect reference is using the `PdfManager.getObject(int)` method:

```
PdfDictionary info = (PdfDictionary)m.getObject(infoRef.getObjectNumber());
```

However, `getObjectIndirect(PdfObject)` is normally more effective because it follows multiple levels of indirection and can be called with a generic `PdfObject` instance.

We now have the document information dictionary which includes document metadata. (Note: PDF starting with version 1.4 has two different ways of storing document metadata, and only one is demonstrated in this example.) We could examine the dictionary's elements by accessing `info.getMap()`. However, we would like to go a step further and modify one of the elements, in this case, the `Title` element. This requires making a copy of the dictionary before modifying the element, because `PdfDictionary.getMap()` returns an unmodifiable `Map` instance:

```
Map newInfo = new HashMap(info.getMap());
newInfo.put(new PdfName("Title"), new PdfString("The New Title"));
```

The old `Info` dictionary can now be replaced with the new one, using the `PdfManager.setObject(PdfObject, int)` method:

```
m.setObject(new PdfDictionary(newInfo), infoRef.getObjectNumber());
```

`PdfManager` does not need to know the object's generation number because it will automatically replace the latest generation of the object. As a result of this last example, the `PdfManager` instance now represents a document that reflects the modified `Info` dictionary.

## 1.4 Writing a document

The document being modified, we now want to write the resultant document to a file. `PdfWriter`, like `PdfReader`, will interface to either memory or the file system. It is constructed with a `File` or `OutputStream` instance, such as in the following examples:

```
PdfWriter w = new PdfWriter(new File("out.pdf"));
OutputStream os = ... ;
PdfWriter w = new PdfWriter(os);
```

Once we have a new `PdfWriter` instance, we can write the modified document using the `PdfManager.writeDocument(PdfWriter)` method:

```
m.writeDocument(w);
```

Finally, the `PdfWriter.close()` method should be called:

```
w.close();
```

This is all that is needed to read a PDF document, make a simple modification to it, and write the result to a file.

## 1.5 Appending documents

It is often useful to combine multiple PDF documents during the course of working with them. The utility class, `PdfAppender`, performs this function on a list of `PdfManager` instances:

```
List m = new ArrayList();
m.add( new PdfManager(new PdfReader(new PdfInputFile(new File("test1.pdf")))) );
m.add( new PdfManager(new PdfReader(new PdfInputFile(new File("test2.pdf")))) );
m.add( new PdfManager(new PdfReader(new PdfInputFile(new File("test3.pdf")))) );
PdfWriter w = new PdfWriter(new File("out.pdf"));
PdfAppender a = new PdfAppender(m, w);
a.append();
w.close();
```

The `PdfManager` instances can be modified normally before appending, but sometimes it is desirable to modify the resultant document *after* appending without having to go through the file system. This can be done by making `PdfAppender` write to a buffer and then constructing a new instance of `PdfManager` based on that buffer:

```

List m = ... ;
...
ByteArrayOutputStream os = new ByteArrayOutputStream();
PdfWriter w = new PdfWriter(os);
PdfAppender a = new PdfAppender(m, w);
a.append();
w.close();
os.close();
ByteBuffer bb = ByteBuffer.wrap(os.toByteArray());
PdfReader r = new PdfReader(new PdfInputBuffer(bb, "Appended Document"));
PdfManager big = new PdfManager(r);

```

Note that once the appending process has completed and the `PdfWriter` and `ByteArrayOutputStream` instances have been closed, the objects referenced by `m`, `a`, and `w` can be discarded, because the resultant document is now completely contained in the stream referenced by `os`. However, they do not have to be discarded, and the original `PdfManager` instances could be kept active to be reused or further modified. The `PdfAppender` and `PdfWriter` instances cannot be reused and therefore should be discarded.

## 1.6 Examining objects recursively

PDF objects can be nested arbitrarily within `PdfArray` and `PdfDictionary` instances, and it is sometimes necessary to examine their contents recursively. One example of this would be to update all `PdfReference` instances that reference a specific object so that they will instead reference some other (probably new) object.

The `PdfObject.filter(PdfObjectFilter)` method is provided for this purpose and is available for all subclasses of `PdfObject`. This method accepts a class that implements the `PdfObjectFilter` interface. It makes multiple calls to the `PdfObjectFilter` methods, `preFilter(PdfObject)` and `postFilter(PdfObject)`, passing to them each object as a parameter, one at a time. The `preFilter()` and `postFilter()` methods can examine the object and optionally modify it by returning a replacement object (or `null` to discard it). They are called in the following order for each object:

1. The `preFilter()` method is called with the object as a parameter.
2. If the object is a container (i.e. `PdfArray` or `PdfDictionary`), the elements it contains are examined recursively following steps 1 to 3. Note that at this point, the object is whatever was returned by `preFilter()` in step 1.
3. The `postFilter()` method is called with the object as a parameter.

We can therefore easily implement the example above to update all instances of a specific indirect reference:

```

public class ModifyReferences implements PdfObjectFilter {

    public ModifyReferences() {
    }
}

```



```
public PdfObject modify(PdfObject obj, PdfReference find, PdfReference replace)
    throws PdfFormatException {
    _find = find;
    _replace = replace;
    return obj.filter(this);
}

public PdfObject preFilter(PdfObject obj) throws PdfFormatException {
    if (obj.equals(_find)) {
        return _replace;
    } else {
        return obj;
    }
}

public PdfObject postFilter(PdfObject obj) throws PdfFormatException {
    return obj;
}

PdfReference _find;
PdfReference _replace;
}
```

The PdfRenumberOffset utility class uses this approach to implement a function for renumbering indirect references.

