PDF Clown Project 0.1

User Guide

Stefano Chizzolini

PDF Clown Project 0.1: User Guide

by Stefano Chizzolini

Revision 0.1.0.0
Published February 24, 2011

This document is an overview of the PDF Clown library, version 0.1.0.

Table of Contents

List of Figures

Chapter 1. Introduction

PDF Clown is an abstract library for creating and modifying PDF documents based upon the Adobe PDF specification 1.6. Its emphasis upon abstractness derives from the willingness to offer a clean design above all the other concerns, instead of serving rough functionality hackings that tipically lead to bloated code without a necessary holistic view.

This book provides an agile overview of the main concepts you may need to understand the way PDF Clown works. Further practical information can be retrieved in the documentation of distribution sections pertaining to PDF Clown's implementations (currently: Java and .NET/C#).

Release note

The current release of this document has been revised and somewhat extended; nonetheless, its contents are still uncomplete and sparse.

1. Audience

This book is written for software developers who want to use PDF Clown to work with PDF documents; it's also useful for those willing to contribute their own code to improve the existing code base.

Topics are presented with a top-down style, providing an overview of the library architecture, followed by details on its actual use.

2. Objectives

PDF Clown's design and implementation are inspired by some assumptions and requirements:

- clean and simple design: quick adoption, easy maintenance;
- openness: any user can directly evaluate the quality of the functional implementation investigating the code base. Extensive documentation is our commitment;
- intelligibility: code MUST be widely commented to support its readability, stressing the ties it keeps with the official PDF specification, as an implementation of its;
- conformance: adoption of established conventions, and comprehensive documentation of possible custom conventions;
- consistency: every operation or state MUST be rendered in the same way when the same conditions occur, principally through inheritance and interface realization (for example, any collective entity is rendered consistently realizing standard collection interfaces throughout the library -- it may seem banal, but you can see horribly naive iterative solutions in lots of libraries out there ;-));
- loose coupling, strong cohesion, no redundancy: classes MUST expose a minimal interface along with an exhaustive contract that provides an igienic enforcement to their encapsulation, avoiding overlapping roles and states, and promoting well-defined cooperative responsabilities;
- evolution: ongoing improvements MUST be accomplished through iterative revisions, without excessive specification details or anticipatory efforts.

3. Installation

3.1. PDF Clown for Java

PDF Clown for Java is packed inside `pdfclown.jar`. It's designed to work with Java VM 6 or later.

To reference it from within your applications is just a matter of adding its path to your `CLASSPATH` or to place it into the `WEB-INF/lib` folder.

3.2. PDF Clown for .NET

PDF Clown for .NET is packed inside `PDFClown.dll`. In order to work, it must be accompanied by its dependencies (ICSharpCode.SharpZipLib.dll), along with .NET Framework 3.5 or later.

To reference it from within your applications is just a matter of adding it to your project references as usual.

4. PDF expert in 5 minutes

OK, I'm joking! After skimming through these notes you surely won't become a PDF guru, but you'll be sufficiently knowledgeable about some basic notions of PDF necessary to grasp the overall logic that backs PDF Clown's library workings.

As you may know, PDF is a mature, versatile, standard format (see ISO 32000-1:2008) for typographic renditions. From a user perspective, a **PDF instance** (informally named 'file') is a *collection of objects deriving from a limited, well-defined bunch of types (primitive objects)*.

The PDF specification makes a good work outlining the structure of an instance: in this phase we can approximately term **file level** the domain of *primitive types*, whilst **document level** the domain of the *objects derived from them*. At file level, the primitive types impose *syntactic constraints* upon the derived objects (i.e. rules about the ways they can be defined and combined); on the other hand, at document level the derived objects impose *semantic constraints* upon your objectives (i.e. rules about the ways they are able to express meaningful information).

4.1. File-level (syntactic) view

File level represents the syntactic aspect of a PDF instance, constituted by primitive objects. Primitive objects can be classified as follows (note: this distinction doesn't appear in the official specification, but it's appropriate for our purposes):

- atomic objects:
  - booleans (logical symbols);
  - numbers (numeric symbols);
  - strings (arbitrary sequences of characters);
  - names (symbols uniquely defined by a sequence of characters);
- composite objects:
  - arrays (one-dimensional collections of possibly-heterogeneous objects arranged sequentially);
  - dictionaries (associative tables containing pairs of objects (key/value), known as the dictionary's entries);
  - streams (multipurpose, incrementally-readable sequences of bytes, used to host potentially large amounts of data such as images, embedded fonts, and so on).

Atomic objects represent simple types.

Composite objects host other objects, either by value or by reference. *By-value objects* are dubbed **direct objects**, while *by-reference objects* are dubbed **indirect objects**. In the latter case, the pointer to the actual object is named **reference**: it's of paramount importance to immediately understand that *indirect objects may be referenced from within other (composite) objects only through their respective references*. Streams are inherently indirect objects, whilst all the other primitive types instances may be alternatively direct or indirect.

A **PDF file** is, in essence, *a collection of indirect objects linked together through multiple references*.

4.2. Document-level (semantic) view

A **PDF document** (semantic view of a PDF instance) consists of *a collection of objects that together describe the appearance of one or more pages*, possibly accompanied by additional interactive elements (bookmarks, viewer preferences, annotations, acroforms, and so on) and higher-level application data (such as document-interchange features (metadata, tags, and so on)).

A document's pages (and other visual elements) can contain any combination of text, graphics, and images. A page's appearance is described by a content stream, which contains a sequence of graphics objects to be painted on the page.

Chapter 2. Architecture

PDF Clown aims at providing both a consistent, rigorous, intelligible implementation of the PDF specification and a rich multi-layered model that allows users to freely access PDF files from different levels of abstraction, in accordance to their coding requirements, style and skills.

Differently from a lot of existing PDF software libraries, PDF Clown doesn't hide the native PDF syntax and semantics behind a simplistic abstraction layer focused towards a limited range of applicability; instead, it intends to exploit the full potential of the format, taming its complexity by smoothing some of its edges and wrinkles. From the user's point of view, being allowed to access the whole data comes necessarily to the cost of becoming accustomed to some concepts and intricacies of the PDF specification; that's not a drama, as the library is shipped along with lots of code samples and a growing documentation.

Manipulation is typically accomplished through either file-level or document-level facilities (or a convenient combination of the two levels). File-level manipulation implies the use of low-level, primitive objects, giving the user a lot of flexibility along with the responsability of constructing proper semantic definitions; on the other hand, document-level manipulation implies the use of high-level, derivative, specialized objects, giving the user semantics constrained within a predefined (although freely-expandable) framework.

1. The layers

The stacked architecture of PDF Clown follows the layout suggested by the official PDF Reference 1.6 published by Adobe Systems Inc.

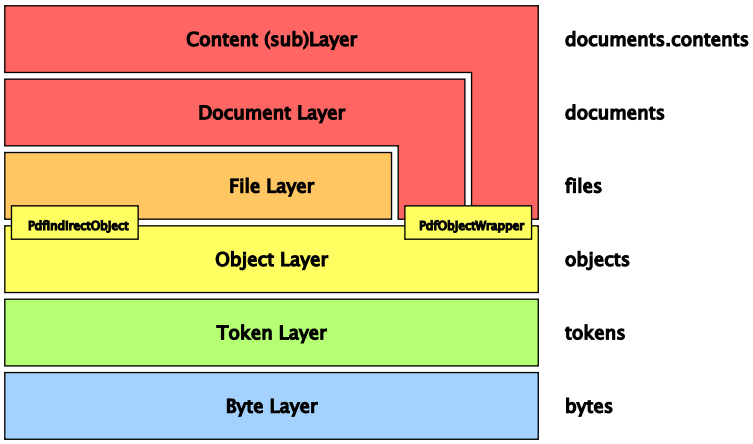

Figure 2.1. PDF Clown's layered architecture

Each layer corresponds to a subnamespace within the `org.pdfclown` project namespace (see Figure 2.1, “PDF Clown's layered architecture”):

1. **Byte Layer**: this is the lowest layer. It enables fine-grained access to the byte representation of a PDF instance. Its subnamespace is `bytes`. Its core interfaces are `IInputStream` and `IOutputStream`.

2. **Token Layer**: this represents the micro-syntactic (lexical) components of a PDF instance. Its subnamespace is `tokens`. Its core classes are `Reader` (file deserialization) and `Writer` (file serialization).

3. **Object Layer**: this represents the mid-syntactic components of a PDF instance. They are serialized by the token layer, bathed in the file layer and compose every concrete functional

("semantic") object available in the document layer. A user can alternatively manipulate them by direct access from the file layer, or by indirect access from the document layer. Their subnamespace is `objects`. Their core class is `PdfObject`, while their "bridging" classes are `PdfIndirectObject` (towards the file layer) and `PdfObjectWrapper` (towards the document layer).

4. **File Layer**: this represents the macro-syntactic structure of a PDF instance, governing the overall rules that constitute a well-formed PDF instance and bridging the document layer to the lower ones. Its subnamespace is `files`. Its core class is `File`.

5. **Document Layer**: This is the semantic framework upon which contents are built, bringing primitive objects to functional context. Its subnamespace is `documents`. Its core class is `Document`.

   Upon the document layer a series of advanced features (interaction, interchange etc.) are built, the most prominent of which is the content sublayer.

6. **Content (sub)Layer**: this is the semantic sublayer that actually defines contents, deputed to manipulate the very graphical representation of the PDF instance. Its subnamespace is `documents.contents`. Its core class is `Contents`, which represents a content stream (a sequence of instructions describing the appearance of a page or other graphical entity).

1.1. An object across the layers

In order to apply the aforementioned concepts to a practical example, the following diagram (see Figure 2.2, “A document object as viewed across the PDF Clown's layers”) represents a Document object (the semantic root of a PDF instance as modelled by PDF Clown) as it's viewed across the layers:

1. **Byte layer** (raw PDF instance): the dotted box inside the PDF instance icon contains a sample data fragment that represents a Catalog Dictionary (root object);

2. **Token layer** (lexical interpretation (parsing) of a PDF instance): the bytes of the Catalog Dictionary are aggregated in atomic items (lexemes);

3. **Object layer** (data structures emerging from token aggregation): an indirect object pattern is recognized, so that a PdfIndirectObject is instantiated to incapsulate the Catalog Dictionary data;

4. **File layer** (higher syntactic representation of a PDF instance in PDF Clown model): the PdfIndirectObject containing the Catalog Dictionary is arrayed among the others to represent the PDF instance structure;

5. **Document layer** (semantic representation of a PDF instance in PDF Clown model): the Catalog Dictionary is encapsulated inside a Document object, which inherits from PdfObjectWrapper (bridge between the object layer and the document layer).

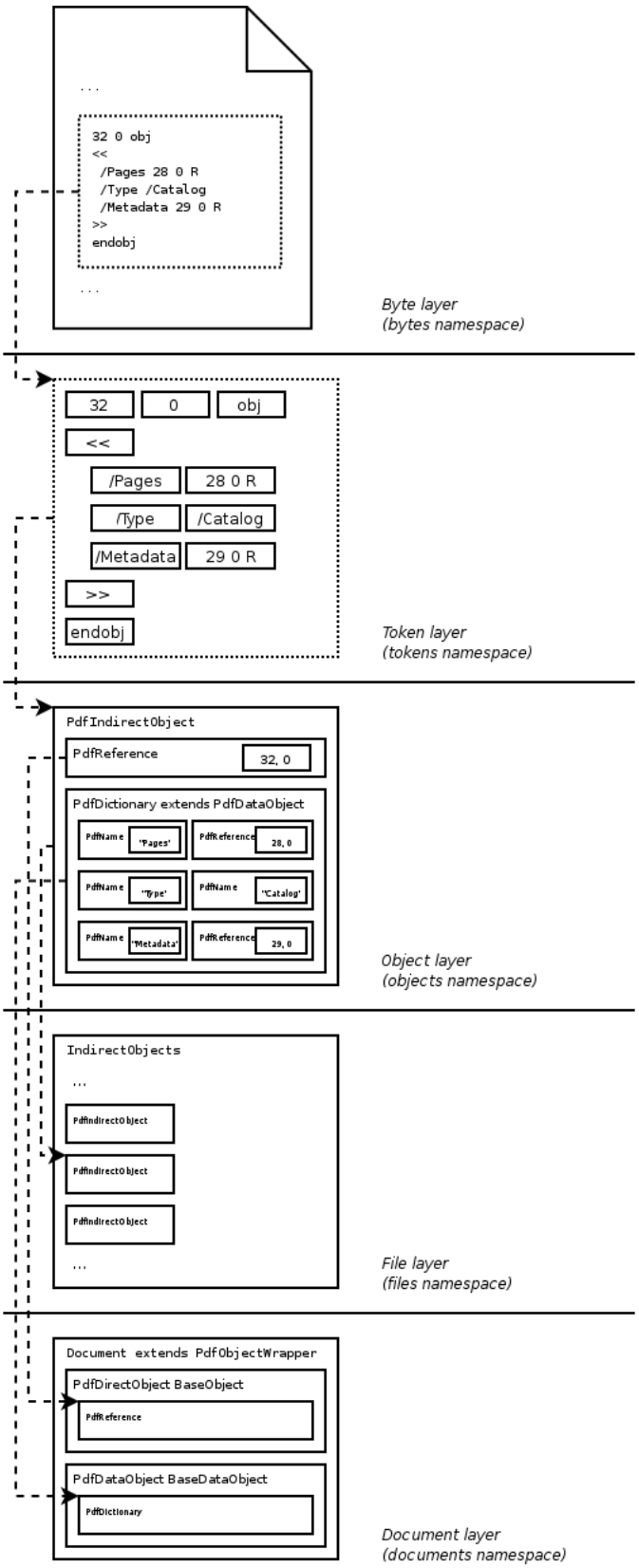

Figure 2.2. A document object as viewed across the PDF Clown's layers

2. Byte layer: low-level I/O

`org.pdfclown.bytes` namespace provides the I/O interfaces and classes extensively used throughout the library to manipulate raw data (both as streams and as buffers).

`bytes.filters` subnamespace contains stream codecs used for data transformations, such as compression/decompression. They are tipically used by PdfStream objects to transparently manage their bodies.

3. Token layer: serialization/deserialization

`org.pdfclown.tokens` namespace is deputed to manage the transitions from/to the serialized PDF form to/from the PDF Clown runtime form, that is between a PDF file and its representation within the PDF Clown's object model.

The `Reader` class has the responsability to read data from an existing PDF file and convert it into its corresponding object representation, while the `Writer` class has the complementary responsability to convert the object representation into its corresponding PDF data and write it to a PDF file.

Ordinary users shouldn't care about this namespace as these operations are transparently managed by PDF Clown itself.

4. Object layer: the primitive objects

Primitive objects are the basic bricks that constitute a PDF instance; PDF Clown implements them as follows:

- data objects:
  - by-value objects:
    - atomic objects:
      - PdfBoolean (boolean);
      - PdfInteger (integer number), PdfReal (real number);
      - PdfString (generic string, either literal or hexadecimal), PdfTextString (text, i.e. specialized textual string), PdfDate (date, i.e. date-formatted string);
      - PdfName (name);
    - composite objects:
      - PdfArray (array);
      - PdfDictionary (dictionary);
      - PdfStream (stream);
  - by-reference objects: PdfReference (indirect reference);
- indirect objects: PdfIndirectObject.

Data objects represent actual contents, whilst indirect objects are containers of data objects whose purpose is to allow them to be referenced by other (composite) data objects through `PdfReference` instances. For example, Figure 2.2, “A document object as viewed across the PDF Clown's layers” shows a `PdfDictionary` that is contained by a `PdfIndirectObject` associated to a

`PdfReference` identified by the tuple <32,0> (by the way: the first element (32) represents the unique object number, the second element (0) represents the generation number used for incremental updates).

Both `PdfIndirectObject` and `PdfReference` implement the `IPdfIndirectObject` interface which handily exposes indirect object, reference and data object at the same time.

4.1. Object referencing

Data objects can be combined either by value or by reference. In order to reference a data object (`PdfArray`, `PdfString`, `PdfDictionary`, and so on), you firstly need to encapsulate it within an indirect object (`PdfIndirectObject`), then you have to use its corresponding reference (`PdfReference`). Keep in mind this triplet (reference, indirect object, data object), as such relation is the most important of the entire PDF model!

Referencing an object *across multiple abstraction levels* (e.g., a data-level object as a `PdfDictionary` that backs a document-level object as a `Pages` collection) is called in PDF Clown's jargon "**deep referencing**" or "vertical referencing".

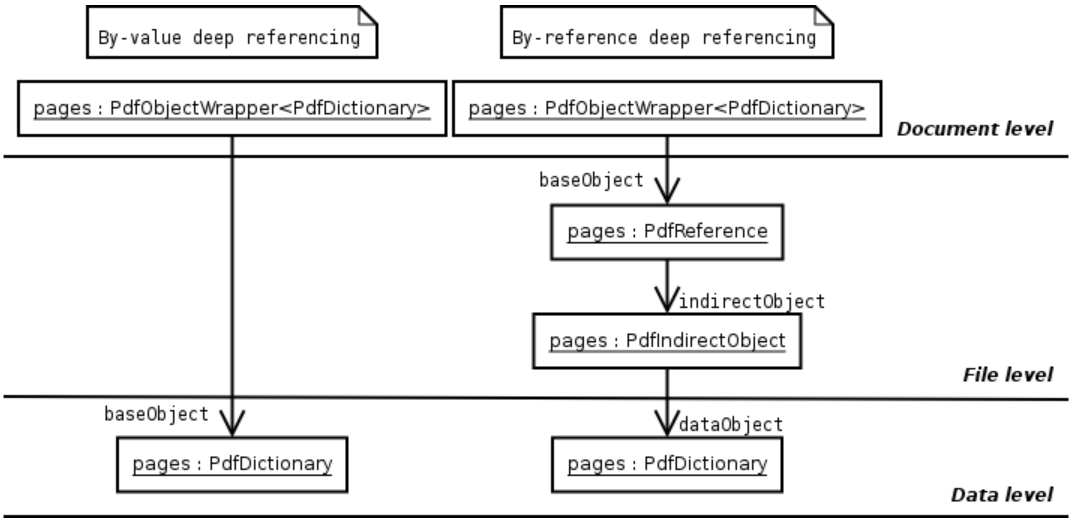

Figure 2.3. Deep referencing (both by-value and by-reference)

All the objects at document level (like a `Pages` collection) are derived from `PdfObjectWrapper`, which wraps a corresponding data object (like a `PdfDictionary`) either directly (by-value deep referencing) or indirectly (by-reference deep referencing). The wrapped object is exposed by the `baseObject` property: in case of indirect wrapping this property returns a `PdfReference` instance that points to a `PdfIndirectObject` which exposes the data object through the `dataObject` property.

On the other hand, referencing an object *within the same abstraction level* (e.g., a `Page` that is associated to a `Pages` collection object) is called "**shallow referencing**" or "horizontal referencing".

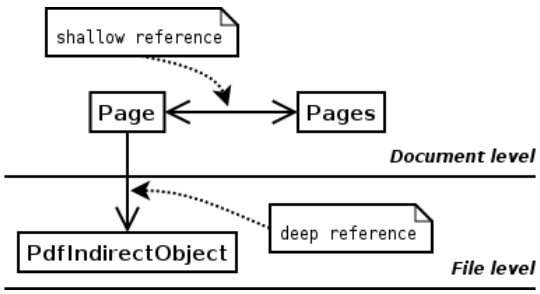

Figure 2.4. Deep and shallow references (simplified view)

4.2. Object reuse

PDF Clown tries to let you achieve a good deal of flexibility when it comes to data object reuse. Particularly, it performs the task of leveraging existing contents in two major ways:

- **dynamic reuse**: content reuse *within the same PDF instance*, performed exploiting the indirect reference functionality natively supported by PDF;
- **static reuse**: content reuse *from other PDF instances*, performed cloning existing contents from a source file to your target file.

4.2.1. Dynamic reuse

Dynamic reuse is the *referencing of contents within the same PDF instance*. It's accomplished through indirect object references (`PdfReference` class). As previously mentioned, each `PdfReference` points to a corresponding `PdfIndirectObject` which contains a corresponding data object. This way the same data object can be used multiple times across the PDF instance just referencing it. For example, an image could appear on several pages yet being defined just one time as an `ImageXObject` (by the way, an `ImageXObject` (which stands for "image external object") is a PdfStream bearing an image's data that can be referenced from content streams in order to be shown).

4.2.2. Static reuse

Static reuse is the *hard-copying of contents from an existing PDF instance to another*. It's accomplished by PDF Clown through a file-level operation named "contextual cloning" that leverages the "deep referencing" or "vertical referencing" functionality described in the previous section. In order to activate such imported content at document level, it's then needed to define appropriate references among the objects within the same PDF instance to link them together ("shallow referencing" or "horizontal referencing") in a meaningful way. See Figure 2.4, “Deep and shallow references (simplified view)”.

Each PDF instance represents an independent context. When a document-level object (e.g. a Page) is created within or imported into a PDF instance, it must be made aware of the context where it lives, so that it can silently perform context-bound operations without further client involvement. Context referencing at creation time is performed passing a document reference to the document-level object constructor, whilst in case of importing it's performed passing a document reference to the contextual cloning method of the document-level object.

Eventually, behind the scenes a document reference degrades to a file reference, as File objects are the entities backing their respective Document objects within the PDF Clown architecture (as suggested by the PDF specification).

As stated at the beginning of this section, when you manipulate a high-level (i.e. document-level) object (inheriting from PdfObjectWrapper), you have to address two reference levels: a deep, file-level reference that represents the existence of the object inside the PDF instance through an indirect object, and one or more shallow, document-level references that represent the application of the object inside the PDF instance. See Figure 2.5, “Deep and shallow references (detailed view)”

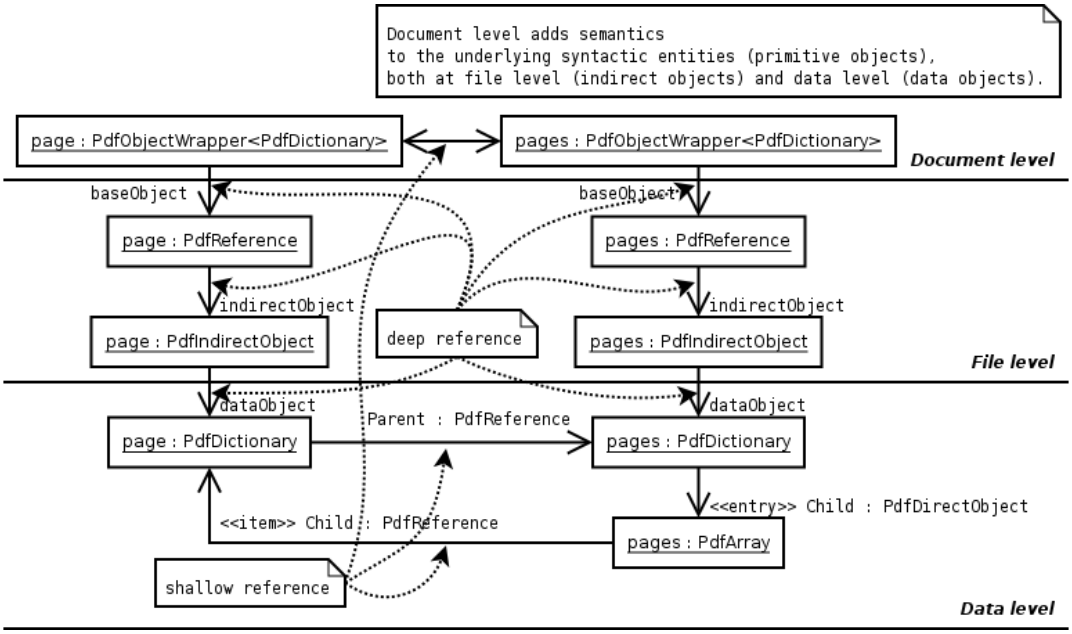

Figure 2.5. Deep and shallow references (detailed view)

For example, if you want to remove a page from a document, it suffices to invoke the `remove` method of the `Pages` collection: this is a shallow, document-level reference, so the page has gone, but... its phantom keeps lurking behind at file level, as you haven't removed the indirect object backing your page! So we also need to perform a file-level removal of its indirect object, to eliminate its deep reference.

You may say: why damn is there such a "duplication"?! Well, flexibility implies more degrees of expression: if you wanted not to remove your page from the document, but just to move it in another position within the same document, you would only perform a shallow removal followed by an insertion of the same page at the target position.

So, accordingly to the layered structure modelled by PDF Clown, inserting external objects into a document is just a matter of accomplishing two operations:

1. to insert a clone of the external object into the intended *file-level context (deep reference)*, causing it to become an imported internal *inactive* object;

2. to insert the imported internal inactive object into the intended *document-level context (shallow reference)*, causing it to become *active*.

As document-level objects work assuming that all encountered entities belong to the same underlying (file) context, *it's client responsability to ensure that objects such as a page are properly contextualized within the same file before being used*. To say: you must always ensure that the document-level object (in our example: a Page) you're working with belongs to the file you're working inside. There are simply three ways to do this: if your document-level object has already been part of the document you're working inside, you're OK; if your document-level object has just been created, you have had to pass a reference of your document to its constructor (contextual creation); if your document-level object has just been imported from another document, you have had to pass a reference of your document to its cloning method (contextual cloning). That's why you aren't allowed to just append an alien page to your document: you must first contextually clone it to force its existence inside your file, then you are allowed to use this clone to append to your document.

5. File layer: syntactic structure

`org.pdfclown.files` namespace represents the actual structure of a PDF instance, made up of a collection of indirect objects (`IndirectObjects`) accessible through the `indirectObjects` property of the `File` class. The latter is the pivotal object of the syntactic model, used to manipulate any PDF instance at low level.

6. Document layer: semantic structure

`org.pdfclown.documents` namespace wraps the syntactic structure into high-level objects revolving around a collection of pages (`Pages`) accessible through the `pages` property of the `Document` class. The latter is the pivotal object of the semantic model, used to manipulate any PDF instance at high level.

7. Content sublayer: the content stream model

Since version 0.0.4, PDF Clown supports a *fully object-oriented content stream model within the content sublayer* (see `documents.contents.objects` subnamespace). This practically implies that you can access each content object (operations and composite objects like Text, InlineImage and so on) that constitutes a page, and modify/add/remove it. A typical scenario of use may be text replacement (discovering text occurrences and modifying them with new text).

Moreover, *content scanning* (see `documents.contents.ContentScanner` class) of the content object sequence within a content stream (see `documents.contents.Contents` class) provides the access to the graphics state transitions, delivering useful information such as current text parameter values (eg: character and word spacing, horizontal scaling, text rise, etc.) and graphics parameter values (eg: line width, line cap style, line join style, current transformation matrix, etc.). This functionality is the powerful base for lots of possible applications, such as text extraction (see 0.0.8 version) and content visual rendering.

7.1. Core types

Contents are described as sequences of content objects within content streams; each content stream is marshalled/unmarshalled through the `documents.contents.Contents` class.

Contents may be included within a content stream either *by value* (so-called **inline** contents) or *by reference* (so-called **external** contents).

Every content stream is associated to a context (see `documents.contents.IContentContext` interface) which provides additional information such as resources (fonts, colors, images and so on) and canvas surface bounds. `IContentContext` inherits from `documents.contents.IContentEntity` interface, a powerful abstraction that applies even to objects outside the PDF model (such as those within the `documents.contents.entities` subnamespace); it allows to transparently obtain inline (`toInlineObject()` method) and external (`toXObject()` method) content representations equivalent to the source entity. This functionality has a twofold applicability: existing content stream transformation (such as in case of a `Page` to be rendered as an external form through its `toXObject()` method implementation) and abstract entities rendering (such as in case of a `documents.contents.entities.Image` or a `documents.contents.entities.Barcode` to be inserted into a PDF instance either inside an existing content stream (`toInlineObject()` method) or creating an external stream (`toXObject()` method)).

7.2. Typesetting

PDF Clown features a stacked architecture for page composition gathered inside the `documents.contents.composition` subnamespace, sitting upon the content stream model. Its layers (see Figure 2.6, “Typesetting stack”) are represented by:

1. **primitive typesetting**: the `PrimitiveFilter` class provides the graphics primitives defined

by the PDF 1.6 specification (e.g. coordinate space transformations, text, path, external object insertion etc.). This is the base for the following composition layers;

2. **positional typesetting**: the `BlockFilter` class forces contents to fit within page areas (i.e. horizontal and vertical alignment for low-level typographic entities insertion such as text and images);

3. **flow typesetting**: the `FlowFilter` class (currently to do yet) allows contents to spread across multiple pages (i.e. high-level typographic entities insertion such as paragraphs and tables).

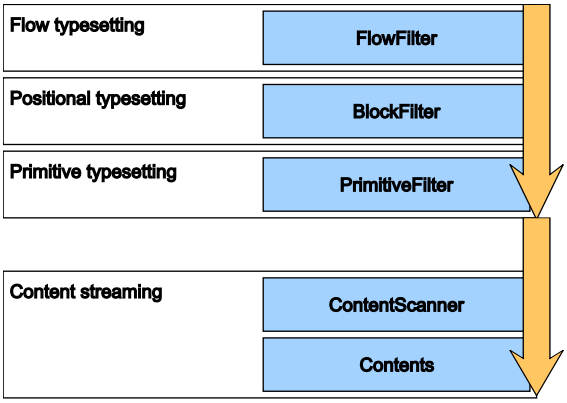

Figure 2.6. Typesetting stack

Chapter 3. Basic usage

Note

The code fragments included in this chapter avoid to repeat, for the sake of succinctness, the declarations of already mentioned variables.

Release note

This chapter is in progress, lacking lots of useful tips that you may like to know; next documentation releases will expand the coverage. Please see the code samples available in the PDF Clown distribution for actually working examples [see Appendix A].

1. Creating a new PDF instance

```
import org.pdfclown.documents.Document;
import org.pdfclown.files.File;

. . .

// 1. File instantiation.
File file = new File();

// 2. Document retrieval.
Document document = file.getDocument();

// 3. Document manipulation.
. . .

// 4. File serialization.
file.writeTo(
  new java.io.File("MyFile.pdf"),
  SerializationModeEnum.Standard
  );
```

As you can see from the above sample, creating a new PDF instance is really simple:

- 1. **File instantiation**: this is the first operation you execute when creating a new PDF file. Then you can either manipulate directly the low-level `IndirectObjects` collection or (see next step) access its high-level representation (`Document`);
- 2. **Document retrieval**: once a file has been instantiated, its corresponding Document can be retrieved in order to access the high-level model (document pages and so on).
- 3. **Document manipulation**: once a document reference has been acquired, any high-level operation over your PDF instance can be done. We'll examine this point in the following samples.
- 4. **File serialization**: once the file has been defined, it must be serialized in order to persist its data.

2. Indirect objects

The following sample gets an indirect object (whose object number is '12') from the indirect objects collection of a file:

```
import org.pdfclown.files.File;
import org.pdfclown.files.IndirectObjects;
import org.pdfclown.object.PdfDataObject;
import org.pdfclown.object.PdfIndirectObject;
import org.pdfclown.object.PdfReference;

. . .
```

```
IndirectObjects indirectObjects = file.getIndirectObjects();
PdfIndirectObject indirectObject = indirectObjects.get(12);
PdfReference reference = indirectObject.getReference();
PdfDataObject dataObject = indirectObject.getDataObject();
```

As `PdfReference` implements the same `IPdfIndirectObject` interface exposed by `PdfIndirectObject`, you can obtain the indirect object and data object associated to it:

```
import org.pdfclown.object.PdfDataObject;
import org.pdfclown.object.PdfIndirectObject;
import org.pdfclown.object.PdfReference;

. . .

PdfIndirectObject indirectObject = reference.getIndirectObject();
PdfDataObject dataObject = reference.getDataObject();
int objectNumber = reference.getObjectNumber();
int generationNumber = reference.getGenerationNumber();
```

The **object number** is the unique identifier of each indirect object within the same file (as you saw in the previous sample, indirect objects are just accessed by object number from the `IndirectObjects` collection associated to the `File` object).

Appendix A. Samples

This is an index to the code samples within this distribution, whose purpose is to demonstrate some of the possible applications of the PDF Clown library. You can find within each implementation subtree ("java" and "dotNET"); they are marked according to their project type:

- [CLI]: command line sample (see "pdfclown.samples.cli" project)
- [GUI]: graphical user interface sample (see "pdfclown.samples.gui" project)
- [WEB]: web sample (see "pdfclown.samples.web" project)

1. Acroform

- AcroFormCreationSample [CLI]: CheckBox, ComboBox, ListBox, PushButton, RadioButton, TextField.
- AcroFormParsingSample [CLI]: Field.

2. Actions

- AcroFormCreationSample [CLI]: JavaScript, WidgetActions.
- ActionSample [CLI]: DocumentActions, GoToLocal, GoToURI, PageActions.
- BookmarksParsingSample [CLI]: Action.
- ComplexTypesettingSample [CLI]: GoToURI.
- LinkCreationSample [CLI]: GoToEmbedded, GoToURI.
- LinkParsingSample [CLI]: Action, Destination, GoToDestination, GoToEmbedded, GotoNonLocal, GoToURI.

3. Annotations

- AnnotationSample [CLI]: CalloutNote, Caret, Ellipse, FileAttachment, Line, Note, Rectangle, RubberStamp, Scribble.
- LinkCreationSample [CLI]: FileAttachment, Link.
- LinkParsingSample [CLI]: Annotation, Destination, Link.

4. Barcodes

- BarcodeSample [CLI]: EAN13Barcode.

5. Bookmarks

- BookmarksParsingSample [CLI]: Bookmark, Bookmarks.
- ComplexTypesettingSample [CLI]: Bookmark, Bookmarks.

6. Content manipulation

- ContentRemovalSample [CLI]: ContainerObject, ContentObject, Contents.
- ContentTweakingSample [CLI]: ContainerObject, ContentObject, Contents

7. Content rasterization (printing and rendering)

- PrintingSample [CLI]: Renderer.
- RenderingSample [CLI]: Renderer.

8. Content reuse

- ComplexTypesettingSample [CLI]: FormXObject, XObject.
- PageToFormSample [CLI]: FormXObject.
- WatermarkSample [CLI]: FormXObject.

9. Content scanning

- ContentScanningSample [CLI]: ContainerObject, ContentObject, ContentScanner, ContentScanner.GraphicsObjectWrapper.
- ParsingSample [CLI]: CompositeObject, ContentObject, Contents, Operation, Resources.
- PdfInspectorSample [GUI (Java-only)]: ContentScanner.

10. Destinations

- ComplexTypesettingSample [CLI]: LocalDestination.
- LinkParsingSample [CLI]: Destination.
- NamedDestinationSample [CLI]: LocalDestination, NamedDestinations.
- NamesParsingSample [CLI]: Destination, NamedDestinations.

11. Graphics composition

- BarcodeSample [CLI]: XObject.
- GraphicsSample [CLI]: BlockFilter, PrimitiveFilter.
- PageCoordinatesSample [CLI]: BlockFilter, PrimitiveFilter.
- PageFormatSample [CLI]: PageFormat.
- PageNumberingSample [CLI]: PageStamper, PrimitiveFilter.
- PageToFormSample [CLI]: BlockFilter, PrimitiveFilter.
- TextInfoExtractionSample [CLI]: PrimitiveFilter.

12. Images

- ComplexTypesettingSample [CLI]: Image.
- ImageExtractionSample [CLI]: PdfStream.
- ImageSubstitutionSample [CLI]: Image, ImageXObject.
- InlineObjectSample [CLI]: Image, PrimitiveFilter.
- TransformationSample [CLI]: Image.

13. Low-level manipulation

- ImageExtractionSample [CLI]: IBuffer, PdfDataObject, PdfDictionary, PdfIndirectObject, PdfStream.
- IndirectObjectEditingSample [CLI]: IBuffer, PdfDataObject, PdfDictionary, PdfIndirectObject, PdfName, PdfReference, PdfStream.
- ParsingSample [CLI]: PdfDataObject, PdfReference.
- PrimitiveObjectSample [CLI]: PdfArray, PdfDictionary, PdfInteger, PdfName.

14. Page management

- ImageSubstitutionSample [CLI]: Resources, XObjectResources.
- PageFormatSample [CLI]: Page, PageFormat
- PageManagementSample [CLI]: PageManager.
- PageTransitionSample [CLI]: Transition.

15. Text extraction

- AdvancedPlainTextExtractionSample [CLI]: TextExtractor.

- AdvancedTextExtractionSample [CLI]: ITextString, TextExtractor.
- BasicTextExtractionSample [CLI]: ContentScanner, Font, ShowText.
- LinkParsingSample [CLI]: ContentScanner, ContentScanner.TextStringWrapper, ContentScanner.TextWrapper, TextChar.
- TextInfoExtractionSample [CLI]: ContentScanner, ContentScanner.TextStringWrapper, ContentScanner.TextWrapper, TextChar.

16. Typesetting

- ComplexTypesettingSample [CLI]: BlockFilter, Font, PrimitiveFilter.
- HelloWorldSample [CLI]: PrimitiveFilter, StandardType1Font.
- StandardFontSample [CLI]: PrimitiveFilter, StandardType1Font.
- TextFrameSample [CLI]: BlockFilter, Font, PrimitiveFilter, StandardType1Font.
- TypesettingSample [CLI]: BlockFilter, Font, PrimitiveFilter.
- UnicodeSample [CLI]: BlockFilter, Font, PrimitiveFilter.

17. Web

- HelloWorld [WEB (Java-only)]: how to use PDF Clown in a servlet context.