Chapter 1. PDF Syntax

We'll begin our exploration of PDF by diving right into the building blocks of the PDF file format. Using these blocks, you'll see how a PDF is constructed to lead to the page-based format that you are familiar with.

PDF Objects

The core part of a PDF file is a collection of "things" that the PDF standard (ISO 32000) refers to as *objects*, or sometimes *COS objects*.

> NOTE
>
> COS stands for Carousel Object System and refers to the original/code name for Adobe's Acrobat product.

These aren't objects in the "object-oriented programming" sense of the word; instead, they are the building blocks on which PDF stands. There are nine types of objects: null, Boolean, integer, real, name, string, array, dictionary, and stream.

Let's look at each of these object types and how they are serialized into a PDF file. From there, you'll then see how to take these object types and use them to build higher-level constructs and the PDF format itself.

Null Objects

The null object, if actually written to a file, is simply the four characters *null*. It is

Boolean Objects

Boolean objects represent the logical values of true and false and are represented accordingly in the PDF, either as `true` or `false`.

> **NOTE**
>
> When writing a PDF, you will always use `true` or `false`. However, if you are reading/parsing a PDF and wish to be tolerant, be aware that poorly written PDFs may use other capatilization forms, including leading caps (`True` or `False`) or all caps (`TRUE` or `FALSE`).

Numeric Objects

PDF supports two different types of numeric objects—integer and real—representing their mathematical equivalents. While older versions of PDF had stated implementation limits that matched Adobe's older implementations, those should no longer be taken to be file format limitations (nor should those of any specific

> **NOTE**
>
> While PDF supports 64-bit numbers (so as to enable very large files), you will find that most PDFs don't actually need them. However, if you are reading a PDF, you may indeed encounter them, so be prepared.

Integer numeric objects consist of one or more decimal digits optionally preceded by a sign, representing a signed value (in base 10). Example 1-1 shows a few examples of integers.

Example 1-1. Integers

```
1
-2
+100
612
```

Real numeric objects consist of one or more decimal digits with an optional sign and a leading, trailing, or embedded period representing a signed real value. Unlike

PostScript, PDF does not support scientific/exponential format, nor does it support

viewer might use *double*, *float*, or even *fixed point* numbers. Since the implementations may differ, the number of decimal places of precision may also differ. It is therefore recommended for reliability and also for file size considerations to not write more than four decimal places.

Example 1-2 shows some examples of what real numbers look like in PDF.

Example 1-2. Reals

```
0.05
.25
-3.14159
300.9001
```

Name Objects

A name object in PDF is a unique sequence of characters (except character code 0, ASCII null) normally used in situations where there is a fixed set of values. Names are written into a PDF with a / (SOLIDUS) character followed by a UTF-8 string, with a special encoding form for any nonregular character. Nonregular characters are those defined to be outside the range of `0x21` (!) through `0x7E` (~), as well as any white-space character (see Table 1-1). These nonregular characters are encoded starting with a # (NUMBER SIGN) and then the two-digit hexadecimal code for the character.

Because of their unique nature, most names that you will write into a PDF are pre-defined in ISO 32000 or will be derived from external data (such as a font or color name).

NOTE

If you need to create your own nonexternal data-based custom names (such as a private piece of metadata), you must follow the rules for *second class names* as defined in ISO 32000-1:2008, Annex E, if you wish your file to be considered a valid PDF. A second class name is one that begins with your four-character ISO-registered prefix followed by an underscore (_) and then the key name. An example is included at the end of Example 1-3.

Example 1-3. Names

/Type

String Objects

Strings as they are serialized into PDF are simply series of (zero or more) 8-bit bytes written either as literal characters enclosed in parentheses, (and), or hexadecimal data enclosed in angle brackets (< and >).

A literal string consists of an arbitrary number of 8-bit characters enclosed in parentheses. Because any 8-bit value may appear in the string, the unbalanced parentheses ( )) and the backslash (\) are treated specially through the use of the backslash to escape special values. Additionally, the backslash can be used with the special `\ddd` notation to specify other character values.

Literal strings come in a few different varieties:

ASCII
: A sequence of bytes containing only ASCII characters

A sequence of bytes encoded according to the PDFDocEncoding (ISO 32000–1:2008, 7.9.2.3)

Text
: A sequence of bytes encoded as either the PDFDocEncoding or as UTF–16BE (with the leading byte order marker)

Date
: An ASCII string whose format `D:YYYYMMDDHHmmSSOHH'mm` is described in ISO 32000–1:2008, 7.9.4

> NOTE
>
> Dates, as a type of string, were added to PDF in version 1.1.

A series of hexadecimal digits (0–9, A–F) can be written between angle brackets, which is useful for including more human-readable arbitrary binary data or Unicode values (UCS-2 or UCS-4) in a string object. The number of digits must always be even, though

white-space characters may be added between pairs of digits to improve human


```
(A\053B)                    % Same as (A+B)
(Français)                  % PDFDocEncoded
<FFFE0040>                  % Text with leading BOM
(D:19990209153925-08'00')   % Date
<1C2D3F>                    % Arbitrary binary data
```

NOTE

The percent sign (%) denotes a comment; any text that follows it is ignored.

The previous discussion about strings was about how the values are serialized into a PDF file, not necessarily how they are handled internally by a PDF processor. While such internal handling is outside the scope of the standard, it is important to remember that different file serializations can produce the same internal representation (like `(A\053B)` and `(A+B)` in Example 1-4).

Array Objects

An array object is a heterogeneous collection of other objects enclosed in square brackets ([and]) and separated by white space. You can mix and match any objects of any type together in a single array, and PDF takes advantage of this in a variety of places. An array may also be empty (i.e., contain zero elements).

While an array consists only of a single dimension, it is possible to construct the equivalent of a multidimensional array. This construct is not used often in PDF, but it does appear in a few places, such as the `Order` array in a data structure known as an optional content group dictionary. (See Optional Content Groups.)

NOTE

There is no limit to the number of elements in a PDF array. However, if you find an alternative to a large array (such as the page tree for a single `Kids` array), it is always better to avoid them for performance reasons.

Some examples of arrays are given in Example 1-5.

Example 1-5. Arrays

Dictionary Objects

As it serves as the basis for almost every higher-level object, the most common object in PDF is the dictionary object. It is a collection of key/value pairs, also known as an *associative table*. Each key is always a name object, but the value may be any other type of object, including another dictionary or even null.

> NOTE
>
> When the value is null, it is treated as if the key is not present. Therefore, it is better to simply not write the key, to save processing time and file size.

A dictionary is enclosed in double angle brackets (<< and >>). Within those brackets, the keys may appear in any order, followed immediately by their values. Which keys appear in the dictionary will be determined by the definition (in ISO 32000) of the

While many existing implementations tend to write the keys sorted alphabetically, that is neither required nor expected. In fact, no assumptions should be made about dictionary processing, either—the keys may be read and processed in any order. A dictionary that contains the same key twice is invalid, and which value is selected is undefined. Finally, while it improves human readability to put line breaks between key/value pairs, that too is not required and only serves to add bytes to the total file size.

> NOTE
>
> There is no limit to the number of key/value pairs in a dictionary.

Example 1-6 shows a few examples.

Example 1-6. Dictionaries

```
% a more human-readable dictionary
<<
```

```
/Type /Page
```


Name trees

A *name tree* serves a similar purpose to a dictionary, in that it provides a way to associate keys with values. However, unlike in a dictionary, the keys are string objects instead of names and are required to be ordered/sorted according to the standard Unicode collation algorithm.

This concept is called a name tree because there is a "root" dictionary (or node) that refers to one or more child dictionaries/nodes, which themselves can refer to one or more child dictionaries/nodes, thus creating many branches of a tree-like structure.

The root node holds a single key, either `Names` (for a simple tree) or `Kids` (for a more complex tree). In the case of a complex tree, each of the intermediate nodes will also have a `Kids` key present; the final/terminal nodes of each branch will contain the `Names` key. It is the array value of the `Names` key that specifies the keys and their values by alternating key/value, as shown in Example 1-7.

Example 1-7. Example name trees

```
% Simple name tree with just some names
1 0 obj
<<
    /Names    [
        (Apple)    (Orange)        % These are sorted, hence A, N, Z...
        (Name 1) 1                 % and values can be any type
        (Name 2) /Value2
        (Zebra) << /A /B >>
    ]
>>
endobj
```

Number trees

A *number tree* is similar to a name tree, except that its keys are integers instead of strings and are sorted in ascending numerical order. Also, the entries in the leaf (or root) nodes containing the key/value pairs are found as the value of the `Nums` key instead of the `Names` key.

Stream Objects

Streams in PDF are arbitrary sequences of 8-bit bytes that may be of unlimited length

containing attributes of the stream and referred to as the *stream dictionary*. The use of the words `stream` (followed by an end-of-line marker) and `endstream` (preceded by an end-of-line marker) serve to delineate the stream data from its dictionary, while also differentiating it from a standard dictionary object. The stream dictionary never exists on its own; it is always a part of the stream object.

The stream dictionary always contains at least one key, `Length`, which represents the number of bytes from the beginning of the line following `stream` until the last byte before the end-of-the-line character preceding `endstream`. In other words, it is the actual number of bytes serialized into the PDF file. In the case of a compressed stream, it is the number of compressed bytes. Although not commonly provided, the original uncompressed length can be specified as the value of a `DL` key.

One of the most important keys that can be present in the stream dictionary is the `Filter` key, which specifies what (if any) compression or encoding was applied to the original data before it was included in the stream. It's quite common to compress large images and embedded fonts using the `FlateDecode` filter, which uses the same lossless

filters are `DCTDecode`, which produces a JPEG/JFIF-compatible stream, and `JPXDecode`, which produces a JPEG2000-compatible stream. Other filters can be found in ISO 32000-12008, Table 6. Example 1-8 shows what a steam object in PDF might look like.

Example 1-8. An example stream

```
<<
    /Type       /XObject
    /Subtype    /Image
    /Filter     /FlateDecode
    /Length     496
    /Height     32
    /Width      32
>>

stream
% 496 bytes of Flate-encoded data goes here...
endstream
```

Direct versus Indirect Objects

Now that you've been introduced to the types of objects, it is important to understand

seen in all of the examples so far.

Indirect objects are those that are referred to (indirectly!) by reference and a PDF reader will have to jump around the file to find the actual value. In order to identify which object is being referred to, every indirect object has a unique (per-PDF) ID, which is expressed as a positive number, and a generation number, which is always a nonnegative number and usually zero (0). These numbers are used both to define the object and to reference the object.

> **NOTE**
>
> While originally intended to be used as a way to track revisions in PDF, generation numbers are almost never used by modern PDF systems, so they are almost always zero.

To use an indirect object, you must first define it using the ID and generation along with the `obj` and `endobj` keywords, as shown in Example 1-9.

Example 1-9. Indirect objects made entirely from direct objects

```
3 0 obj         % object ID 3, generation 0
<<
 /ProcSet [ /PDF /Text /ImageC /ImageI ]
 /Font <<
     /F1 <<
        /Type /Font
        /Subtype /Type1
        /Name /F1
        /BaseFont/Helvetica
        >>
     >>
>>
endobj

5 0 obj
(an indirect string)
endobj

% an indirect number
4 0 obj
1234567890
endobj
```

```
3 0 obj                  % object ID 3, generation 0
<<
 /ProcSet 5 0 R          % reference the indirect object with ID 5, generatio
 /Font <</F1 4 0 R >>    % reference the indirect object with ID 4, generatio
>>
endobj
4 0 obj                  % object ID 4, generation 0
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont/Helvetica
>>
endobj
5 0 obj                  % object ID 5, generation 0
[ /PDF /Text /ImageC /ImageI ]
endobj
```

By using a combination of ID and generation, each object can be uniquely identified inside of a given PDF. Using the cross-reference table feature of PDF, each indirect object can easily be located and loaded on demand from the reference.

> **NOTE**
>
> Unless otherwise indicated by ISO 32000, any time you use an object it can be of either type—except for streams, which can only be indirect.

File Structure

If you were to view a simple PDF file—let's call it *Hello World.pdf*—in a PDF viewer, it would look like Figure 1-1.

Figure 1-1. Hello World.pdf

But if you were to view *Hello World.pdf* in a text editing application, it would look something like Example 1-11.

Example 1-11. What "Hello World.pdf" looks like in a text editor

```
%PDF-1.4
%%EOF
```

Developing with PDF by Leonard Rosenthol

```
1 0 obj
<<
 /Type /Page
 /Parent 5 0 R
 /MediaBox [ 0 0 612 792 ]
 /Resources 3 0 R
 /Contents 2 0 R
>>
endobj

4 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont/Helvetica
>>
endobj

2 0 obj

 /Length 68
>>
stream
BT
 /F1 24 Tf
 1 0 0 1 260 600 Tm
 (Hello World)Tj
ET
endstream
endobj

5 0 obj
<<
 /Type /Pages
 /Kids [ 1 0 R ]
 /Count 1
>>
endobj

3 0 obj
<<
 /ProcSet[/PDF/Text]
 /Font <</F1 4 0 R >>
>>
endobj
```

```
xref
0 7
```



```
0000000333 00000 n
0000000009 00000 n
trailer
<<
 /Size 7
 /Root 6 0 R
>>
startxref
488
%%EOF
```

Looking at that, you might get the mistaken impression that a PDF file is a text file that can be routinely edited using a text editor—it is *not*! A PDF file is a structured 8-bit binary document delineated by a series of 8-bit character-based tokens, separated by white space and arranged into (arbitrarily long) lines. These tokens serve not only to delineate the various objects and their types, as you saw in the previous section, but also to define where the four logical sections of the PDF begin and end. (See Figure 1-2.)

> **NOTE**
>
> As noted previously, the tokens in a PDF are always encoded (and therefore decoded) as 8-bit bytes in ASCII. They cannot be encoded in any other way, such as Unicode. Of course, specific data or object values can be encoded in Unicode; we'll discuss those cases as they arise.

White-Space

The white-space characters shown in Table 1-1 are used in PDF to separate syntactic constructs such as names and numbers from each other.

Table 1-1. White-space characters

9	09	011	HORIZONTAL TAB (HT)
10	0A	012	LINE FEED (LF)
12	0C	014	FORM FEED (FF)
13	0D	015	CARRIAGE RETURN (CR)
32	20	040	SPACE (SP)

In all contexts except comments, strings, cross-reference table entries, and streams, PDF treats any sequence of consecutive white-space characters as one character.

The CARRIAGE RETURN (*0Dh*) and LINE FEED (*0Ah*) characters, also called *newline characters*, are treated as end-of-line (EOL) markers. The combination of a CARRIAGE RETURN followed immediately by a LINE FEED is treated as one EOL marker. EOL markers are typically treated the same as any other white-space characters. However, sometimes an EOL marker is required, preceding a token that appears at the beginning of a line.

The Four Sections of a PDF

Figure 1-2 illustrates the four sections of a PDF: the header, trailer, body, and cross-reference table.

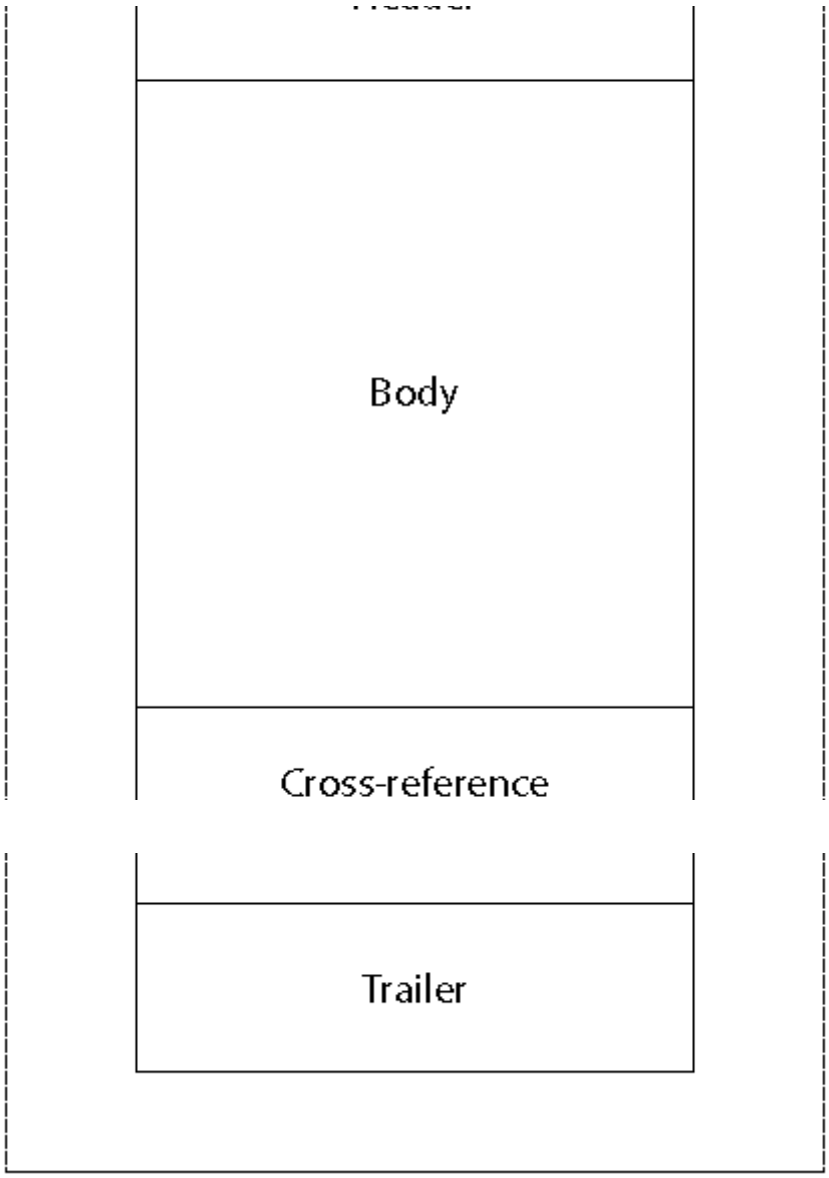

Figure 1-2. The four sections of a PDF

Header

The header of a PDF starts at byte 0 of the file and consists of at least 8 bytes followed by an end-of-line marker. These 8 bytes serve to clearly identify that the file is a PDF (*%PDF-*) and suggest a version number of the standard that the file complies with (e.g., 1.4). If your PDF contains actual binary data (and these days, pretty much all of them do) a second line will follow, which also starts with the PDF comment character, % (PERCENT SIGN). Following the % on the second line will be at least four characters

whose ASCII values are greater than 127. Although any four (or more) values are fine,

order ASCII values. Including those values ensures that PDFs will always be considered as binary.

Trailer

At the opposite end of the PDF from the header, one can find the trailer. A simple example is shown in Example 1-12. The trailer is primarily a dictionary with keys and values that provides document-level information that is necessary to understand in order to process the document itself.

Example 1-12. A simple trailer

```
trailer
<<
 /Size 23
 /Root 5 0 R
 /ID[<E3FEB541622C4F35B45539A690880C71><E3FEB541622C4F35B45539A690880C71>]
 /Info 6 0 R
```

The two most important keys, and the only two that are required, are `Size` and `Root`. The `Size` key tells us how many entries you should expect to find in the cross-reference table that precedes the trailer dictionary. The `Root` key has as its value the document's catalog dictionary, which is where you will start in order to find all the objects in the PDF.

Other common keys in the trailer are the `Encrypt` key, whose presence quickly identifies that a given PDF has been encrypted; the `ID` key, which provides two unique IDs for the document; and the `Info` key, which represents the original method of providing document-level metadata (this has been replaced, as described in Chapter 12).

Body

The body is where all the nine types of objects that comprise the actual document itself are located in the file. You will see more about this in Document Structure as you look at the various objects and how they are organized.

Cross-reference table

The cross-reference table is simple in concept and implementation, but it is one of the

how large the “numeric jump” in the page numbers is. Having the cross-reference table at the end of the file also provides two additional benefits: creation of the PDF in a single pass (no backtracking) is possible, and support for incremental updates of the document is facilitated (see Incremental Update for an example).

The original form (from PDF 1.0 to 1.4) of the cross-reference table is comprised of one or more cross-reference sections, where each of these sections is a series of entries (one line per object) with the object’s file offset, its generation, and whether it is still in use. The most common type of table, shown in Figure 1-3, has only a single section listing all objects.

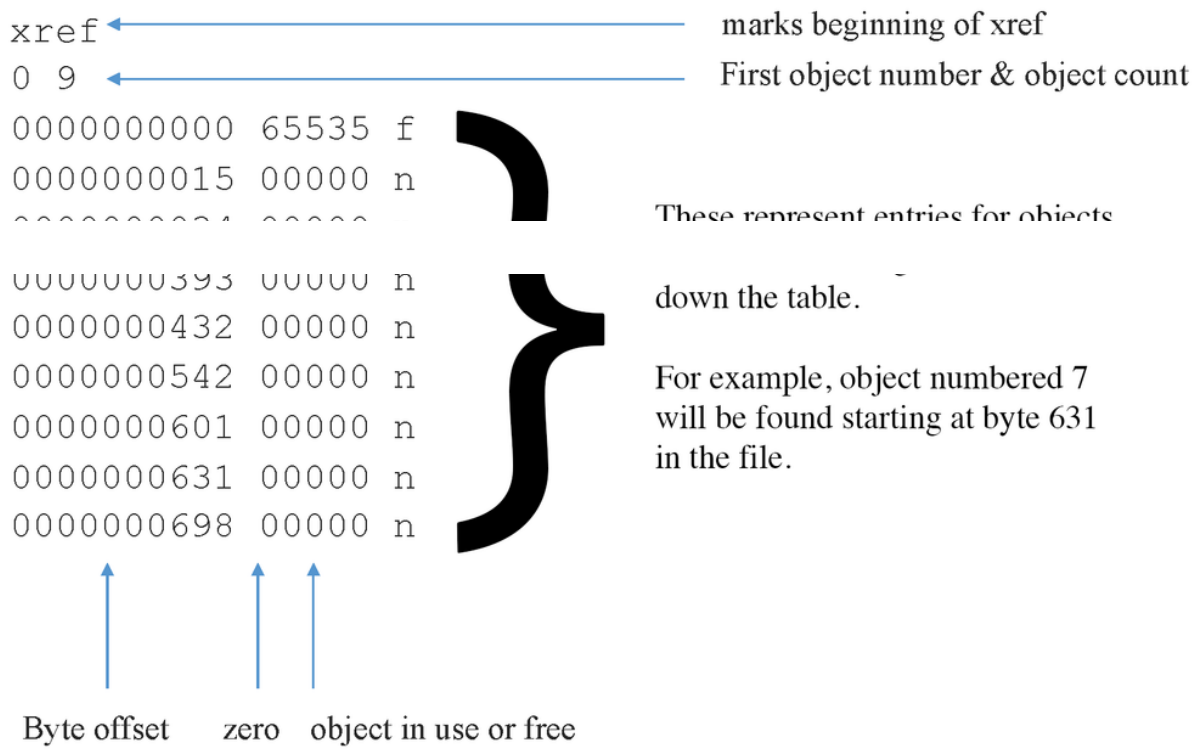

Figure 1-3. Classic cross-reference table

NOTE

This type of cross-reference table follows a very rigid format where the column positions are fixed and the zeros are required.

You may notice that the values of the numbers in the second column of each line of the

However, when a PDF contains an incremental update, you may see a cross-reference section that looks like the one in Example 1-13.

Example 1-13. Updated cross-reference section

```
xref
0 1
0000000000 65535 f
4 1
0000000000 00001 f
6 2
0000014715 00000 n
0000018902 00000 n
10 1
0000019077 00000 n
trailer
<</Size 18/Root 9 0 R/Prev 14207
/ID[<86E7D6BF23F4475FB9DEED829A563FA7><507D41DDE6C24F52AC1EE1328E44ED26>]>>
```

As PDF documents became larger, it was clear that having this very verbose (and uncompressable) format was a problem that needed addressing. Thus, with PDF 1.5, a new type of cross-reference storage system called *cross-reference streams* (because the data is stored as a standard stream object) was introduced. In addition to being able to be compressed, the new format is more compact and supports files that are greater than 10 gigabytes in size, while also providing for other types of future expansion (that have not yet been utilized). In addition to moving the cross-reference table to a stream, this new system also made it possible to store collections of indirect objects inside of another special type of stream called an *object stream*. By intelligently splitting the objects among multiple streams, it is possible to optimize the load time and/or memory consumption for the PDF. Example 1-14 shows what a cross-reference stream looks like.

Example 1-14. Inside a cross-reference stream

```
stream
01 0E8A 0      % Entry for object 2 (0x0E8A = 3722)
02 0002 00     % Entry for object 3 (in object stream 2, index 0)
02 0002 01     % Entry for object 4 (in object stream 2, index 1)
02 0002 02     % . . .
02 0002 03
02 0002 04
```

```
02 0002 05
```


Incremental Update

As mentioned earlier, one of the key features of PDF that was made possible through the use of a trailer and cross-reference table at the end of the document is the concept of *incremental update*. Since changed objects are written to the end of the PDF, as illustrated in Figure 1-4, saving modifications is very quick as there is no need to read and process every object.

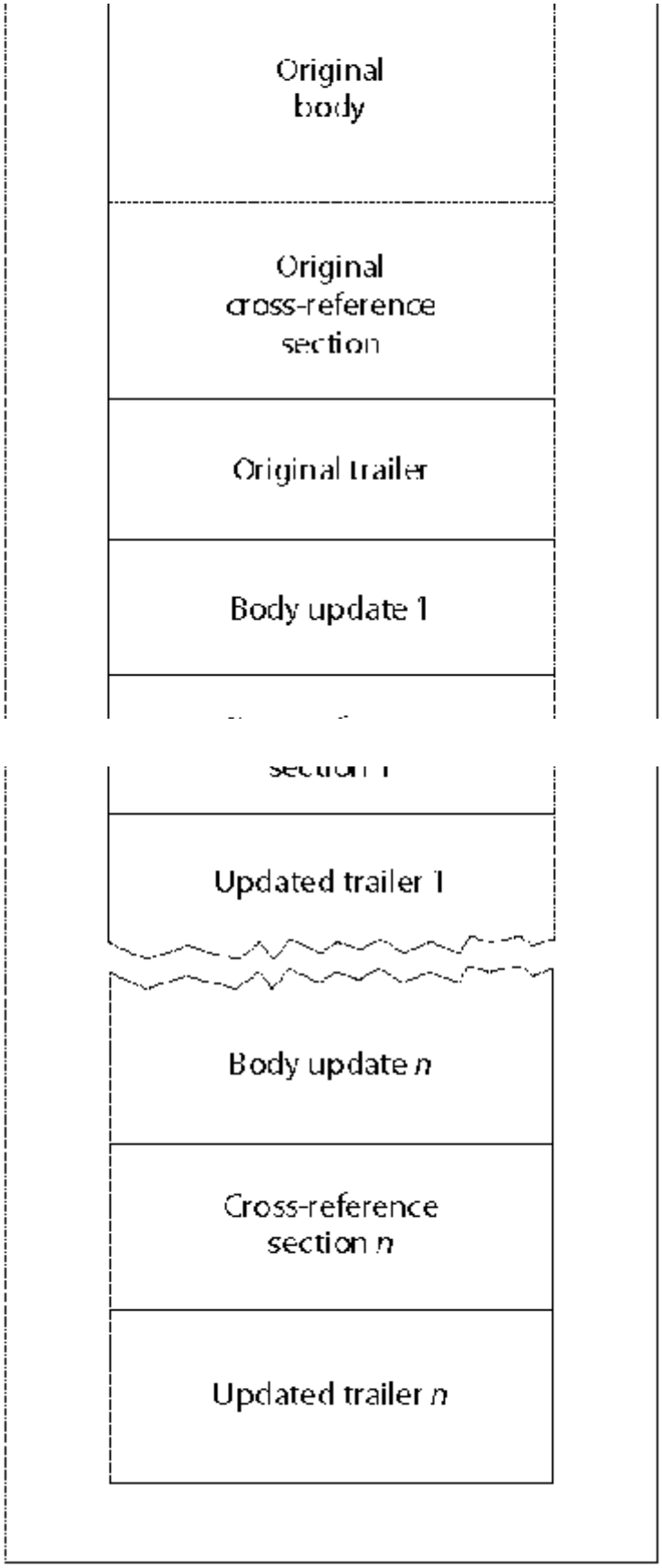

Figure 1-4. Layout of a PDF with incremental update sections

only lists the new, changed, or deleted objects in the new table, as seen in Example 1-13.

Although viewers don't actually offer this feature (except after a digital signature, like in Signature Fields, has been applied), the use of incremental updates means that it is possible to support multiple undos across save boundaries. However, that also brings dangers from people who are looking through your (uncollected) garbage. Even though you thought you deleted something from the file, it may still be there if an incremental update was applied instead of a full save.

> **NOTE**
>
> When incrementally updating a PDF, it is extremely important that you do not mix classic cross-references with cross-reference streams. Whatever type of cross-reference is used in the original must also be used in the update section(s). If you do mix them, a PDF reader may choose to ignore the updates.

Linearization

As you've seen, having the cross-reference table at the end of the file offers various advantages. However, there is also one large disadvantage, and that's when the PDF has to be read over a "streaming interface" such as an HTTP stream in a web browser. In that case, a normal PDF would have to be downloaded in its entirety before even a single page could be read—not a great user experience.

To address this, PDF provides a feature called *linearization* (ISO 32000-1:2008, Annex F), but better known as "Fast Web View."

A linearized file differs from a standard PDF in three ways:

1. The objects in the file are ordered in a special way, such that all of the objects for a particular page are grouped together and then organized in numerical page order (e.g., objects for page 1, then objects for page 2, etc.).

2. A special *linearization parameter dictionary* is present, immediately following the header, which identifies the file as being linearized and contains various information needed to process it as such.

3. A partial cross-reference table and trailer are placed at the beginning of the file

fragment of a linearized PDF is shown in Example 1-15.

Example 1-15. Linearized PDF fragment

```
%PDF-1.7
%%EOF
8 0 obj
<</Linearized 1/L 7546/O 10/E 4079/N 1/T 7272/H [ 456 176]>>
endobj
xref
8 8
0000000016 00000 n
0000000632 00000 n
0000000800 00000 n
0000001092 00000 n
0000001127 00000 n
0000001318 00000 n
0000003966 00000 n
0000000456 00000 n
trailer

<6DCD1D><068A37E2007240EF9D346D00AD08F696>]/Prev 7264>>
startxref
0
%%EOF

% body objects go here...
```

> **NOTE**
>
> Mixing linearization and incremental updates can yield unexpected results, since the linearized cross-reference table will be used instead of the updated versions, which only exist at the end of the file. Therefore, it is important that files destined for use online should be fully saved, to remove updates and ensure the correct linearization tables.

Document Structure

Now that you've learned about the various objects in the PDF and how they are put together to form the physical file layout/structure, it's time to put them together to form an actual document.

The Catalog Dictionary

content.

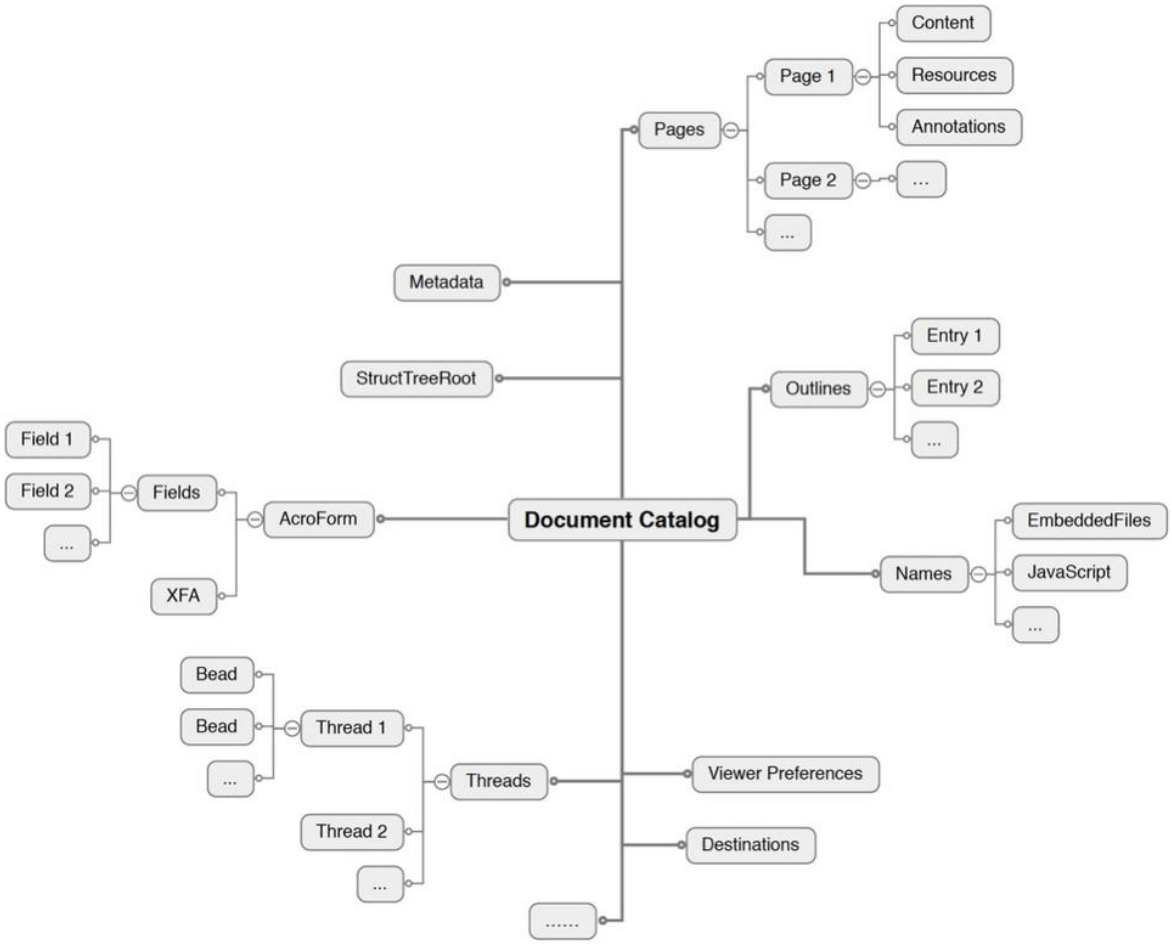

Figure 1-5. Graph-like structure of PDF objects

The `Root` is always an object of type `Catalog` and is known as the document's *catalog dictionary*. It has two required keys:

1. `Type`, whose value will always be the name object `Catalog`
2. `Pages`, whose value is an indirect reference to the page tree (The Page Tree)

While being able to get to the pages of the PDF is obviously important, there are over two dozen optional keys that can also be present (see ISO 32000-1:2008, Table 28). These represent document-level information including such things as:

- XML-based metadata (XMP)

- OpenActions (Actions)

Example 1-16 shows an example of a catalog object.

Example 1-16. Catalog object

```
<<
    /Type /Catalog
    /Pages 533 0 R
    /Metadata 537 0 R
    /PageLabels 531 0 R
    /OpenAction 540 0 R
    /AcroForm 541 0 R
    /Names 542 0 R
    /PageLayout /SinglePage
    /ViewerPreferences << /DisplayDocTitle true >>
>>
```

Let's look at a few keys (and their values) that you may find useful to include in your PDFs in order to improve the user experience:

`PageLayout`
: The `PageLayout` key is used to tell the viewer how to display the PDF pages. Its value is a name object (see Name Objects). To display them one at a time, use a value of `SinglePage`, or if you want the pages all in a long continuous column, use a value of `OneColumn`. There are also values that can be specified for two pages at a time (sometimes called a *spread*), depending on where you want the odd-numbered pages to fall: `TwoPageLeft` and `TwoPageRight`.

`PageMode`
: In addition to just having the PDF page content displayed, you may wish to have some of the navigational elements of a PDF immediately accessible to the user. For example, you might want the bookmarks or outlines visible (see Bookmarks or Outlines for more on these). The value of the `PageMode` key, which is a name object, determines what (if any) extra elements are shown, such as `UseOutlines`, `UseThumbs`, or `UseAttachments`.

`ViewerPreferences`

Unlike the previous two examples, where the values of the keys were name

Chapter 12) is shown in the previous example: `DisplayDocTitle`. Having that present with a value of `true` instructs a PDF viewer to display not the document's filename in the title bar of the window, as shown in Figure 1-6, but rather its real title, as shown in Figure 1-7.

Figure 1-6. Window title bar showing the filename

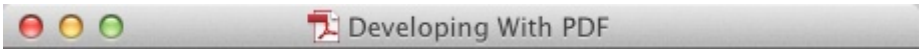

Figure 1-7. Window title bar showing document title

The Page Tree

The pages in a PDF are accessed through the page tree, which defines the ordering of the pages. The page tree is usually implemented as a balanced tree but can also be just a simple array of pages.

> **NOTE**
>
> It is recommend that you have no more than about 25–50 pages in a single leaf of the tree. This means that any document larger than that should not be using a single array, but instead should be building a balanced tree. The reason for doing so is that the design of a balanced tree means that on devices with limited memory or resources, it is possible to find any specific page without having to load the entire array and then sequentially access each page in the array.

As you can see in Figure 1-8, there are two types of nodes in the page tree: intermediate nodes (of type `Pages`) and terminal or leaf nodes (of type `Page`). Intermediate nodes, which include the starting node of the tree, provide indirect references to their parents (if any) and children, along with a count of the leaf nodes in their particular branches of the tree. The leaf node is the actual `Page` object.

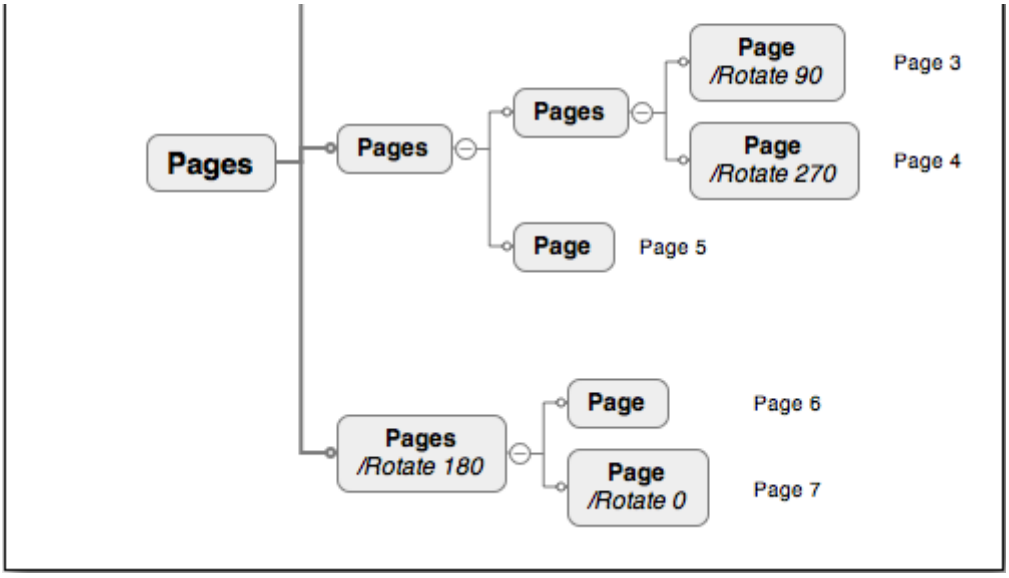

Figure 1-8. Image of a page tree

A portion of the Figure 1-8, represented in PDF syntax might look like Example 1-17.

Example 1-17. Objects making up a sample page tree

```
2 0 obj
<<
    /Type /Pages
    /Kids[ 4 0 R ]
    /Count 3
>>
endobj

4 0 obj
<<
    /Type /Pages
    /Parent 2 0 R
    /Rotate    90
    /Kids[ 5 0 R  6 0 R ]
    /Count 3
>>
endobj

5 0 obj
<<
    /Type /Page
    % Additional entries describing the attributes of Page 1
>>
```

```
endobj
```



```
endobj
```

Pages

As just discussed, each leaf node in the page tree represents a page object. The page object is a dictionary whose `Type` key has a value of `Page`; it also contains a few other required keys and may contain a dozen or more optional keys and their values.

Example 1-18 shows a few sample page dictionaries.

Example 1-18. Two sample page dictionaries

```
% simplest valid page object, with the four required keys
<<
    /Type /Page
    /Parent 2 0 R
    /MediaBox [ 0 0 612 792 ]    % Page Size == 8.5 x 11 in (612/72 x 792/72)
    /Resources <<>>
>>


% a real-world page object
<<
    /Type /Page
    /Parent 532 0 R
    /MediaBox [ 0 0 612 792 ]
    /CropBox [ 0 0 500 600 ]
    /Contents 564 0 R
    /Resources <<
        /ExtGState << /GS0 571 0 R /GS1 572 0 R >>
        /Font << /T1_0 566 0 R >>
        /XObject << /Im0 577 0 R >>
    >>
    /Trans << /S /Dissolve >>
    /Rotate 90
    /Annots 549 0 R
    /AA << /C 578 0 R /O 579 0 R >>
>>
```

There are a few keys to point out here, some of which we will delve into in future chapters:

Content

`Rotate`
: This key can be used to rotate the page in increments of 90 degrees. However, while a proper and valid part of PDF, it is frequently ignored by many lower-end tools. Therefore, consider using properly sized pages and (if necessary) transformed content, as described in Transformations.

`Trans`
: If present, this key tells a viewer that when displaying the page in a "presentation style," it should use the defined transition when moving to this page from the one that precedes it. Details of the values for this key can be found in ISO 32000-1:2008, 12.4.4.

`Annots`
: The value of this key is an array of all of the annotations (see Chapter 6) that are present on top of the page's content.

`AA`
: Actions represent things that the viewer will do upon implicit actions by the user, such as opening or closing a page (see Actions for more).

`Resources`
: These are used to help complete the definitions of graphic objects, such as the font or color to use, that are necessary in order to draw on a page. They will be presented in the next chapter.

PDF units

Often when you work with graphic systems, you are working directly at the resolution of the output device, such as a 72 or 90 dpi (dots per inch) screen or a 600 dpi printer. This is referred to as *device space* (Figure 1-9).

Figure 1-9. Device space

However, as this figure shows, if you want the same-sized object to appear regardless of the device's characteristics, you need to work in something other than device space. With PDF, that is called *user space*, and it stays the same regardless of the output device (Figure 1-10).

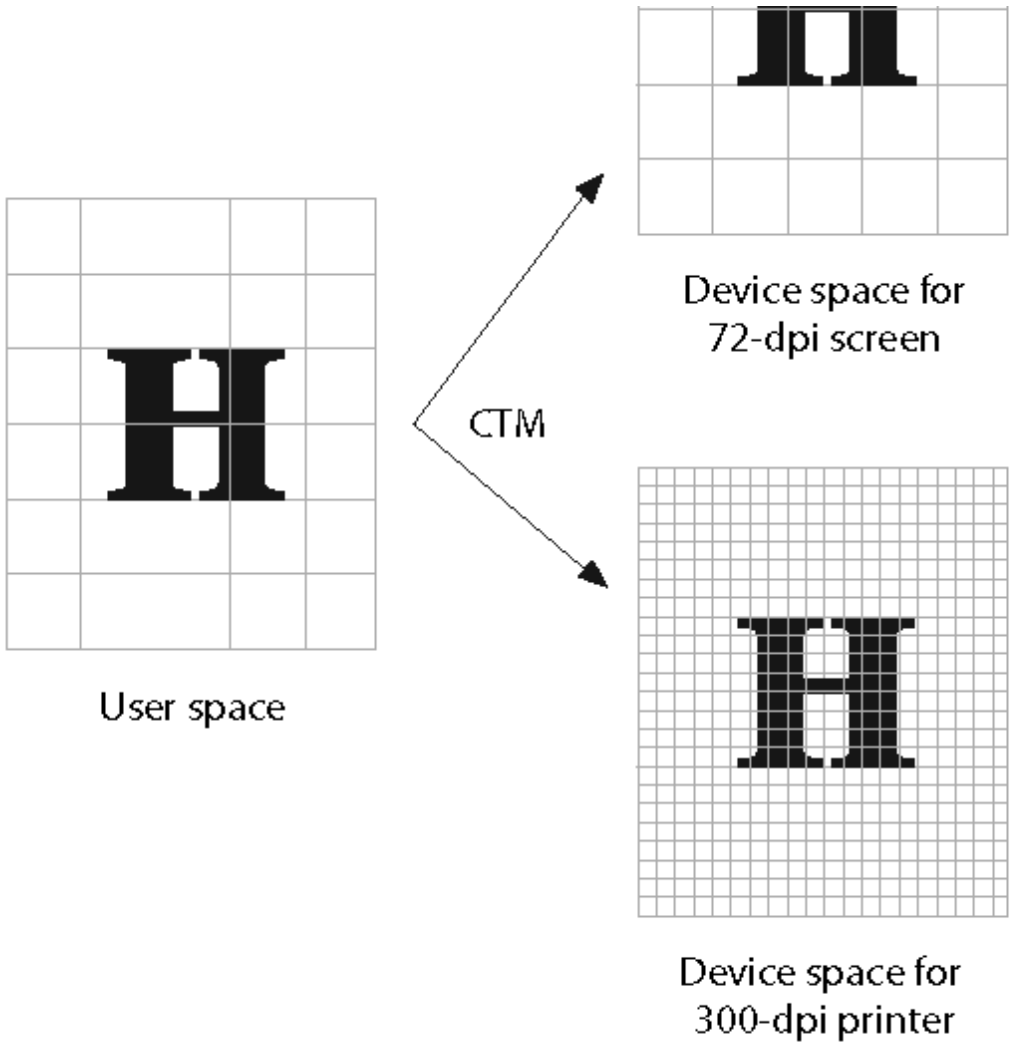

Figure 1-10. User space

User space defaults to 72 user units per inch (aka “points”), with the origin at the bottom left. It is possible to change the number of *user units* either through the use of a coordinate transform in the page content (see Transformations) or the presence of a `UserUnit` key in the page dictionary (as illustrated in Example 1-19). The origin of the coordinate system will always be [0 0], but that may not correspond to the bottom-left corner of the visible PDF page, depending on the values of the page boxes (see Rects and boxes).

Example 1-19. Example pages that use the UserUnit key

```
2 0 obj
```



```
endobj

3 0 obj
<<
    /Type /Page
    /Parent 2 0 R
    /UserUnit    1.0             % default of 72 units/inch
    /MediaBox [ 0 0 612 792 ]    % 8.5 x 11 inches
    % more keys here...
>>
endobj

4 0 obj
<<
    /Type /Page
    /Parent 2 0 R
    /UserUnit    2.0             % 144 units/inch (2 * 72)
    /MediaBox [ 0 0 612 792 ]    % 17 x 22 inches
    % more keys here...
>>
endobj

5 0 obj
<<
    /Type /Page
    /Parent 2 0 R
    /UserUnit    3.14159         % something funny but perfectly valid
    /MediaBox [ 0 0 612 792 ]    % 26.70 x 34.56 inches
    % more keys here...
>>
endobj
```

Rects and boxes

When describing a rectangle in PDF syntax, an array of four numbers is used. The order of the numbers is: left, bottom, width, height. You will find rects used in various places in PDF syntax, but the type of rect that you will be using most frequently is to define the sizes of various regions on a page—the *page boxes*.

Each of the five page boxes (ISO 32000-1:2008, 14.11.2) represents a rectangular viewing area (a “box”) for the graphic elements that are drawn on the page, either directly or via annotations. The four numbers in the array are always in user units, the units of user space (see Figure 1-10. Since it represents a view into the coordinate system of the page, the rectangle need not have its bottom-left corner at [0 0].

The `MediaBox` of a page defines the size of the page on which the drawing will take

> A `MediaBox` of `[0 0 612 792]`, in default (1.0) user units, is equivalent to a US Letter–sized piece of paper (8.5*72 = 612; 11*72 = 792).

In addition to the `MediaBox`, there are four other page boxes that may appear on a page. They are shown in Figure 1-11.

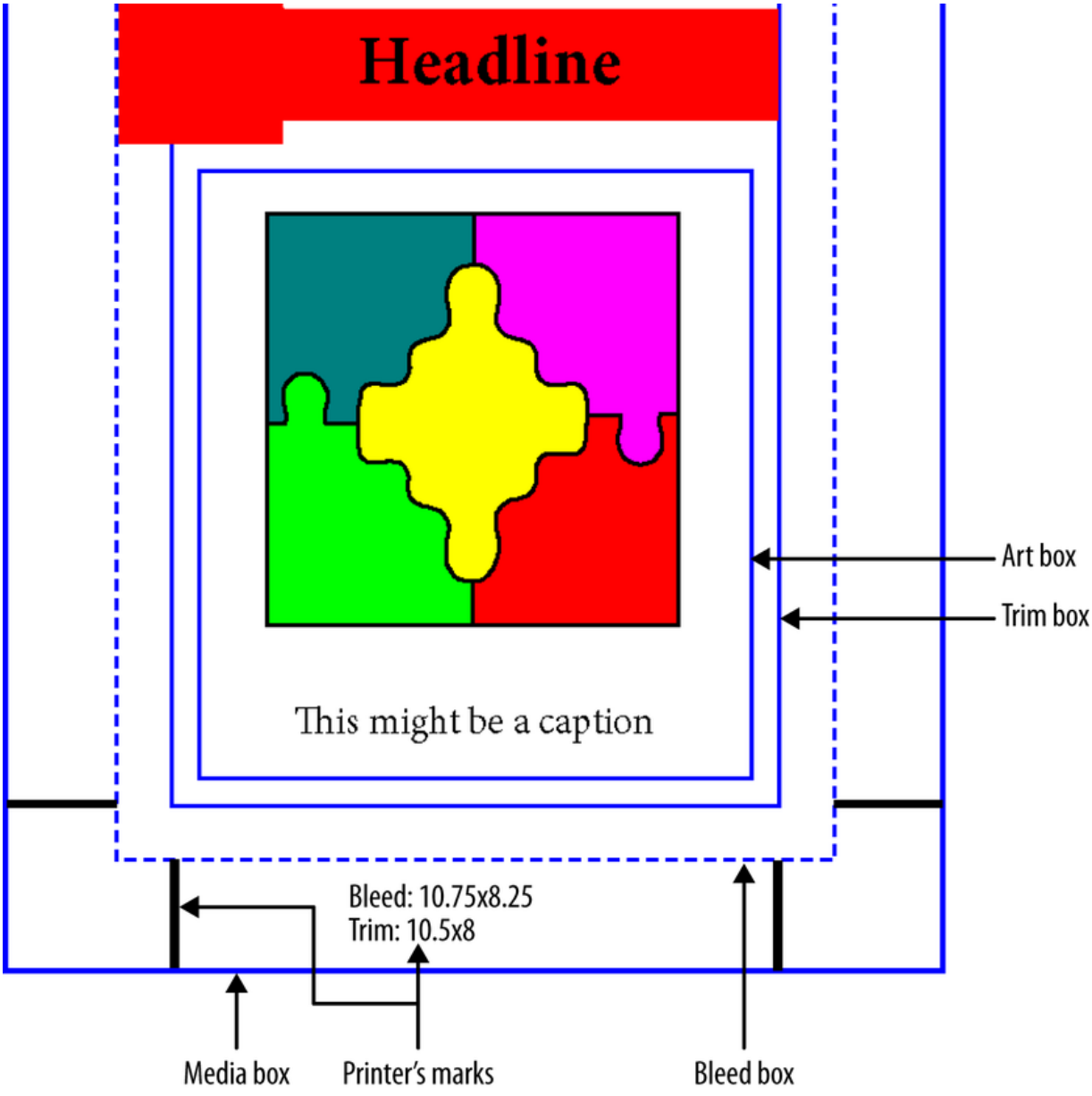

Figure 1-11. Page with boxes

A `CropBox` is used to instruct a PDF viewer of the actual visible area of the page when it is displayed or printed. This is primarily used when you have content on a page that you don't want a user to see, so you "crop" it out. Unlike in an image editor, applying a `CropBox` doesn't remove anything; it simply hides it outside the visible area.

NOTE

In the printing industry, a `TrimBox` serves a somewhat similar purpose. It defines where a cutter will trim (cut) the paper after it's been printed, thus removing content outside of the `TrimBox` from the final piece. It is used when you have something you want to come right up to the edge of the paper, without any white space or gap. For this to work, there is almost always a related `BleedBox`, which defines the area outside of the `TrimBox` where the content "bleeds" out so that it can be properly trimmed.

The final box, called the `ArtBox`, is almost never used. It was originally supposed to be used to represent an area that covered just the "artwork" of the page that one might use to repurpose, placing or imposing it onto another sheet. However, it never really caught on, and you should simply not bother using them in your documents.

Inheritance

As you saw in Pages, some of the values that would normally be present in a `Page` object can also be present in the intermediate nodes (`Pages` objects). When this happens, those values are inherited by all of the children of that node, unless they choose to override them. For example, if all the pages of a document are the same size, you could put the `MediaBox` key in the root node of the page tree.

Not all of the keys that can be present in a page object can be inherited, only those identified as such in ISO 32000-1:2008, Table 30.

NOTE

A linearized PDF cannot use inheritance. All values must be specified in each page object directly.

The Name Dictionary

Some types of objects in a PDF file can be referred to by name rather than by object reference. This correspondence between names and objects is established by using something called a document's *name dictionary*. The name dictionary is specified by including a `Names` key in the document's catalog dictionary (see The Catalog Dictionary). Each of the defined keys that can be present in this dictionary designates the root of a name tree that defines the names for that particular category of objects. Some of the types of objects that can be referenced by name are listed in Table 1-2:

Table 1-2. Some name dictionary entries

	Destinations)
AP	Appearance streams for annotations (Appearance Streams)
JavaScript	JavaScript files
EmbeddedFiles	Embedded files (The EmbeddedFiles Name Tree)

What's Next

In this chapter you learned about the basic syntax of PDF, starting from the basic types of objects and moving to the structure of a PDF file. You also learned about how these objects come together to form the document and its pages, and what keys can be found in a page object.

Next you'll learn about the PDF imaging model, content streams, and how to actually draw things on a page.

```
<othercredit role="interiordesigner">
  <firstname>David</firstname>
  <surname>Futato</surname>
</othercredit>
<othercredit role="illustrator">
  <firstname>Rebecca</firstname>
  <surname>Demarest</surname>
</othercredit>
<othercredit role="coverdesigner">
  <firstname>Randy</firstname>
  <surname>Comer</surname>
</othercredit>
```

```
<!-- All rights reserved. -->
```

```
<publisher>
```



```
    <postcode>95472</postcode>
  </address>
</publisher>
```

```
<legalnotice role="printlocation">
  <para>Printed in the United States of America.</para>
</legalnotice>
```

```
<legalnotice role="printer">
  <para>[LSI]</para>
</legalnotice>
```

```
<legalnotice role="use">
  <para>O’Reilly books may be purchased for educational, business, or sales p
        titles (<ulink role="orm:hideurl:ital" url="http://my.safaribooksonli
        800-998-9938 or <phrase role="keep-together"><email>corporate@oreilly
</legalnotice>
```

```
<legalnotice role="trademarks">
  <para>Nutshell Handbook, the Nutshell Handbook logo, and the O’Reilly logo
        are registered trademarks of O’Reilly Media, Inc. <emphasis>Developin
        and related trade dress are trademarks of O’Reilly Media, Inc.    </p
  <para>Many of the designations used by manufacturers and sellers to disting
        their products are claimed as trademarks. Where those designations ap
        in this book, and O’Reilly Media, Inc., was aware of a trademark clai
        the designations have been printed in caps or initial caps.    </para
</legalnotice>
```

```
  <legalnotice role="damages">
    <para>While every precaution has been taken in the preparation of this bo
        the publisher and author assume no responsibility for errors or omiss
        or for damages resulting from the use of the information contained he
  </legalnotice>
<isbn>9781449327910</isbn><edition>1</edition><author><firstname>Leonard</fir
```

<mediaobject role="cover"> <!-- source in the cover --> <imageobject role="front-