

California Condor

The California Condor (*Gymnogyps californianus*) is a species of North American animal in the New World vulture family Cathartidae. Currently, this condor inhabits only the Grand Canyon area and western coastal mountains of California and northern Baja California. Although other fossil members are known, it is the only surviving member of the genus *Gymnogyps*.

It is a large, white vulture with patches of white on the underside of the wings and a largely bald head with skin color ranging from yellowish to a bright red, depending on the bird's mood. It has the largest wingspan of any bird found in North America and is one of the heaviest. The condor is a scavenger and eats large amounts of carrion. It is one of the world's longest-living birds, with a lifespan of up to 50 years.

Turbo Pascal

Turbo Pascal is a complete software development system that includes a compiler and an Integrated Development Environment (IDE) for the Pascal programming language running under CP/M, MS-DOS and CP/M-86, developed by Borland under Philippe Kahns leadership. The name Borland Pascal was generally reserved for the high end packages (with more libraries and standard library source code) while the original cheap and widely known version was sold as Turbo Pascal. The name Borland Pascal is also used more generically for Borlands dialect of Pascal.

Borland has released three versions of Turbo Pascal of historical interest free of charge: versions 1.0, 3.02 and 5.5 for MS-DOS.

Table of Contents

1. PDF Syntax.....	1
PDF Objects	1
Null Objects	1
Boolean Objects	2
Numeric Objects	2
Name Objects	3
String Objects	4
Array Objects	5
Dictionary Objects	5
Stream Objects	7
Direct versus Indirect Objects	8
File Structure	10
White-Space	13
The Four Sections of a PDF	14
Incremental Update	18
Linearization	20
Document Structure	21
The Catalog Dictionary	21
The Page Tree	24
Pages	26
The Name Dictionary	32

TRIAL VERSION

4. Text.....	63
Fonts	63
Glyphs	63
Font Types	65
The Font Dictionary	66
Encodings	69
Text State	71
Font and Size	71
Rendering Mode	73
Drawing Text	74
Positioning Text	75

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*.



COS stands for Carouzel 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 synonymous with a missing value, which is why it's extremely rare to see one in a PDF. If you have reason to work with the null value, be sure to consult ISO 32000 carefully about the subtleties involving its handling.

Boolean Objects

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



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 capitalization 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 implementation by any vendor).



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 non-decimal radices.



While the term “real” is used in PDF to represent the object type, the actual implementation of a given 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).



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  
/ThisIsName37  
/Lime#20Green  
/SSCN_SomeSecondClassName
```

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

PDFDocEncoded

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



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 readability. **Example 1-4** shows a few examples of strings in PDF.

Example 1-4. Strings

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



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” on page 151](#).)



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

```
[ 0 0 612 792 ]           % 4-element array of all integers
[ (T) -20.5 (H) 4 (E) ]   % 5-element array of strings, reals, and integers
[ [ 1 2 3 ] [ 4 5 6 ] ]  % 2-element array of 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.



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 higher-level object that is being authored.

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.



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
  /Author (Leonard Rosenthal)
  /Resources << /Font [ /F1 /F2 ] >>
>>

% a dictionary with all white-space stripped out
<</Length 3112/Subtype/XML/Type/Metadata>>
```

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 and can be compressed or encoded. As such, they are the object type used to store large blobs of data that are in some other standardized format, such as XML grammars, font files, and image data.

A stream object is represented by the data for the object preceded by a dictionary 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 compression technology used by the ZIP file format. For images, the two most common 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 stream 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 that these objects can be represented either directly or indirectly in the PDF.

Direct objects are those objects that appear “inline” and are obtained directly (hence the name) when the objects are being read from the file. They are usually found as the value of a dictionary key or an entry in an array and are the type of object that you've 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.



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
```

When you refer to an indirect object, you do so using its ID, its generation, and the character R. For example, it's quite common to see something like [Example 1-10](#), where two indirect objects (IDs 4 and 5) are referenced.

Example 1-10. An indirect object that references other indirect objects

```
3 0 obj          % object ID 3, generation 0
<<
  /ProcSet 5 0 R      % reference the indirect object with ID 5, generation 0
  /Font <</F1 4 0 R >> % reference the indirect object with ID 4, generation 0
>>
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.



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**.

TRIAL VERSION

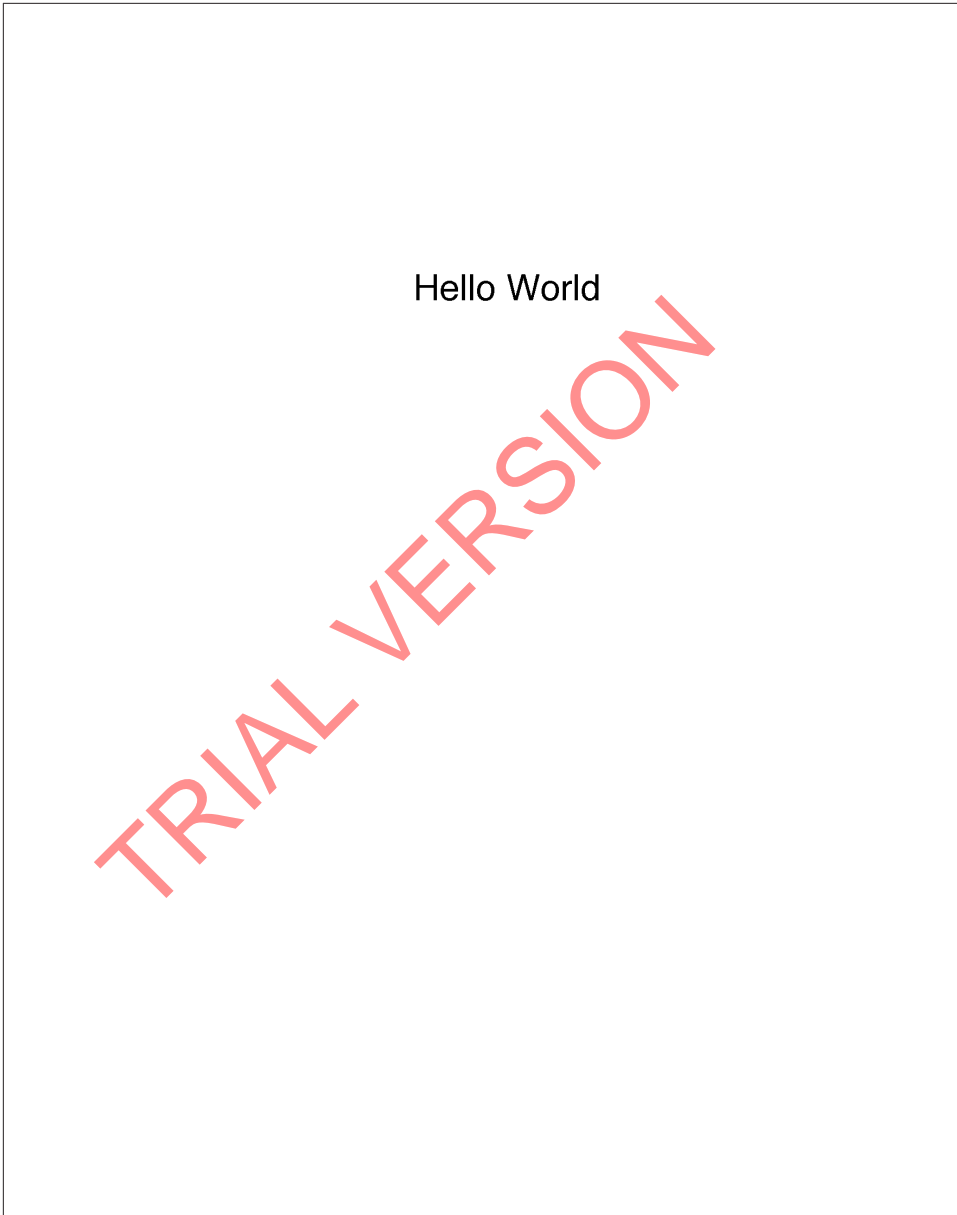


Figure 1-1. Hello World.pdf

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

6 0 obj
<<
  /Type /Catalog
  /Pages 5 0 R
>>
endobj

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 53
>>
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
0000000000 65535 f
0000000060 00000 n
0000000228 00000 n
0000000424 00000 n
0000000145 00000 n
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](#).)



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

Decimal	Hexadecimal	Octal	Name
0	00	000	NULL (NUL)
9	09	011	HORIZONTAL TAB (HT)

Decimal	Hexadecimal	Octal	Name
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.

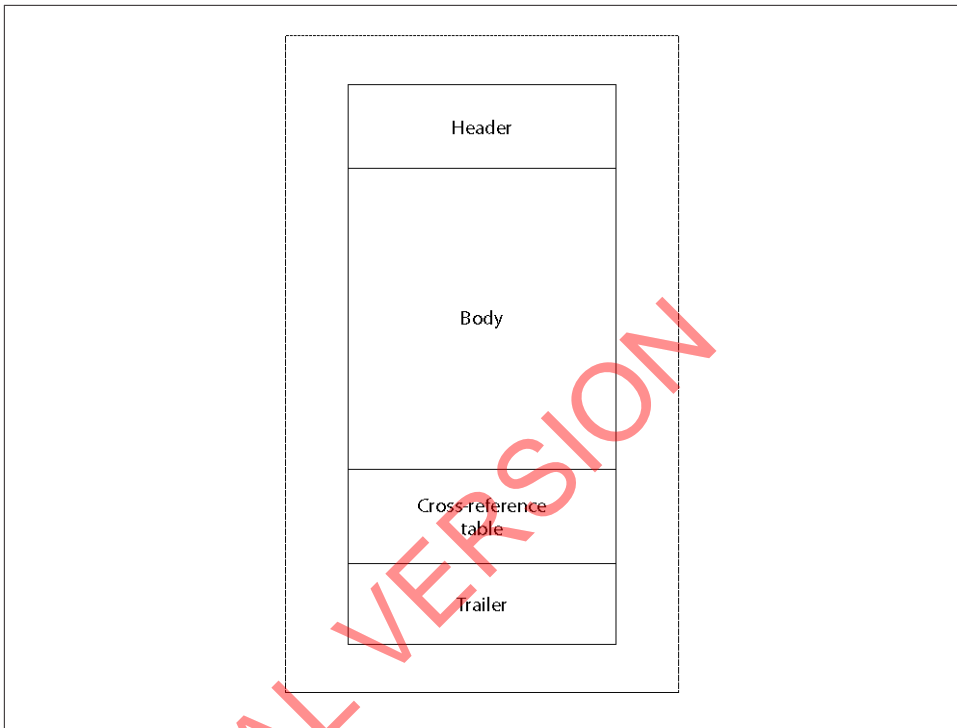


Figure 1-2. The four sections of a PDF

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, the most commonly used are `âãĴ` (0xE2E3CFD3).



The second line is there to trick programs that do ASCII vs. binary detection by simply counting high-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” on page 21](#) 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 core attributes of PDF. This table provides the binary offset from the beginning of the file for each and every indirect object in the file, allowing a PDF processor to quickly seek to and then read any object at any time. This model for random access means that a PDF can be opened and processed quickly, without having to load the entire document into memory. Additionally, navigation between pages is quick, regardless of 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” on page 18](#) 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.

xref				
0	9			
0000000000	65535	f		
0000000015	00000	n		
0000000034	00000	n		
0000000393	00000	n		
0000000432	00000	n		
0000000542	00000	n		
0000000601	00000	n		
0000000631	00000	n		
0000000698	00000	n		

Annotations:

- xref ← marks beginning of xref
- 0 9 ← First object number & object count
- These represent entries for objects numbered 0 through 8 as we read down the table.
- For example, object numbered 7 will be found starting at byte 631 in the file.
- Byte offset
- zero
- object in use or free

Figure 1-3. Classic cross-reference table



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 cross-reference table are always zero, except for the first one, which is 65535. That value, combined with the f, gives the clear indication that the object with that ID is not valid. Since a PDF file may never have an object of ID 0, that first line always looks the way you see it in this example.

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 uncompressible) 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
02 0002 06
02 0002 07     % Entry for object 10 (in object stream 2, index 7)
01 1323 0      % Entry for object 11 (0x1323 = 4899)
endstream

```

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.

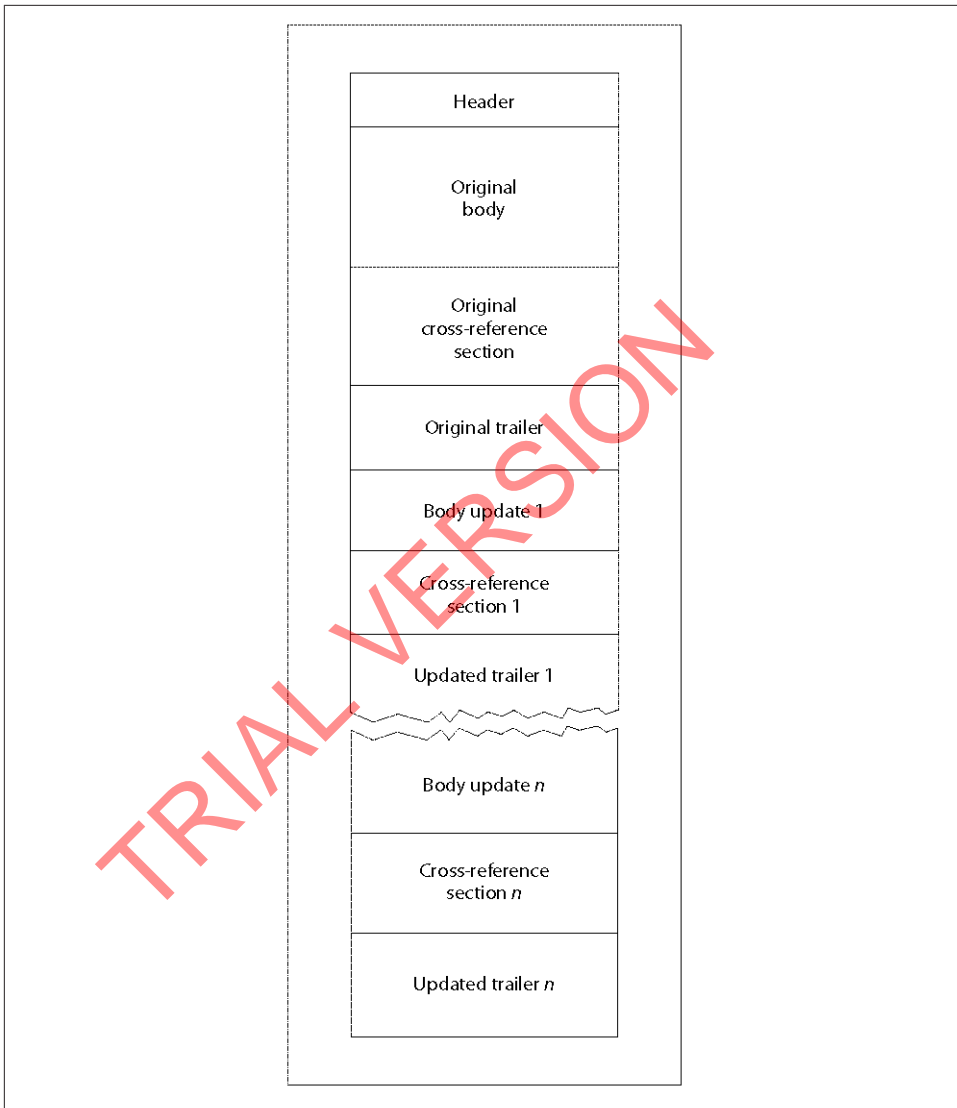


Figure 1-4. Layout of a PDF with incremental update sections

Each cross-reference section after the first points backward to the cross-reference section that preceded it via the Prev key in the trailer dictionary (see “Trailer” on page 16), and then 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” on page 119, 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.



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 to enable access to all objects needed by the Root object, plus those objects representing the first page to be displayed (usually 1).

Of course, as with a standard PDF, objects are still referenced in the same way, continuing to enable random access to any object through the cross-reference table. A 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
<</Size 16/Root 9 0 R/Info 7 0 R/ID[<568899E9010A45B5A30E98293
C6DCD1D><068A37E2007240EF9D346D00AD08F696>]/Prev 7264>>
startxref
0
%%EOF

% body objects go here...

```



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

A PDF document is a collection of objects, starting with the Root object (Figure 1-5). The reason that it is called the root is that if you think of the objects in a PDF as a tree (or a directed graph), this object is at the root of the tree/graph. From this object, you can find all the other objects that are needed to process the pages of the PDF and their content.

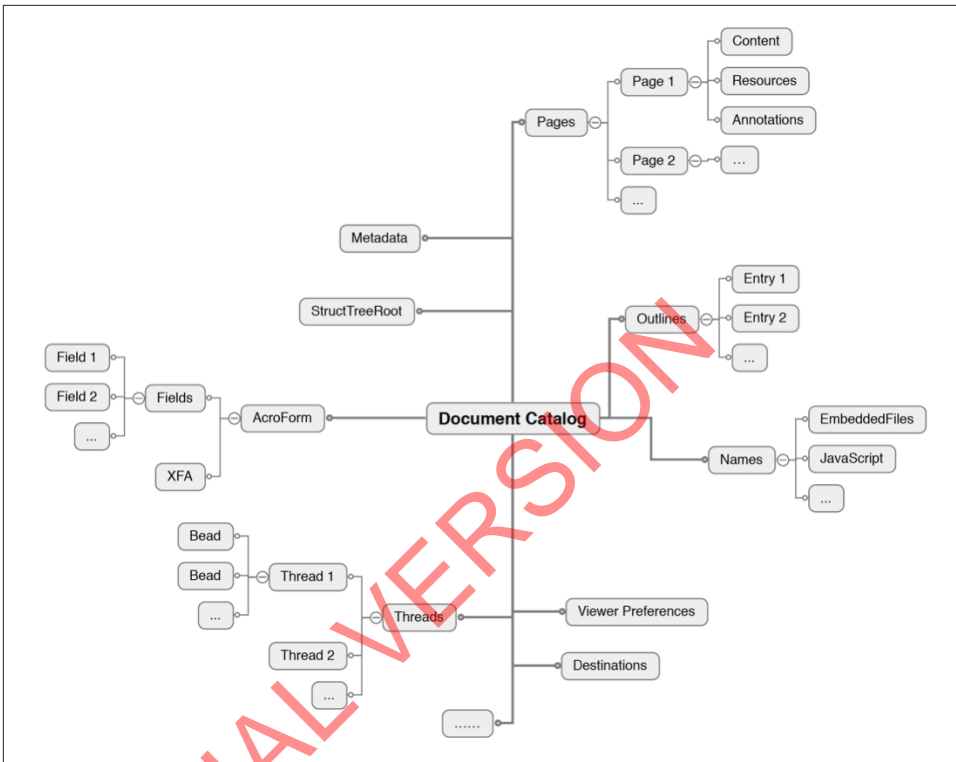


Figure 1-5. Graph-like structure of PDF objects

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” on page 24](#))

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” on page 179](#))
- OpenActions ([“Actions” on page 79](#))
- Fillable forms ([Chapter 7](#))
- Optional content ([Chapter 10](#))

- Logical structure and tags ([Chapter 11](#))

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” on page 3](#)). 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” on page 83](#) 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 objects, the ViewerPreferences key has a value that is a viewer preferences dictionary (see ISO 32000-1:2008, 12.2). Among the many keys available for use in the viewer preferences dictionary, the most important one to use (provided you add metadata to your document, as discussed in [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](#).

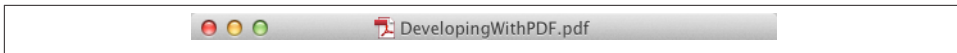


Figure 1-6. Window title bar showing the filename

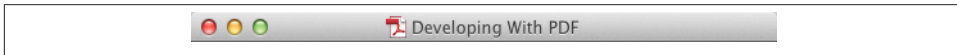


Figure 1-7. Window title bar showing document title

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.



It is recommended 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.

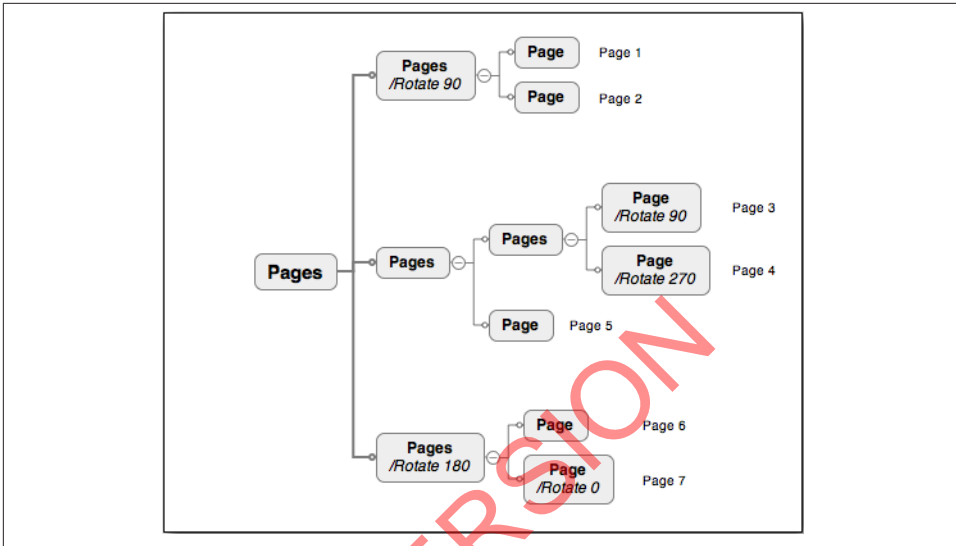


Figure 1-8. Image of a page tree

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
```

```

6 0 obj
<<
  /Type /Page
  % Additional entries describing the attributes of Page 2
>>
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

Unless you want blank pages in your PDF, this is the most important key in the page dictionary as it points to a content stream containing the instructions for what to draw on the page (see **“Content Streams” on page 35**).

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](#)” on page 42.

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](#)” on page 79 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](#)).

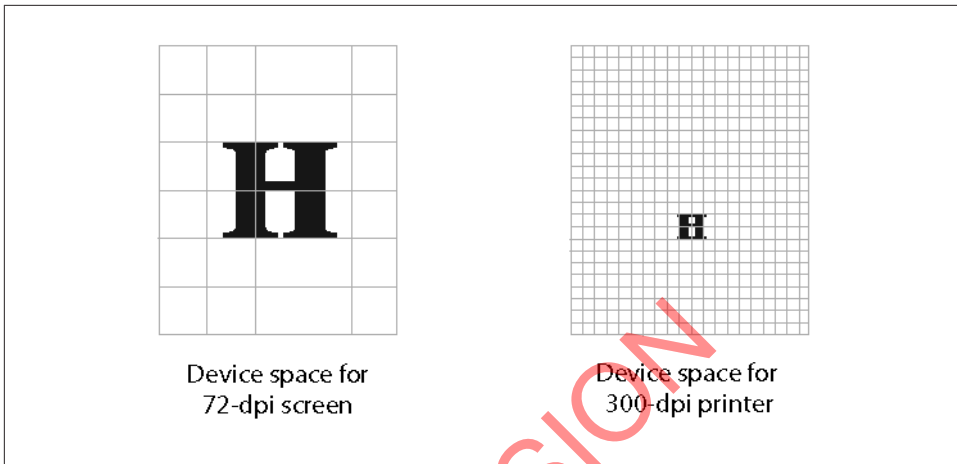


Figure 1-9. Device space

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).

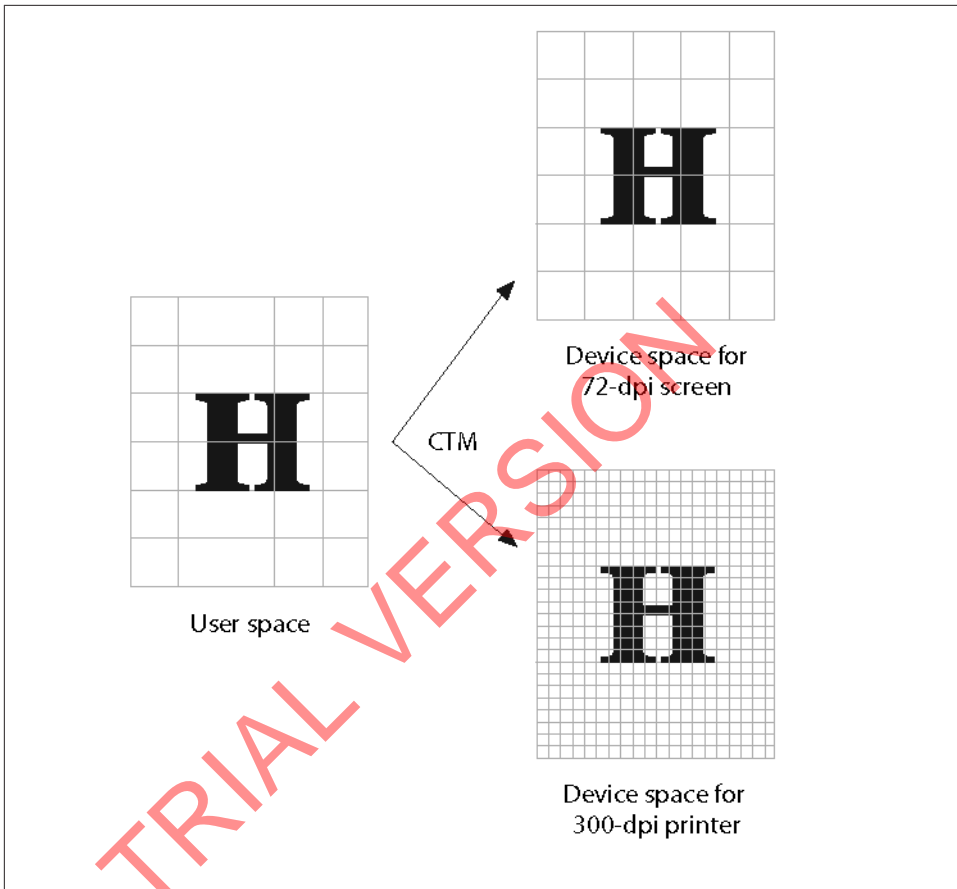


Figure 1-10. User space

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](#)” on page 42) 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](#)” on page 30).

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

```
2 0 obj
<<
  /Type /Pages
  /Kids[ 3 0 R 4 0 R 5 0 R ]
  /Count 3
```

```

>>
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

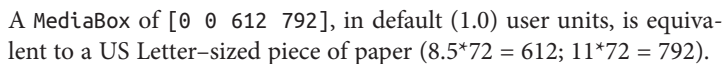
```

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 place. Normally this is equivalent to a common paper size, such as *US Letter* (8.5 x 11 inches) or *A4* (21 x 29.7 cm), although it can be any size.



The diagram illustrates a page layout with several key components and dimensions:

- Media box:** The outermost blue border representing the physical page size.
- Printer's marks:** Black registration marks (crosshairs) located at the corners of the page.
- Bleed box:** A dashed blue line indicating the area where content can extend to the edge of the page.
- Art box:** A solid blue line defining the primary content area.
- Trim box:** A dashed blue line indicating the final size of the page after trimming.
- Headline:** A red rectangular area at the top containing the word "Headline".
- Image:** A central graphic composed of four colored puzzle pieces (teal, magenta, green, red) surrounding a yellow cross-like shape.
- Caption:** The text "This might be a caption" located below the image.
- Dimensions:**
 - Bleed: 10.75x8.25
 - Trim: 10.5x8

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.



Although the CropBox may extend beyond the MediaBox, a PDF viewer will effectively pin the values to those of the MediaBox.

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" on page 26, 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.



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" on page 21). 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

Key	Object type
Dests	Named destinations (“Named Destinations” on page 78)
AP	Appearance streams for annotations (“Appearance Streams” on page 88)
JavaScript	JavaScript files
EmbeddedFiles	Embedded files (“The EmbeddedFiles Name Tree” on page 127)

TRIAL VERSION

In this chapter you will learn how to draw text on a page. Drawing text is the most complex part of PDF graphics, but it is also what helped PDF beat its competitors to become the international standard that it is today. While the other original players converted text to raster images or vector paths (to maintain the visual integrity), the inventors of PDF knew that users needed text that could be searched and copied and didn't just look pretty on the screen. With the depth of experience and understanding of fonts that Adobe's engineers had, they were able to integrate actual text with visual presentation.

While the text support in PDF enables the rendering of any glyphs from any font representing any language, the mechanics (as you'll see shortly) were all created prior to **Unicode**. This means that many things that developers working in other file formats take for granted, such as just putting down Unicode codepoints and letting the renderer do all the hard work, have to be done manually with PDF.

Now that you've been given fair warning, let's start!

Fonts

In the previous chapters you learned how to draw vector graphics (or paths) as well as raster graphics (images) on a page. These types of drawing operations are fairly simple as they don't normally need extra information—it's just the instructions and (in the case of raster images) the image data. Text, however, requires more pieces. The most important of these pieces is the font.

Glyphs

A font, sometimes called a *font program*, is a collection of unique drawing instructions called *glyphs*. In general, each glyph is no different from the paths or rasters that you

drew yourself in previous chapters. However, the font also contains a bunch of metadata about the various glyphs, including something called an *encoding* that provides a mapping from a known character set (such as ASCII, Unicode, or Shift-JIS) to the glyphs. Not every font has glyphs for every value in every encoding.

Figure 4-1 is an example of three common values in different encodings and their glyphs in different fonts.

	Arial	Mistal	Kozuka Mincho
ASCII 0x61	a	a	a
Unicode 0x03A6	Φ	□	Φ
Shift_JIS 0x8C91	□	□	倦

Figure 4-1. Example of glyphs in three different fonts



The glyphs that look like rectangles are commonly known as “not-def” glyphs. Notdef (short for *not defined*) is a special glyph present in all fonts for the specific purpose of filling in when the requested glyph doesn’t exist. If you see these in your PDFs, you know you did something wrong! (Though in this case, it’s on purpose.)

While a glyph is primarily concerned with its shape and appearance, it also has a set of values associated with it called *glyph metrics*. These metrics, some of which are explicitly defined in the font itself while others are computed from a series of values, enable software to do such things as lay out text. For example, to draw “Hello” you need to know where to place the “e” after the “H.” **Figure 4-2** tells you to simply place the “e” at the “Next glyph origin.”

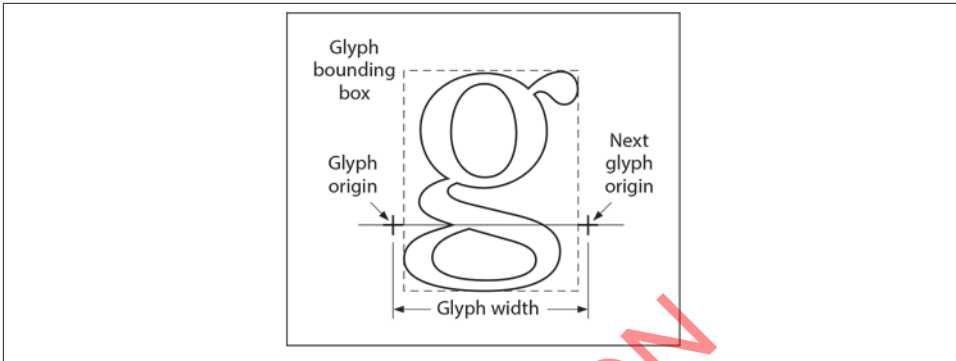


Figure 4-2. Various metrics shown around a glyph

Font Types

When graphical interfaces such as the original Lisa and Macintosh first came into existence, the fonts that came with them were known as “bitmapped fonts,” because each glyph was described as a bitmap or raster image. While quite useful for the screen, this wasn’t flexible enough for printing, so the outline (path-based) font formats were born. PDF supports the three most common outline formats:

Type 1

This was the original outline format created by Adobe along with the Postscript language for printers. The glyph outlines are described using a simplified version of Postscript.

TrueType

Created by Apple and Microsoft for their operating systems, this is the most well known of the font formats. Glyph outlines are described using a special language that is unique to this font format.

OpenType

While Type 1 and TrueType fonts each have their advantages, the industry grew tired of the “font wars,” so OpenType was born. OpenType combines the best of the other formats, with the option of glyph outlines being described either as Type 1 or as TrueType.

For these font types, the actual font program is defined in a separate font file, which may be either embedded in a PDF stream object or obtained from an external source.

PDF also has two types of fonts that are specific to PDF, where the glyph data must be defined in the PDF:

Type 3

Originally provided as a way to embed bitmapped fonts, a Type 3 font is really a PDF dictionary where each glyph is defined by a standard content stream. This

allows not only raster-based glyphs, but the use of any/all PDF graphic operators to define the glyph. Although they can be very powerful, they are not used in most PDF-producing systems today.

Type 0

Also known as a composite font or CIDFont, a Type 0 font is created by taking glyph descriptions from one or more other fonts and creating an amalgam or composite. This was originally necessary when working with fonts in Chinese, Japanese, or Korean (CJK) that didn't also have English/Latin characters and a single font with both was desired. Though not used to actually amalgamate glyphs from multiple fonts, it is still the method used for Unicode fonts, especially when dealing with two-byte data.

The Font Dictionary

In [Chapter 1](#), you saw our first PDF, which drew “Hello World” on the page. [Example 4-1](#) shows a few relevant pieces that we’re now going to look at in more detail.

Example 4-1. Parts of “Hello World.pdf”

```
1 0 obj
<<
  /Type /Page
  /Parent 5 0 R
  /MediaBox [ 0 0 612 792 ]
  /Resources 3 0 R
  /Contents 2 0 R
>>
endobj
3 0 obj
<<
  /Font <</F1 4 0 R >>
>>
endobj
4 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /Name /F1
  /BaseFont/Helvetica
>>
endobj
```

In this example, you know that object #1 is the page, and it has just the bare minimum necessary: a `MediaBox` (for the size of the page), the `Contents` (for the drawing instructions), and the `Resources` (needed by the `Contents`). There is a single resource type, `Font`, with a single entry, `F1`, which is the font dictionary at object #4.

A font dictionary contains information that specifies the type of font (i.e., Type 1 or TrueType), its PostScript name, its encoding, and information that can be used to provide a substitute when the font program is not embedded in the PDF file.

ISO 32000-1:2008, 9.6.2.2 describes the Standard 14 (aka Base 14) fonts that every PDF renderer is required to know about and provide either directly or via appropriate substitutes. Additionally, the renderer is supposed to know about their metrics, which will enable us (for now) to not have to determine those values from the font program and incorporate them into the font dictionary. Object #4 in our example is quite small because it is using one of those fonts (Helvetica). The full list of standard fonts is:

- Times-Roman
- Times-Bold
- Times-Italic
- Times-BoldItalic
- Helvetica
- Helvetica-Bold
- Helvetica-Oblique
- Helvetica-BoldOblique
- Courier
- Courier-Bold
- Courier-Oblique
- Courier-BoldOblique
- Symbol
- ZapfDingbats

When creating a font dictionary for one of these fonts, there are only three required keys: `Type` (with a value of `Font`), `Subtype` (with a value of `Type1`), and `BaseName` (with one of the values from the preceding list).

You may have noticed that some of these fonts actually include what you might normally think of as a *style* (such as bold or italic). However, when working with PDF there are no styles, just fonts. So, if you want the Times font shown in italic, you need to use the font `Times-Italic`.

Example 4-2 is a variation of “Hello World” using a few different fonts.

Example 4-2. “Font World” (four different fonts)

Hello World
Hello World
Hello World
Hello World

```
9 0 obj
<<
  /Type/Page
  /Contents 24 0 R
  /MediaBox[0 0 612 792]
  /Parent 5 0 R
  /Resources 13 0 R
>>
endobj
13 0 obj
<<
  /Font <<
    /F1 14 0 R
    /F2 13 0 R
    /F3 12 0 R
    /F4 11 0 R
  >>
>>
endobj
14 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont/Helvetica
>>
endobj
13 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont/Helvetica
>>
endobj
12 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont/Symbol
>>
endobj
11 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont/Courier-Bold
>>
endobj
```

```

<<
  /Type /Font
  /Subtype /Type1
  /BaseFont/Times-Italic
>>
endobj
24 0 obj
<</Length 182>>
stream
BT
  /F1 48 Tf
  1 0 0 1 10 100 Tm
  (Hello World)Tj

  0 50 Td
  /F2 48 Tf
  (Hello World)Tj

  0 50 Td
  /F3 48 Tf
  (Hello World)Tj

  0 50 Td
  /F4 48 Tf
  (Hello World)Tj
ET
endstream
endobj

```

Encodings

The text that we've used so far has been simple English text. PDF, however, considers text associated with the Standard 14 fonts to be in something called *StandardEncoding*, which is a subset of ISO Latin-1 (ISO 8859-1) defined in ISO 32000-1:2008, D.2. If all the text in your PDFs can be expressed in that encoding, then you're good to go. However, most folks need at least the full Latin-1 to represent other standard characters used by other Roman/Latin-based languages such as French, Spanish, or German. To use those, you need to add an explicit encoding to the font dictionary, and continue to use simple text strings as you have done.

Here is an example of using *WinAnsiEncoding* (otherwise known as Windows code page 1252) to write out some text in other languages.

Example 4-3. Drawing text in other languages

Español
Français
English

```
14 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont/Helvetica
  /Encoding/WinAnsiEncoding
>>
endobj
24 0 obj
<</Length 182>>
stream
BT
  /F1 48 Tf
  1 0 0 1 10 100 Tm
  (English)Tj

  0 50 Td
  /F2 48 Tf
  (Français)Tj

  0 50 Td
  /F3 48 Tf
  (Español)Tj
ET
endstream
endobj
```

In order to support arbitrary non-Latin-based languages, it is necessary to use embedded fonts, a topic which is not covered in this edition of this book.



It is possible to support Chinese, Japanese, and Korean (CJK) text in a PDF using nonembedded fonts, but it will require users who want to view the PDF to install extra fonts on their computers. We won't be covering how to do this, as it is not recommended.

Text State

Now that you have a good handle on fonts and glyphs, let's see how you can use them to actually draw text on the page ([Example 4-4](#)).

Example 4-4. Parts of Hello World.pdf

```
1 0 obj
<<
  /Type /Page
  /Parent 5 0 R
  /MediaBox [ 0 0 612 792 ]
  /Resources 3 0 R
  /Contents 2 0 R
>>
endobj
2 0 obj
<< /Length 53 >>
stream
BT
  /F1 24 Tf
  1 0 0 1 260 600 Tm
  (Hello World)Tj
ET
endstream
endobj
```

If you examine the content stream at object #2, you will see five new operators that you haven't seen before. The first one, BT, appears all by itself (i.e., with no operands) on the first line. As you can probably guess, it stands for "Begin Text" and is required to delineate a series of text-related instructions. It is paired with the ET (End Text) operator, which can be seen on the last line of the content stream.

Just as PDF has a graphic state (see "[Graphic State](#)" on page 36 for more on this), it also has a text state that incorporates all the text-drawing-related attributes. The BT/ET pairing declares a new text state and then clears it, much as q and Q do with the graphic state. However, there is no push/pop. In fact, it is not permitted to have nested pairs of BT/ET.

Font and Size

While there are many attributes in a text state, and you'll look at a few here, the three most important are the font to be used, the text size, and where to put the text.

The Tf operator specifies the name of a font resource—that is, an entry in the font subdictionary of the current resource dictionary. The value of that entry is a font dictionary (see "[The Font Dictionary](#)" on page 66). In the previous example, the font is

named F1, and if you refer back to [Figure 4-1](#) you can see that it is present in the Font resources in object #3.

As you learned in “[Transformations](#)” on [page 42](#), PDF uses a general “user unit” concept for defining the size and location of objects. In PDF, the standard glyph size is 1 unit in user space, and the nominal height of tightly spaced lines of text is also 1 unit. Therefore, in order to draw a glyph at a specific size, you need to scale it. The scale factor is specified as the second operand of the Tf operator, thereby setting the text font size parameter in the graphic state. In our example, the font size is 24.

There is a second way to set the scale factor for the glyphs, which is similar to how other graphic objects are scaled—using a transformation matrix. For text-specific transformations, you use the Tm operator, which takes the same parameters as the cm operator we looked at in [Chapter 2](#). In our example, no additional scaling is taking place but the text is being positioned at (260,600). This is the normal way of setting the scaling and position for your first glyph to be drawn. However, it could also be done this way:

```
/F1 1 Tf
24 0 0 24 260 600 Tm
(Hello World)Tj
```

Is one way better than the other? On modern systems and current implementations, the differences should be indistinguishable. If you are trying for the best possible implementation of your content, it will depend on how the font is used. If you are only using the text at a single size (on a given page), using the scale factor as part of the Tf is best, because then the glyphs will be prescaled by the font loader. However, should you be using the same font at multiple sizes on the page, then loading at 1 unit and using Tm to scale each time is probably more optimal—but only if you are switching back and forth a lot. Otherwise, just create a second instance of the font with a second Tf, as shown in [Example 4-5](#).

Example 4-5. Drawing in both big and little text

Big Text

Small Text

```
BT
  /F1 24 Tf
  1 0 0 1 100 150 Tm
  (Big Text)Tj

  /F1 12 Tf
  1 0 0 1 100 100 Tm
  (Small Text)Tj
ET
```

Rendering Mode

In “The Painter’s Model” on page 39, you learned that paths drawn in PDF can be filled, stroked, or both, based on the operator (e.g., f vs. S) that ends the path description. With text, instead of using different operators, there is a single operator (Tr) that sets the text rendering mode. Figure 4-3 lists the possible operand values that can be used with Tr and the impact they have on the text.

	Rendering mode	Description
R	0	Fill text
R	1	Stroke text
R	2	Fill then stroke text
	3	Text with no fill and no stroke (invisible)
R	4	Fill text and add it to the clipping path
R	5	Stroke text and add it to the clipping path
R	6	Fill then stroke text and add it to the clipping path
R	7	Add text to the clipping path

Figure 4-3. The seven text rendering modes

When you combine the rendering modes with the graphic state attributes that you already know about, you can create the text in Example 4-6.

Example 4-6. Stroked and filled text

ABC

```
BT
  /F1 80 Tf
  1 0 0 1 100 100 Tm
  1 0 0 RG
  [2] 0 d
  0.75 g
  2 Tr
  (ABC)Tj
ET
```

Drawing Text

In case you hadn't figured it out by now, the Tj operator is used to draw text (also known as "showing" a string) on a page. It is quite simple, in that the operator causes the PDF renderer to align the first glyph's "glyph origin" with the current pen location and draw the glyph. Then the renderer advances the pen by the width of the glyph to the "next glyph origin" and draws the next glyph, and so on for the entire string.

For the majority of text rendering, this is perfectly acceptable and what most users are used to seeing on the screen. However, for those instances where you wish more precise control over glyph positioning, you will need to use the TJ operator.



Many fonts include information about how to more precisely place certain glyphs in relation to each other, known as *Kerning*. However, this is not supported by the Tj operator when drawing a string. If you want to use that information, you need to obtain it from the font yourself and then use the TJ operator to obtain the more visually appealing result.

The TJ operator, instead of taking a string as an operand, takes an array. The array consists of one or more strings interspersed with numbers, where the numbers serve to adjust the text position (Tm). The numbers are expressed in thousandths of a unit, and the value is subtracted from the current horizontal coordinate.



This means that in the default coordinate system, a positive adjustment has the effect of moving the next glyph painted to the left by the given amount, while a negative adjustment will move the next glyph to the right.

Example 4-7 shows drawing a word using the simple Tj operator, and manually kerning it via TJ.

Example 4-7. Manually kerned text

AWAY
AWAY

```
BT
/F1 48 Tf
1 0 0 1 10 150 Tm
(AWAY)Tj
1 0 0 1 10 100 Tm
[ (A) 120 (W) 120 (A) 95 (Y) ] TJ
ET
```

Positioning Text

In all of the previous examples, the text has been explicitly positioned using the Tm operator. However, that is a fairly heavyweight operation if all you want to do is move the pen in a single direction (e.g., down to the next line, or over to the right). For simpler movements, the Td operator should be used. It takes two parameters, t_x and t_y , representing how to move the pen in the X and Y directions (respectively). If either parameter is 0, the pen isn't moved in that direction. **Example 4-8** illustrates using Td to draw a “4-square.”

Example 4-8. A “4 square” of numbers

1 3
2 4

BT

```
/F1 48 Tf  
1 0 0 1 10 700 Tm  
(1)Tj  
0 -50 Td  
(2)Tj  
50 50 Td  
(3)Tj  
0 -50 Td  
(4)Tj
```

ET



Remember that in PDF, the y coordinate is 0 at the bottom of the page, so to draw text down the page, you start high and subtract.