PDF Succinctly

by Ryan Hodson

Syncfusion | Technology Resource Portal

PDF Succinctly

By
Ryan Hodson

Foreword by Daniel Jebaraj

Syncfusion
Deliver innovation with ease

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

Edited by

This publication was edited by Stephen Jebaraj, senior product manager, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books **8**

Introduction **10**

The PDF standard 10

Chapter 1 Conceptual Overview **11**

Header 11

Body 12

Cross-reference table 13

Trailer 13

Summary 14

Chapter 2 Building a PDF **15**

Header 15

Body 15

The page tree 15

Page(s) 16

Resources 17

Content 18

Catalog 19

Cross-reference table 19

Trailer 19

Compiling the valid PDF 20

Header binary 20

Content stream length 21

Cross-reference table 21

Trailer dictionary 21

Summary 22
Chapter 3 Text Operators 23
The basics 23
Positioning text 24
Text state operators 25
The Tf operator 25
The Tc operator 26
The Tw operator 26
The Tr operator 27
The Ts operator 27
The TL operator 28
Text positioning operators 29
The Td operator 29
The T* operator 30
The Tm operator 30
Text painting operators 31
The Tj operator 32
The ' (single quote) operator 32
The " (double quote) operator 32
The TJ operator 33
Summary 34
Chapter 4 Graphics Operators 35
The basics 35
Graphics state operators 36
The w operator 37
The d operator 37

The J, j, and M operators 37
The cm operator 39
The q and Q operators 40
The RG, rg, K, and k operators 40
Path construction operators 41
The m operator 41
The l (lowercase L) operator 41
The c operator 42
The h operator 42
Graphics painting operators 43
The S and s operators 43
The f operator 43
The B and b operators 44
The * (asterisk) operators 44
Summary 45
Chapter 5 Navigation and Annotations 46
Preparations 46
The document outline 47
The initial destination 50
Hyperlinks 50
Text annotations 51
Summary 52
Chapter 6 Creating PDFs in C# 53
Disclaimer 53
Installation 53
The basics 53

Compiling 54
iTextSharp text objects 55
Chunks 55
Phrases 56
Paragraphs 56
Lists 57
Formatting a document 57
Document dimensions 57
Colors 58
Selecting fonts 58
Custom fonts 59
Formatting text blocks 60
Summary 61
Conclusion 62

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

Introduction

Adobe Systems Incorporated's Portable Document Format (PDF) is the de facto standard for the accurate, reliable, and platform-independent representation of a paged document. It's the only universally accepted file format that allows pixel-perfect layouts. In addition, PDF supports user interaction and collaborative workflows that are not possible with printed documents.

PDF documents have been in widespread use for years, and dozens of free and commercial PDF readers, editors, and libraries are readily available. However, despite this popularity, it's still difficult to find a succinct guide to the native PDF format. Understanding the internal workings of a PDF makes it possible to dynamically generate PDF documents. For example, a web server can extract information from a database, use it to customize an invoice, and serve it to the customer on the fly.

This book introduces the fundamental components of the native PDF language. With the help of a utility program called pdftk from PDF Labs, we'll build a PDF document from scratch, learning how to position elements, select fonts, draw vector graphics, and create interactive tables of contents along the way. The goal is to provide enough information to let you start building your own documents without bogging you down with the many complexities of the PDF file format.

In addition, the last chapter of this book provides an overview of the iTextSharp library (http://itextpdf.com/). iTextSharp is a C# library that provides an object-oriented wrapper for native PDF elements. Having a C# representation of a document makes it much easier to leverage existing .NET components and streamline the creation of dynamic PDF files.

The sample files created in this book can be downloaded here: https://bitbucket.org/syncfusion/pdf-succinctly/.

The PDF standard

The PDF format is an open standard maintained by the International Organization for Standardization. The official specification is defined in ISO 32000-1:2008, but Adobe also provides a free, comprehensive guide called *PDF Reference, Sixth Edition, version 1.7*.

Chapter 1 Conceptual Overview

We'll begin with a conceptual overview of a simple PDF document. This chapter is designed to be a brief orientation before diving in and creating a real document from scratch.

A PDF file can be divided into four parts: a header, body, cross-reference table, and trailer. The header marks the file as a PDF, the body defines the visible document, the cross-reference table lists the location of everything in the file, and the trailer provides instructions for how to start reading the file.

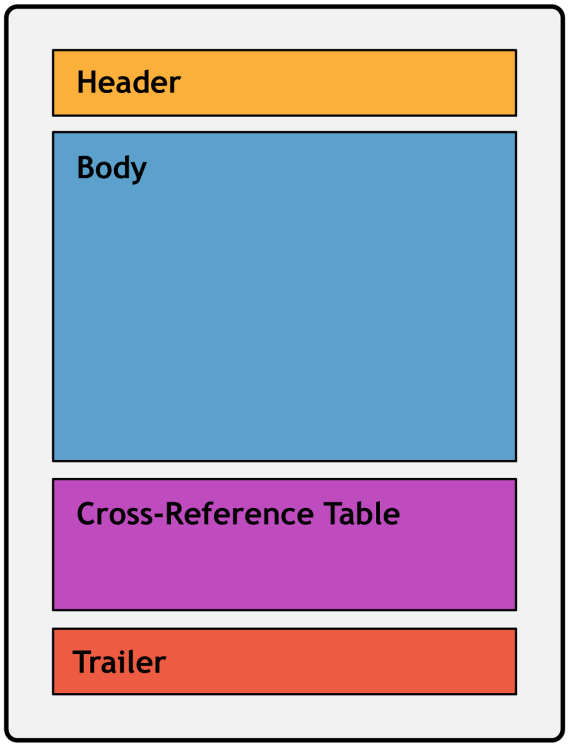

Figure 1: Components of a PDF document

Every PDF file *must* have these four components.

Header

The header is simply a PDF version number and an arbitrary sequence of binary data. The binary data prevents naïve applications from processing the PDF as a text file. This would result in a corrupted file, since a PDF typically consists of both plain text and binary data (e.g., a binary font file can be directly embedded in a PDF).

Body

The body of a PDF contains the entire visible document. The minimum elements required in a valid PDF body are:

- A page tree
- Pages
- Resources
- Content
- The catalog

The **page tree** serves as the root of the document. In the simplest case, it is just a list of the pages in the document. Each **page** is defined as an independent entity with metadata (e.g., page dimensions) and a reference to its resources and content, which are defined separately. Together, the page tree and page objects create the "paper" that composes the document.

Resources are objects that are required to render a page. For example, a single font is typically used across several pages, so storing the font information in an external resource is much more efficient. A **content** object defines the text and graphics that actually show up on the page. Together, content objects and resources define the appearance of an individual page.

Finally, the document's **catalog** tells applications where to start reading the document. Often, this is just a pointer to the root page tree.

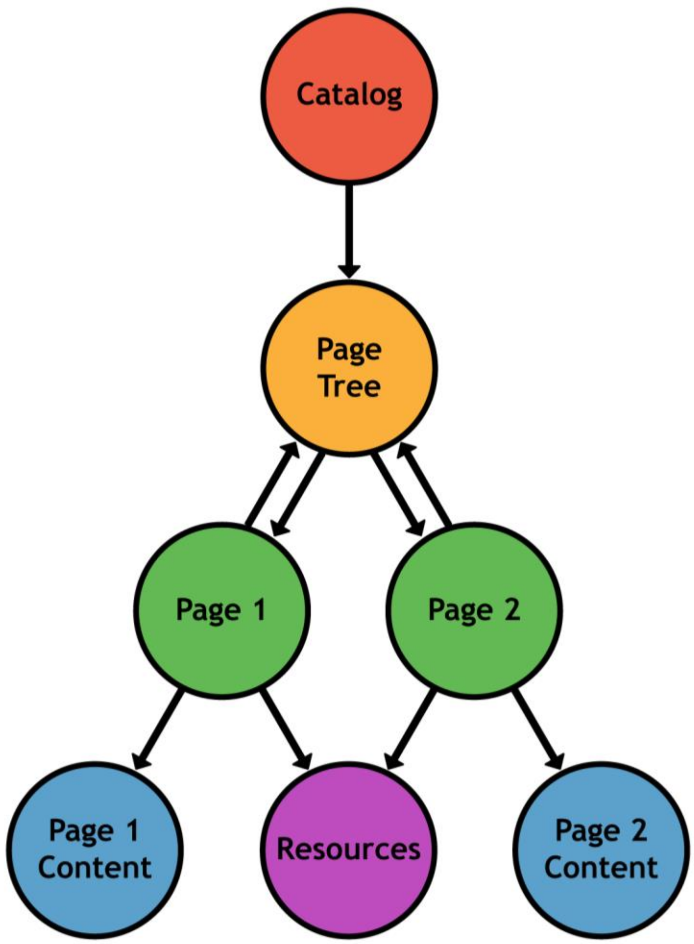

Figure 2: Structure of a document's body

Cross-reference table

After the header and the body comes the cross-reference table. It records the byte location of each object in the body of the file. This enables random-access of the document, so when rendering a page, only the objects required for that page are read from the file. This makes PDFs much faster than their PostScript predecessors, which had to read in the entire file before processing it.

Trailer

Finally, we come to the last component of a PDF document. The trailer tells applications how to start reading the file. At minimum, it contains three things:

- A reference to the catalog which links to the root of the document.
- The location of the cross-reference table.
- The size of the cross-reference table.

Since a trailer is all you need to begin processing a document, PDFs are typically read back-to-front: first, the end of the file is found, and then you read backwards until you arrive at the beginning of the trailer. After that, you should have all the information you need to load any page in the PDF.

Summary

To conclude our overview, a PDF document has a header, a body, a cross-reference table, and a trailer. The trailer serves as the entryway to the entire document, giving you access to any object via the cross-reference table, and pointing you toward the root of the document. The relationship between these elements is shown in the following figure.

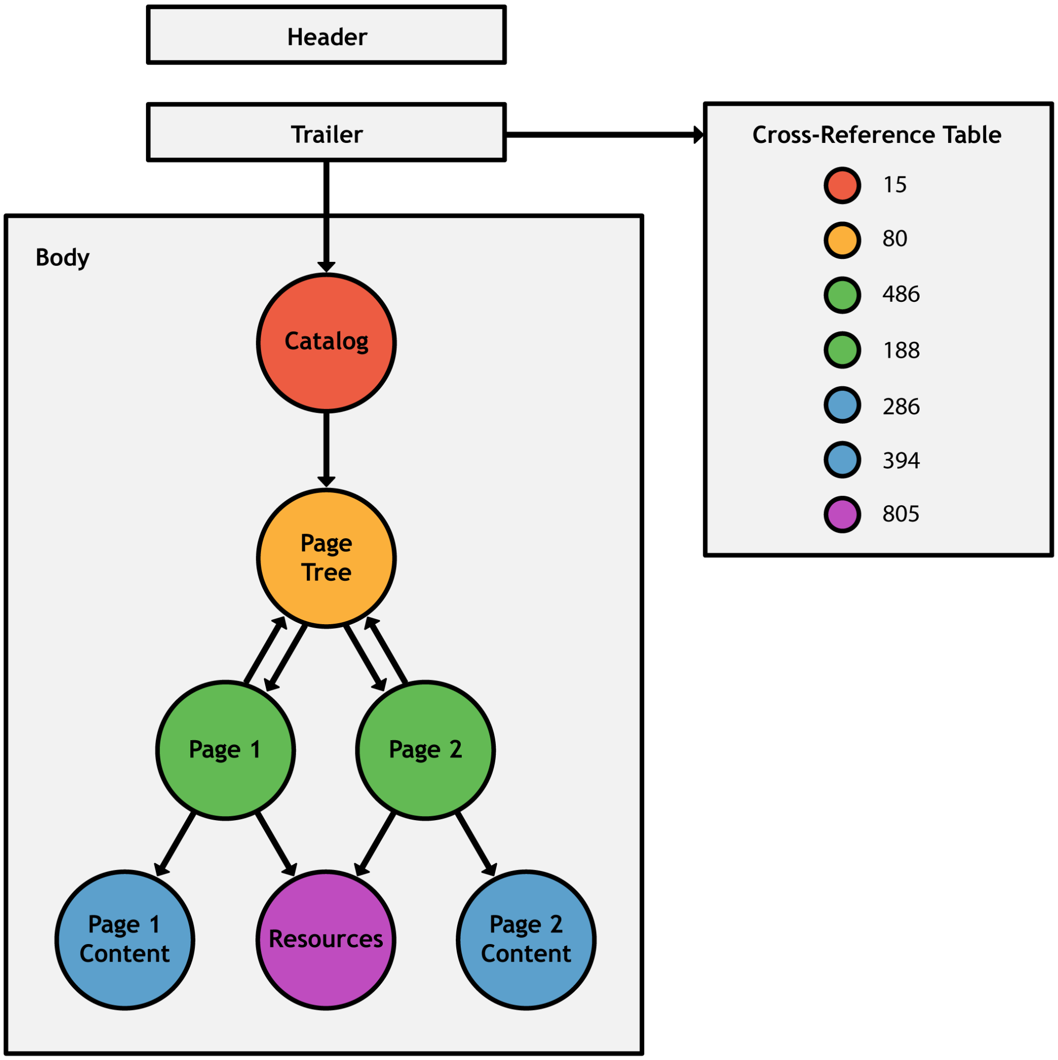

Figure 3: Structure of a PDF document

Chapter 2 Building a PDF

PDFs contain a mix of text and binary, but it's still possible to create them from scratch using nothing but a text editor and a program called pdftk. You create the header, body, and trailer on your own, and then the pdftk utility goes in and fills in the binary blanks for you. It also manages object references and byte calculations, which is not something you would want to do manually.

First, download pdftk from PDF Labs. For Windows users, installation is as simple as unzipping the file and adding the resulting folder to your **`PATH`**. Running **`pdftk --help`** from a command prompt should display the help page if installation was successful.

Next, we'll manually create a PDF file for use with pdftk. Create a plain text file called **hello-src.pdf** (this file is available at https://bitbucket.org/syncfusion/pdf-succinctly) and open it in your favorite text editor.

Header

We'll start by adding a header to **hello-src.pdf**. Remember that the header contains both the PDF version number and a bit of binary data. We'll just add the PDF version and leave the binary data to pdftk. Add the following to **hello-src.pdf**.

```
%PDF-1.0
```

The `%` character begins a PDF comment, so the header is really just a special kind of comment.

Body

The body (and hence the entire visible document) is built up using **objects**. Objects are the basic unit of PDF files, and they roughly correspond to the data structures of popular programming languages. For example, PDF has Boolean, numeric, string, array, and dictionary objects, along with streams and names, which are specific to PDF. We'll take a look at each type as the need arises.

The page tree

The page tree is a dictionary object containing a list of the pages that make up the document. A minimal page tree contains just one page.

```
1 0 obj
<< /Type /Pages
   /Count 1
   /Kids [2 0 R]
>>
endobj
```

Objects are enclosed in the **`obj`** and **`endobj`** tags, and they begin with a unique identification number (**`1 0`**). The first number is the *object number*, and the second is the *generation number*. The latter is only used for incremental updates, so all the generation numbers in our examples will be **`0`**. As we'll see in a moment, PDFs use these identifiers to refer to individual objects from elsewhere in the document.

Dictionaries are set off with angle brackets (**<<** and **>>**), and they contain key/value pairs. White space is used to separate both the keys from the values *and* the items from each other, which can be confusing. It helps to keep pairs on separate lines, as in the previous example.

The **`/Type`**, **`/Pages`**, **`/Count`**, and **`/Kids`** keys are called **names**. They are a special kind of data type similar to the constants of high-level programming languages. PDFs often use names as dictionary keys. Names are case-sensitive.

`2 0 R` is a reference to the object with an identification number of **`2 0`** (it hasn't been created yet). The **`/Kids`** key wraps this reference in square brackets, turning it into an array: **`[2 0 R]`**. PDF arrays can mix and match types, so they are actually more like C#'s **`List<object>`** than native arrays.

Like dictionaries, PDF arrays are also separated by white space. Again, this can be confusing, since the object reference is also separated by white space. For example, adding a second reference to **`/Kids`** would look like: **`[2 0 R 3 0 R]`** (don't actually add this to **hello-src.pdf**, though).

Page(s)

Next, we'll create the second object, which is the only page referenced by **`/Kids`** in the previous section.

```
2 0 obj
<< /Type /Page
   /MediaBox [0 0 612 792]
   /Resources 3 0 R
   /Parent 1 0 R
   /Contents [4 0 R]
>>
endobj
```

The **/Type** entry always specifies the type of the object. Many times, this can be omitted if the object type can be inferred by context. Note that PDF uses a name to identify the object type—not a literal string.

The **/MediaBox** entry defines the dimensions of the page in points. There are 72 points in an inch, so we've just created a standard 8.5 × 11 inch page. **/Resources** points to the object containing necessary resources for the page. **/Parent** points back to the page tree object. Two-way references are quite common in PDF files, since they make it very easy to resolve dependencies in either direction. Finally, **/Contents** points to the object that defines the appearance of the page.

Resources

The third object is a resource defining a font configuration.

```
3 0 obj
<< /Font
   << /F0
      << /Type /Font
         /BaseFont /Times-Roman
         /Subtype /Type1
      >>
   >>
>>
endobj
```

The **/Font** key contains a whole dictionary, opposed to the name/value pairs we've seen previously (e.g., **/Type /Page**). The font we configured is called **/F0**, and the font face we selected is **/Times-Roman**. The **/Subtype** is the format of the font file, and **/Type1** refers to the PostScript type 1 file format.

The specification defines 14 "standard" fonts that all PDF applications should support.

Times-Roman	Helvetica	Courier
Times-Bold	**Helvetica-Bold**	**Courier-Bold**
Times-Italic	*Helvetica-Oblique*	*Courier-Oblique*
Times-BoldItalic	***Helvetica-BoldOblique***	***Courier-BoldOblique***

Symbol (∀∋Φπ⊆) ZapfDingbats (✂☞✔✣☺)

Figure 4: Standard fonts for PDF-compliant applications

Any of these values can be used for the **`/BaseFont`** in a **`/Font`** dictionary. Non-standard fonts *can* be embedded in a PDF document, but it's not easy to do manually. We'll put off custom fonts until we can use iTextSharp's high-level framework.

Content

Finally, we are able to specify the actual content of the page. Page content is represented as a **stream** object. Stream objects consist of a dictionary of metadata and a stream of bytes.

```
4 0 obj
<< >>
stream
BT
    /F0 36 Tf
    50 706 Td
    (Hello, World!) Tj
ET
endstream
endobj
```

The **<< >>** creates an empty dictionary. pdftk will fill this in with any required metadata. The stream itself is contained between the **`stream`** and **`endstream`** keywords. It contains a series of instructions that tell a PDF viewer how to render the page. In this case, it will display "Hello, World!" in 36-point Times Roman font near the top of the page.

The contents of a stream are entirely dependent on context—a stream is just a container for arbitrary data. In this case, we're defining the content of a page using PDF's built-in **operators**. First, we created a text block with **`BT`** and **`ET`**, then we set the font with **`Tf`**, then we positioned the text cursor with **`Td`** and finally drew the text “Hello, World!” with **`Tj`**. This new operator syntax will be discussed in full detail over the next two chapters.

But, it is worth pointing out that PDF streams are in *postfix notation*. Their operands are *before* their operators. For example, **`/F0`** and **`36`** are the parameters for the **`Tf`** command. In C#, you would expect this to look more like **`Tf(/F0, 36)`**. In fact, *everything* in a PDF is in postfix notation. In the statement **`1 0 obj`**, **`obj`** is actually an operator and the object/generation numbers are parameters.

You’ll also notice that PDF streams use short, ambiguous names for commands. It’s a pain to work with manually, but this keeps PDF files as small as possible.

Catalog

The last section of the body is the catalog, which points to the root page tree (**`1 0 R`**).

```
5 0 obj
<< /Type /Catalog
   /Pages 1 0 R
>>
endobj
```

This may seem like an unnecessary reference, but dividing a document into multiple page trees is a common way to optimize PDFs. In such a case, programs need to know where the document starts.

Cross-reference table

The cross-reference table provides the location of each object in the body of the file. Locations are recorded as byte-offsets from the beginning of the file. This is another job for pdftk—all we have to do is add the **`xref`** keyword.

```
xref
```

We’ll take a closer look at the cross-reference table after we generate the final PDF.

Trailer

The last part of the file is the trailer. It’s comprised of the **`trailer`** keyword, followed by a dictionary that contains a reference to the catalog, then a pointer to the cross-reference table, and finally an end-of-file marker. Let’s add all of this to **hello-src.pdf**.

```
trailer
<< /Root 5 0 R
>>
startxref
%%EOF
```

The **`/Root`** points to the *catalog*, not the root page tree. This is important because the catalog can also contain important information about the document structure. The **`startxref`** keyword points to the location (in bytes) of the beginning of the cross-reference table. Again, we'll leave this for pdftk. Between these two bits of information, a program can figure out the location of anything it needs.

The **`%%EOF`** comment marks the end of the PDF file. Incremental updates make use of multiple trailers, so it's possible to have multiple **`%%EOF`** lines in a single document. This helps programs determine what new content was added in each update.

Compiling the valid PDF

Our **hello-src.pdf** file now contains a complete document, minus a few binary sequences and byte locations. All we have to do is run pdftk to fill in these holes.

```
pdftk hello-src.pdf output hello.pdf
```

You can open the generated **hello.pdf** file in any PDF viewer and see "Hello, World!" in 36-point Times Roman font in the upper left corner.

*Figure 5: Screenshot of **hello.pdf** (not drawn to scale)*

Let's take a look at what `pdtfk` had to add to our source file…

Header binary

If you open up **hello.pdf**, you'll find another line in the header.

```
%PDF-1.0
%âãÏÓ
```

Again, this prevents programs from processing the file as text. We didn’t have much binary in our “Hello, World!” example, but many PDFs embed complete font files as binary data. Performing a naïve find-and-replace on such a file has the potential to corrupt the font data.

Content stream length

Next, scroll down to object **`4 0`**.

```
4 0 obj
<< /Length 62
>>
stream
...
```

pdftk added a **`/Length`** key that contains the length of the stream, in bytes. This is a useful bit of information for programs reading the file.

Cross-reference table

After that, we have the complete **`xref`** table.

```
endobj xref
0 6
0000000000 65535 f
0000000015 00000 n
0000000074 00000 n
0000000182 00000 n
0000000280 00000 n
0000000395 00000 n
```

It begins by specifying the length of the **`xref`** (6 lines), then it lists the byte offset of each object in the file on a separate line. Once a program has located the **`xref`**, it can find any object using only this information.

Trailer dictionary

Also note that pdftk added the size of the **`xref`** to the trailer dictionary.

```
<<
/Root 5 0 R
/Size 6
>>
```

Finally, pdftk filled in the **`startxref`** keyword, enabling programs to quickly find the cross-reference table.

```
startxref
445
```

Summary

And that's all there is to a PDF document. It's simply a collection of objects that define the pages in a document, along with their contents, and some pointers and byte offsets to make it easier to find objects.

Of course, real PDF documents contain much more text and graphics than our **hello.pdf**, but the process is the same. We got a small taste of how PDFs represent content, but skimmed over many important details. The next chapter covers the text-related operators of content streams.

Chapter 3 Text Operators

As we saw in the previous chapter, PDFs use streams to define the appearance of a page. Content streams typically consist of a sequence of commands that tell the PDF viewer or editor what to draw on the page. For example, the command **`(Hello, World!) Tj`** writes the string "Hello, World!" to the page. In this chapter, we'll discover exactly how this command works, and explore several other useful operators for formatting text.

The basics

The general procedure for adding text to a page is as follows:

1. Define the font state (**`Tf`**).
2. Position the text cursor (**`Td`**).
3. "Paint" the text onto the page (**`Tj`**).

Let's start by examining a simplified version of our existing stream.

```
BT
    /F0 36 Tf
    (Hello, World!) Tj
ET
```

First, we create a text block with the **`BT`** operator. This is required before we can use any other text-related operators. The corresponding **`ET`** operator ends the current text block. Text blocks are isolated environments, so the selected font and position won't be applied to subsequent text blocks.

The next line sets the font face to **`/F0`**, which is the Times Roman font we defined in the **`3 0 obj`**, and sets the size to 36 points. Again, PDF operators use postfix notation—the command (**`Tf`**) comes last, and the arguments come first (**`/F0`** and **`36`**).

Now that the font is selected, we can draw some text onto the page with **`Tj`**. This operator takes one parameter: the string to display (**`(Hello, World!)`**). String literals in a PDF must be enclosed in parentheses. Nested parentheses do not need to be escaped, but single ones need to be preceded by a backslash. So, the following two lines are both valid string literals.

```
(Nested (parentheses) don't need a backslash.)
(But a single \(parenthesis needs one.)
```

Of course, a backslash can also be used to escape itself (`\\`).

Positioning text

If you use pdftk to generate a PDF with the content stream at the beginning of this chapter (without the **`Td`** operator), you'll find that "Hello, World!" shows up at the bottom-left corner of the page.

Since we didn't set a position for the text, it was drawn at the origin, which is the bottom-left corner of the page. PDFs use a classic Cartesian coordinate system with **`x`** increasing from left to right and **`y`** increasing from bottom to top.

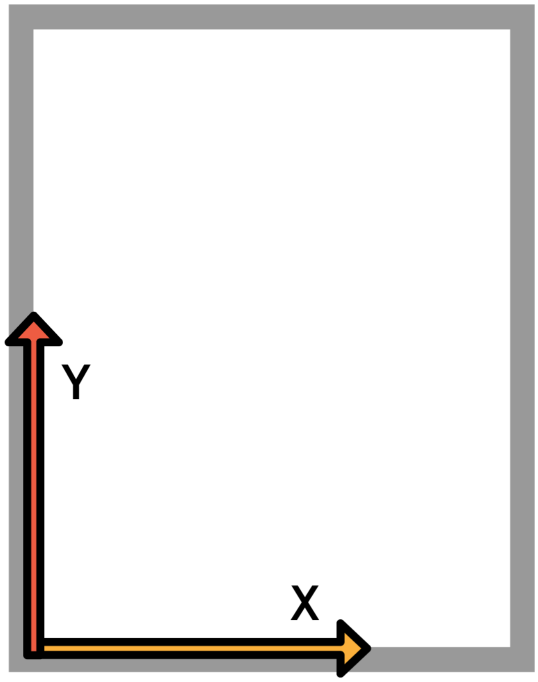

Figure 6: The PDF coordinate system

We have to manually determine where our text should go, then pass those coordinates to the **`Td`** operator *before* drawing it with **`Tj`**. For example, consider the following stream.

```
BT
    /F0 36 Tf
    50 706 Td
    (Hello, World!) Tj
ET
```

This positions our text at the top-left of the page with a 50-point margin. Note that the text block's origin is *its* bottom-left corner, so the height of the font had to be subtracted from the y-position (**`792-50-36=706`**). The PDF file format only defines a method for *representing* a document. It does *not* include complex layout capabilities like line wrapping or line breaks—these things must be determined manually (or with the help of a third-party layout engine).

To summarize, pages of text are created by selecting the text state, positioning the text cursor, and then painting the text to the page. In the digital era, this process is about as close as you'll come to hand-composing a page on a traditional printing press.

Next, we'll take a closer look at the plethora of options for formatting text.

Text state operators

The appearance of all text drawn with **`Tj`** is determined by the text state operators. Each of these operators defines a particular attribute that all subsequent calls to **`Tj`** will reflect. The following list shows the most common text state operators. Each operator's arguments are shown in angled brackets.

- **`<font> <size> Tf`**: Set font face and size.
- **`<spacing> Tc`**: Set character spacing.
- **`<spacing> Tw`**: Set word spacing.
- **`<mode> Tr`**: Set rendering mode.
- **`<rise> Ts`**: Set text rise.
- **`<leading> TL`**: Set leading (line spacing).

The Tf operator

We've already seen the **`Tf`** operator in action, but let's see what happens when we call it more than once:

```
BT
    /F0 36 Tf
    50 706 Td
    (Hello, World!) Tj
    /F0 12 Tf
    (Hello, Again!) Tj
ET
```

This changes the font size to 12 points, but it's still on the same line as the 36-point text:

Figure 7: Changing the font size with `Tf`

The **`Tj`** operator leaves the cursor at the end of whatever text it added—new lines must be explicitly defined with one of the positioning or painting operators. But before we start with positioning operators, let's take a look at the rest of the text state operators.

The Tc operator

The **`Tc`** operator controls the amount of space between characters. The following stream will put 20 points of space between each character of "Hello, World!"

```
BT
    /F0 36 Tf
    50 706 Td
    20 Tc
    (Hello, World!) Tj
ET
```

This is similar to the tracking functionality found in document-preparation software. It is also possible to specify a negative value to push characters closer together.

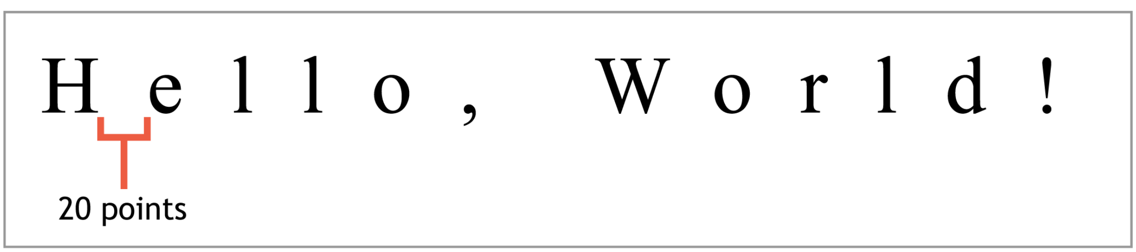

Figure 8: Setting the character spacing to 20 points with `Tc`

The Tw operator

Related to the **`Tc`** operator is **`Tw`**. This operator controls the amount of space between words. It behaves exactly like **`Tc`**, but it only affects the space character. For example, the following command will place words an extra 10 points apart (on top of the character spacing set by **Tc**).

```
10 Tw
```

Together, the **Tw** and **Tc** commands can create justified lines by subtly altering the space in and around words. Again, PDFs only provide a way to represent this—you must use a dedicated layout engine to figure out how words and characters should be spaced (and hyphenated) to fit the allotted dimensions.

That is to say, there is no "justify" command in the PDF file format, nor are there "align left" or "align right" commands. Fortunately, the iTextSharp library discussed in the final chapter of this book *does* include this high-level functionality.

The Tr operator

The **Tr** operator defines the "rendering mode" of future calls to painting operators. The rendering mode determines if glyphs are filled, stroked, or both. These modes are specified as an integer between 0 and 2.

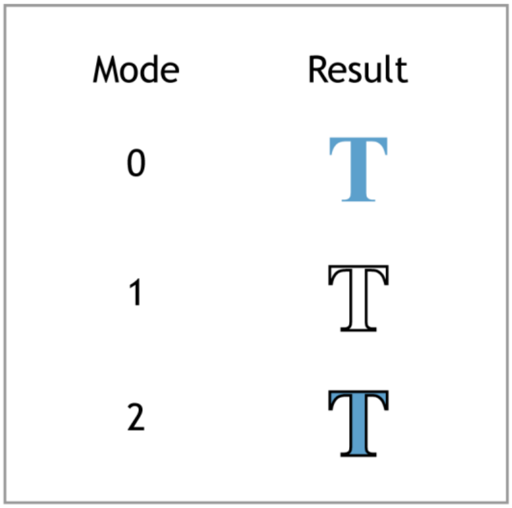

Figure 9: Text rendering modes

For example, the command **2 Tr** tells a PDF reader to outline any new text in the current stroke color and fill it with the current fill color. Colors are determined by the graphics operators, which are described in the next chapter.

The Ts operator

The **Ts** command offsets the vertical position of the text to create superscripts or subscripts. For example, the following stream draws "x^2".

```
BT
    /F0 12 Tf
    50 706 Td
    (x) Tj
    7 Ts
    /F0 8 Tf
    (2) Tj
ET
```

Text rise is always measured relative to the baseline, so it isn't considered a text positioning operator in its own right.

The TL operator

The **TL** operator sets the leading to use between lines. Leading is defined as the distance from baseline to baseline of two lines of text. This takes into account the ascenders and descenders of the font face. So, instead of defining the amount of space you want between lines, you need to add it to the height of the current font to determine the total value for **TL**.

Figure 10: Measuring leading from baseline to baseline

For example, setting the leading to 16 points after selecting a 12-point font will put 4 points of white space between each line. However, font designers can define the height of a font independently of its glyphs, so the actual space between each line might be slightly more or less than what you pass to **TL**.

```
BT
    /F0 36 Tf
    50 706 Td
    (Hello, World!) Tj
    /F0 12 Tf
    16 TL
    T*
    (Hello, Again!) Tj
ET
```

T* moves to the next line so we can see the effect of our leading. This positioning operator is described in the next section.

Text positioning operators

Positioning operators determine where new text will be inserted. Remember, PDFs are a rather low-level method for representing documents. It's not possible to define the width of a paragraph and have the PDF document fill it in until it runs out of text. As we saw earlier, PDFs can't even line-wrap on their own. These kinds of advanced layout features must be determined with a third-party layout engine, and then represented by manually moving the text position and painting text as necessary.

The most important positioning operators are:

- **<x> <y> Td**: Move to the start of the next line, offset by (**<x>**, **<y>**).
- **T***: Move to the start of the next line, offset by the current leading.
- **<a> <b> <c> <d> <e> <f> Tm**: Manually define the text matrix.

The Td operator

Td is the basic positioning operator. It moves the text position by a horizontal and vertical offset measured from the beginning of the current line. We've been using **Td** to put the cursor at the top of the page (**50 706 Td**), but it can also be used to jump down to the next line.

```
BT
    /F0 36 Tf
    50 706 Td
    (Hello, World!) Tj
    /F0 12 Tf
    0 -16 Td
    (Hello, Again!) Tj
ET
```

The previous stream draws the text "Hello, World!" then moves down 16 points with **Td** and draws "Hello, Again!" Since the height of the second line is 12 points, the result is a 4-point gap between the lines. This is the manual way to define the leading of each line.

Note that positive y values move *up*, so a negative value must be used to move to the next line.

The T* operator

`T*` is a shortcut operator that moves to the next line using the current leading. It is the equivalent of `0 -<leading> Td`.

The Tm operator

Internally, PDFs use a *transformation matrix* to represent the location and scale of all text drawn onto the page. The following diagram shows the structure of the matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Figure 11: The text transformation matrix

The `e` and `f` values determine the horizontal and vertical position of the text, and the `a` and `d` values determine its horizontal and vertical scale, respectively. Altering more than just those entries creates more complex transformations like skews and rotations.

This matrix can be defined by passing each value as an argument to the `Tm` operator.

```
<a> <b> <c> <d> <e> <f> Tm
```

Most of the other text positioning and text state commands are simply predefined operations on the transformation matrix. For example, setting `Td` adds to the existing `e` and `f` values. The following stream shows how you can manually set the transformation matrix instead of using `Td` or `T*` to create a new line.

```
BT
    /F0 36 Tf
    1 0 0 1 50 706 Tm
    (Hello, World!) Tj
    1 0 0 1 50 670 Tm
    (Hello, World!) Tj
ET
```

Likewise, we can change the matrix's **a** and **d** values to change the font size without using **Tf**. The next stream scales down the initial font size by 33%, resulting in a 12-point font for the second line.

```
BT
    /F0 36 Tf
    1 0 0 1 50 706 Tm
    (Hello, World!) Tj
    .33 0 0 .33 50 694 Tm
    (Hello, World!) Tj
ET
```

Of course, the real utility of **Tm** is to define more than just simple translation and scale operations. It can be used to combine several complex transformations into a single, concise representation. For example, the following matrix rotates the text by 45 degrees and moves it to the middle of the page.

```
BT
    /F0 36 Tf
    .7071 -.7071 .7071 .7071 230 450 Tm
    (Hello, World!) Tj
ET
```

More information about transformation matrices is available from any computer graphics textbook.

Text painting operators

Painting operators display text on the page, potentially modifying the current text state or position in the process. The **Tj** operator that we've been using is the core operator for displaying text. The other painting operators are merely convenient shortcuts for common typesetting tasks.

The PDF specification defines four text painting operators:

- **<text> Tj**: Display the text at the current text position.
- **<text>** ': Move to the next line and display the text.
- **<word-spacing> <character-spacing> <text>** ": Move to the next line, set the word and character spacing, and display the text.
- **<array> TJ**: Display an array of strings while manually adjusting intra-letter spacing.

The Tj operator

The **`Tj`** operator inserts text at the current position and leaves the cursor wherever it ended. Consider the following stream.

```
BT
    /F0 36 Tf
    50 706 Td
    (Hello, World!) Tj
    (Hello, Again!) Tj
ET
```

Both **`Tj`** commands will paint the text on the same line, without a space in between them.

The ' (single quote) operator

The `'` (single quote) operator moves to the next line *then* displays the text. This is the exact same functionality as **`T*`** followed by **`Tj`**:

```
BT
    50 706 Td
    /F0 36 Tf
    36 TL
    (Hello, World!) Tj T*
    (I'm On Another Line!) Tj
    (So Am I!) '
ET
```

Like **`T*`**, the `'` operator uses the current leading to determine the position of the next line.

The " (double quote) operator

The `"` (double quote) operator is similar to the single quote operator, except it lets you set the character spacing and word spacing at the same time. Thus, it takes three arguments instead of one.

```
2 1 (Hello!) "
```

This is the exact same as the following.