PDF

Succinctly

by Ryan Hodson

Syncfusion | Technology Resource Portal

PDF Succinctly

By
Ryan Hodson

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Copyright © 2012 by Syncfusion Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

All rights reserved.

Important licensing information. Please read.
This book is available for free download from www.syncfusion.com on completion of a registration form.
If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.
This book is licensed for reading only if obtained from www.syncfusion.com.
This book is licensed strictly for personal, educational use.
Redistribution in any form is prohibited.
The authors and copyright holders provide absolutely no warranty for any information provided.
The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.
Please do not use this book if the listed terms are unacceptable.
Use shall constitute acceptance of the terms listed.

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.

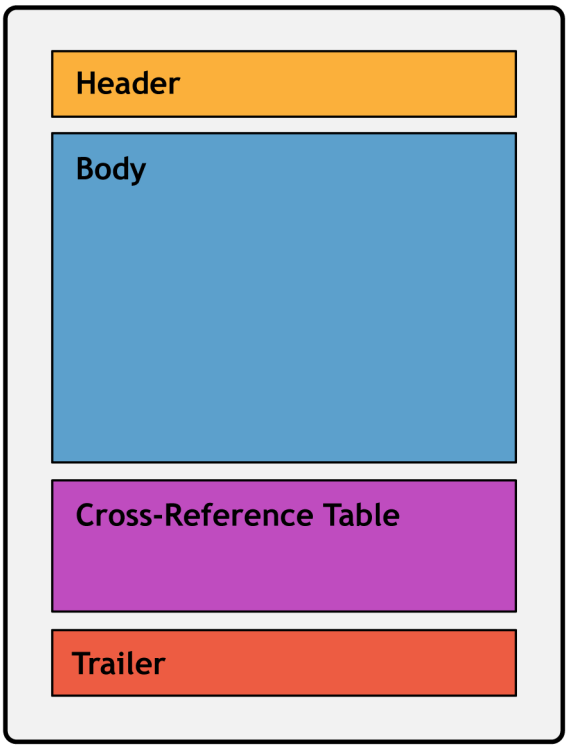

Figure 1: Components of a PDF document

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.

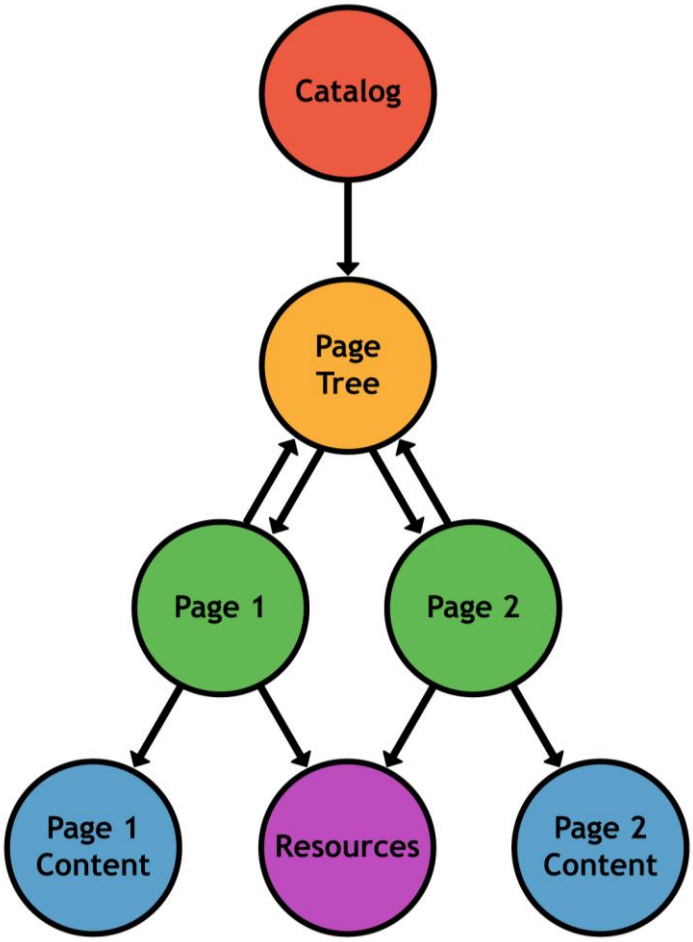

Figure 2: Structure of a document's body

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.

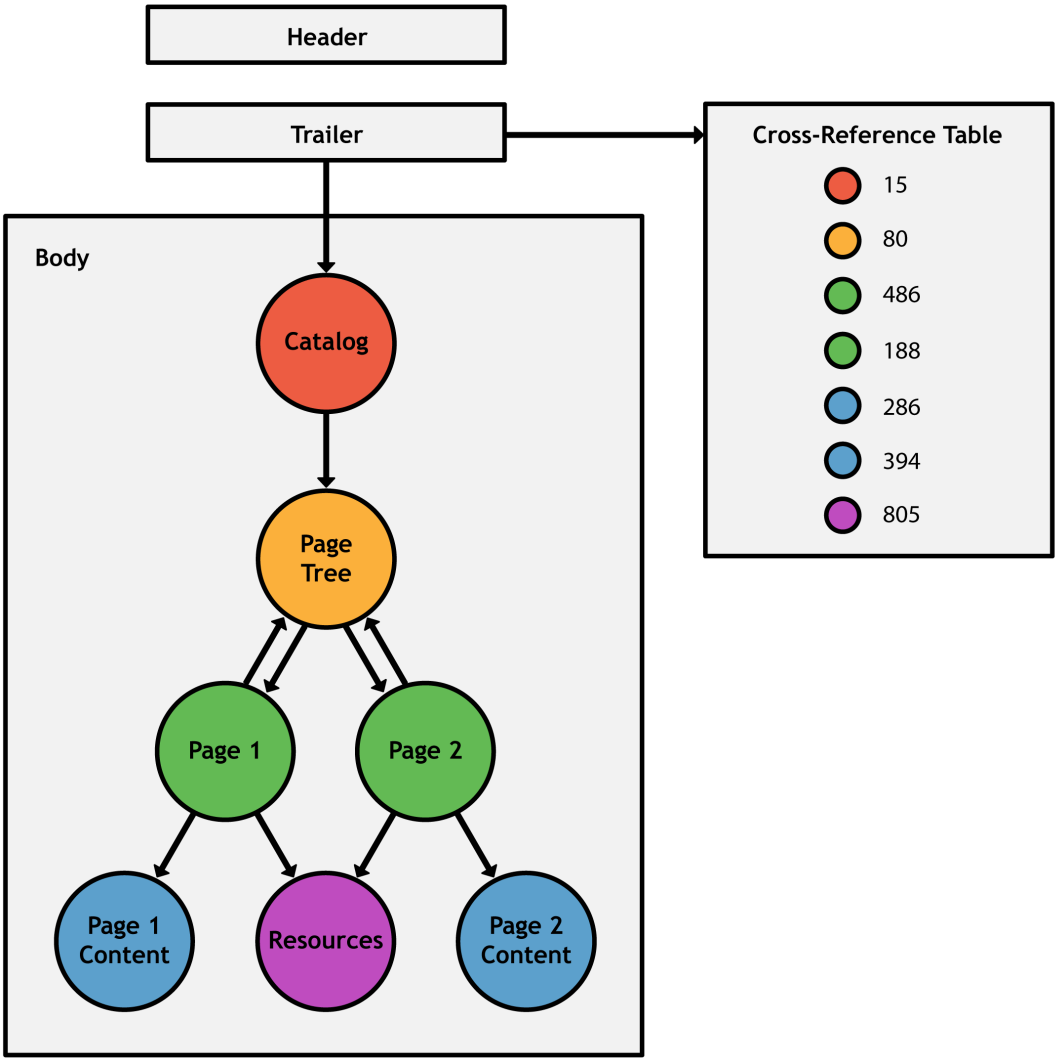

Figure 3: Structure of a PDF document

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.

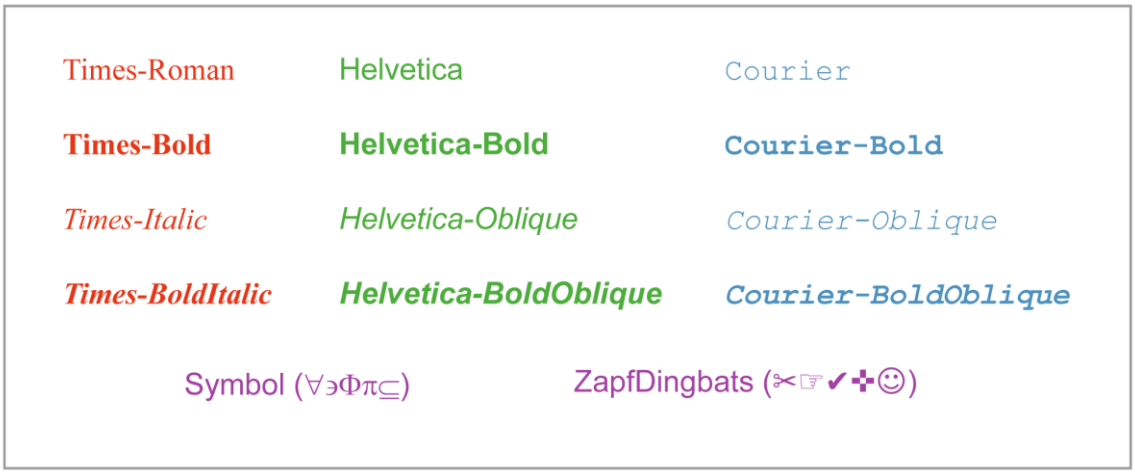

Figure 4: Standard fonts for PDF-compliant applications

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.

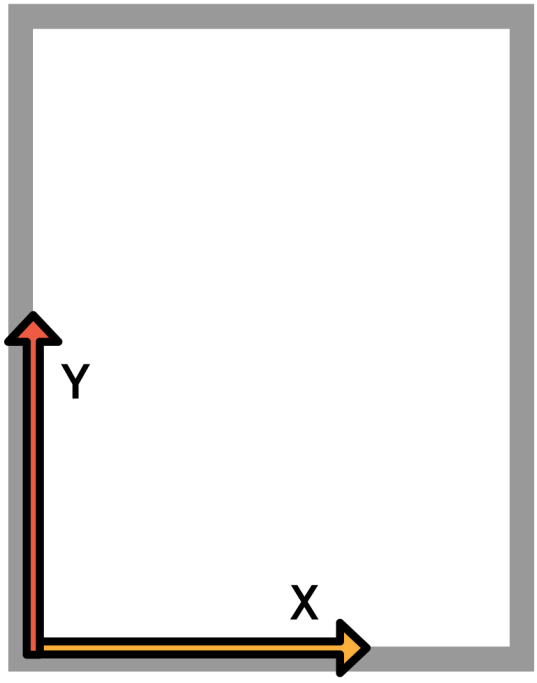

Figure 6: The PDF coordinate system

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.

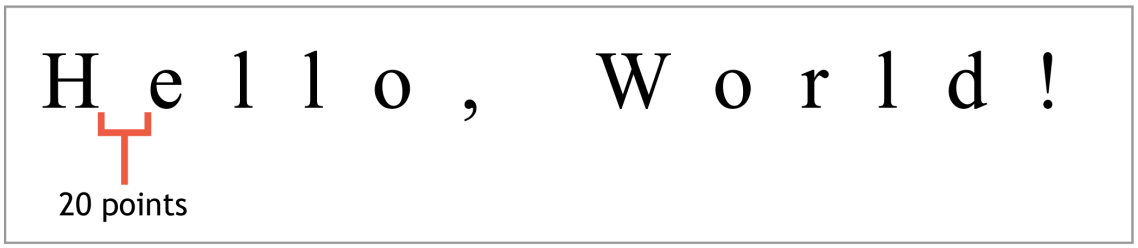

Figure 8: Setting the character spacing to 20 points with `Tc`

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.

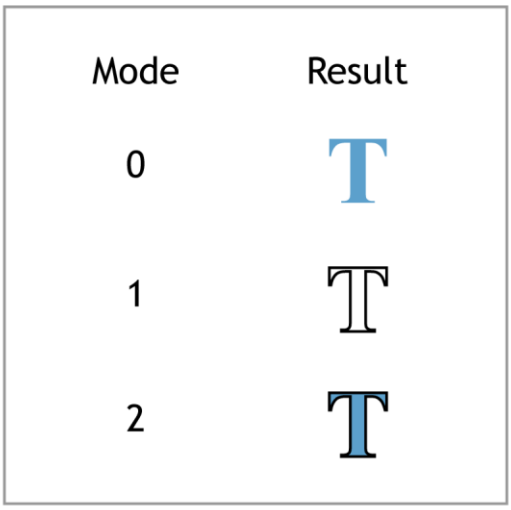

Figure 9: Text rendering modes

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.

```
2 Tw
1 Tc
(Hello!) '
```

Remember that **Tw** and **Tc** are often used for justifying paragraphs. Since each line usually needs distinct word and character spacing, the **"** operator is a very convenient command for rendering justified paragraphs.

```
BT
    /F0 36 Tf
    50 706 Td
    36 TL
    (The double quote oper-) Tj
    1 1 (ator is very useful for) "
    1 1.7 (creating justifed text) "
ET
```

This stream uses character and word spacing to justify three lines of text:

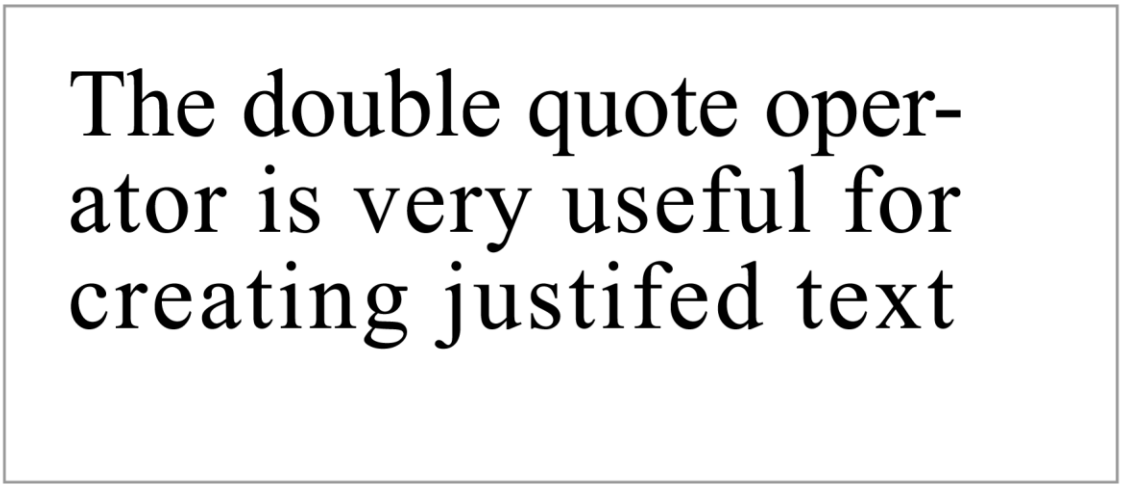

Figure 12: Adjusting character and word spacing to create justified lines

The TJ operator

The **TJ** operator provides even more flexibility by letting you independently specify the space between letters. Instead of a string, **TJ** accepts an array of strings and numbers. When it encounters a string, **TJ** displays it just as **Tj** does. But when it encounters a number, it subtracts that value from the current horizontal text position.

This can be used to adjust the space between individual letters in an entire line using a single command. In traditional typography, this is called kerning.

```
BT
    /F0 36 Tf
    50 706 Td
    36 TL
    (Away With You!) Tj T*
    [(A) 100 (way W) 60 (ith Y) 150 (ou!)] TJ
ET
```

This stream uses **TJ** to kern the “Aw”, “Wi”, and “Yo” pairs. The idea behind kerning is to eliminate conspicuous white space in order to create an even gray on the page. The result is shown in the following figure.

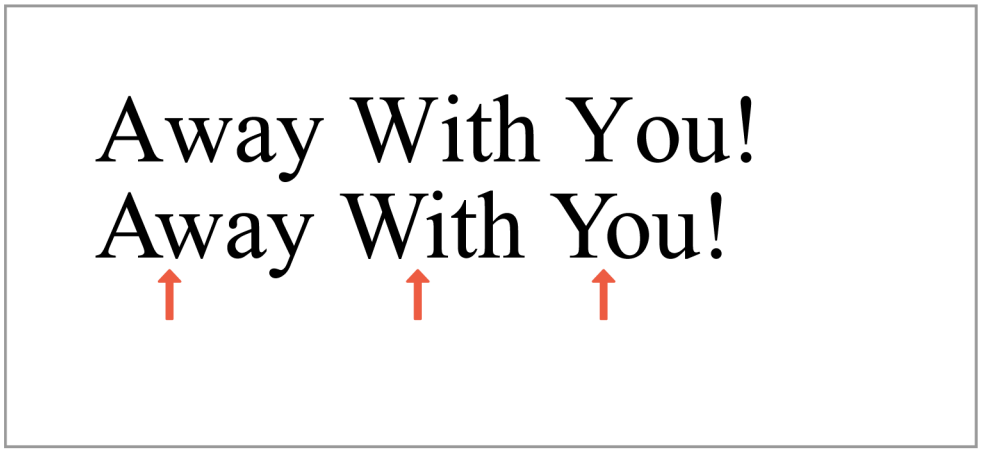

*Figure 13: Kerning letter pairs with **TJ***

Summary

This chapter presented the most common text operators used by PDF documents. These operators make it possible to represent multi-page, text-based documents with a minimum amount of markup. If you’re coming from a typographic background, you’ll appreciate many of the convenience operators like **TJ** for kerning and **"** for justifying lines.

You’ll also notice that PDFs do not separate content from presentation. This is a fundamental difference between creating a PDF versus an HTML document. PDFs represent content and formatting at the same time using *procedural* operators, while other popular languages like HTML and CSS apply style rules to semantic elements. This allows PDFs to represent pixel-perfect layouts, but it also makes it much harder to extract text from a document.

Chapter 4 Graphics Operators

In addition to text, PDFs are also a reliable format for the accurate reproduction of vector graphics. In fact, the Adobe Illustrator file format (.ai) is really just an extended form of the PDF file format. This chapter introduces the core components of the PDF graphics model.

The basics

Like text operators, graphics operators only provide the low-level functionality for *representing* graphics in a page's content stream. PDFs do not have "circles" and "rectangles"—they have only paths.

Drawing paths is similar to drawing text, except instead of positioning the text cursor, you must construct the entire path before painting it. The general process for creating vector graphics is:

1. Define the graphics state (fill/stroke colors, opacity, etc.).
2. Construct a path.
3. Paint the path onto the page.

For example, the following stream draws a vertical line down the middle of the page:

```
10 w
306 396 m
306 594 l
S
```

First, this sets the stroke width to 10 points with the **`w`** operator. Then, we begin constructing a path by moving the graphics cursor to the point (306, 396) with **`m`**. This is similar to the **`Td`** command for setting the text position. Next, we draw a line from the current position to the point (306, 594) using the **`l`** (lowercase L) operator. At this point, the path isn't visible—it's still in the construction phase. The path needs to be painted using the **`S`** operator. All paths must be explicitly stroked or filled in this manner.

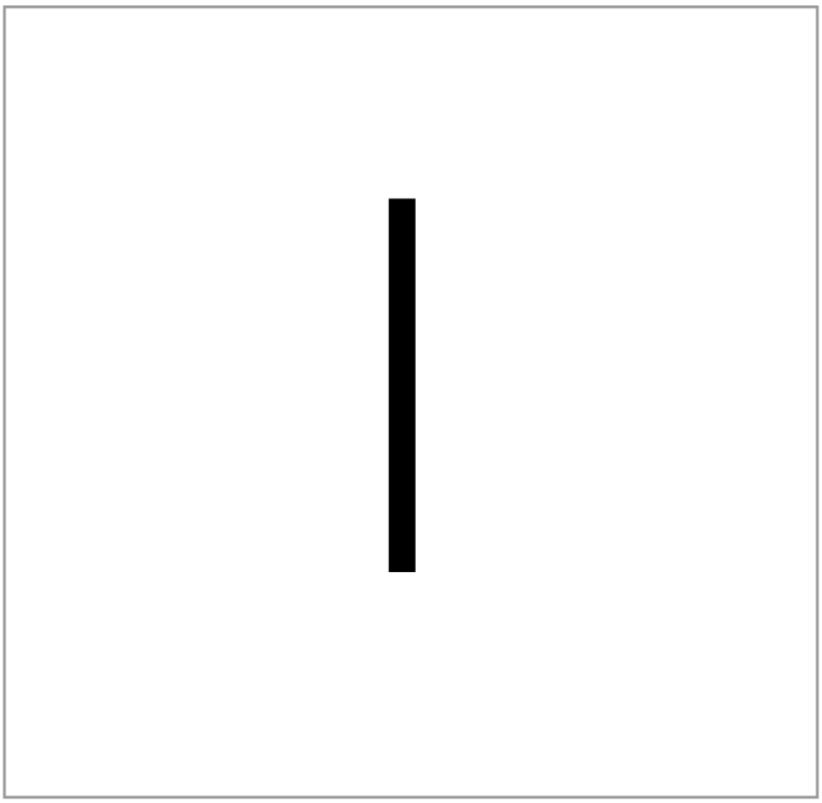

Figure 14: Screenshot of the previous stream (not drawn to scale)

Also notice that graphics don't need to be wrapped in **`BT`** and **`ET`** commands like text operators.

Next, we'll take a closer look at common operators for each phase of producing graphics.

Graphics state operators

Graphics state operators are similar to text state operators in that they both affect the appearance of all painting operations. For example, setting the stroke width will determine the stroke width of all subsequent paths, just like **`Tf`** sets the font face and size of all subsequent text. This section covers the following graphics state operators:

- **`<width> w`**: Set the stroke width.
- **`<pattern> <phase> d`**: Set the stroke dash pattern.
- **`<cap> J`**: Set the line cap style (endpoints).
- **`<cap> j`**: Set the line join style (corners).
- **`<limit> M`**: Set the miter limit of corners.
- **`<a> <b> <c> <d> <e> <f> cm`**: Set the graphics transformation matrix.
- **`q`** and **`Q`**: Create an isolated graphics state block.

The w operator

The **w** operator defines the stroke width of future paths, measured in points. Remember though, PDFs don't draw the stroke of a path as it is being constructed—that requires a painting operator.

The d operator

The **d** operator defines the dash pattern of strokes. It takes two parameters: an array and an offset. The array contains the dash pattern as a series of lengths. For example, the following stream creates a line with 20-point dashes with 10 points of space in between.

```
10 w
[20 10] 0 d
306 396 m
306 594 l
S
```

A few dash examples are included in the following figure. The last one shows you how to reset the dash state to a solid line.

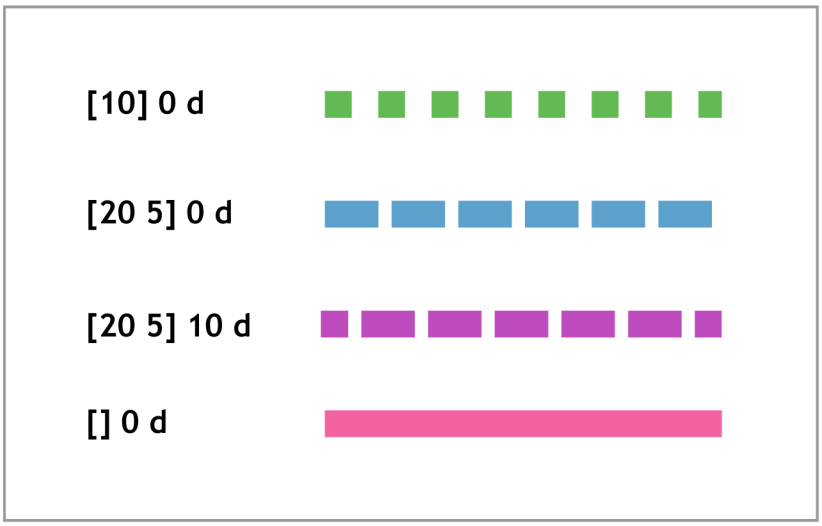

Figure 15: Dashed lines demonstrating the behavior of ***d***

The J, j, and M operators

All three of these operators relate to the styling of the ends of path segments. The **J** operator defines the cap style, and **j** determines the join style. Both of them take an

integer representing the style to use. The available options are presented in the following figure.

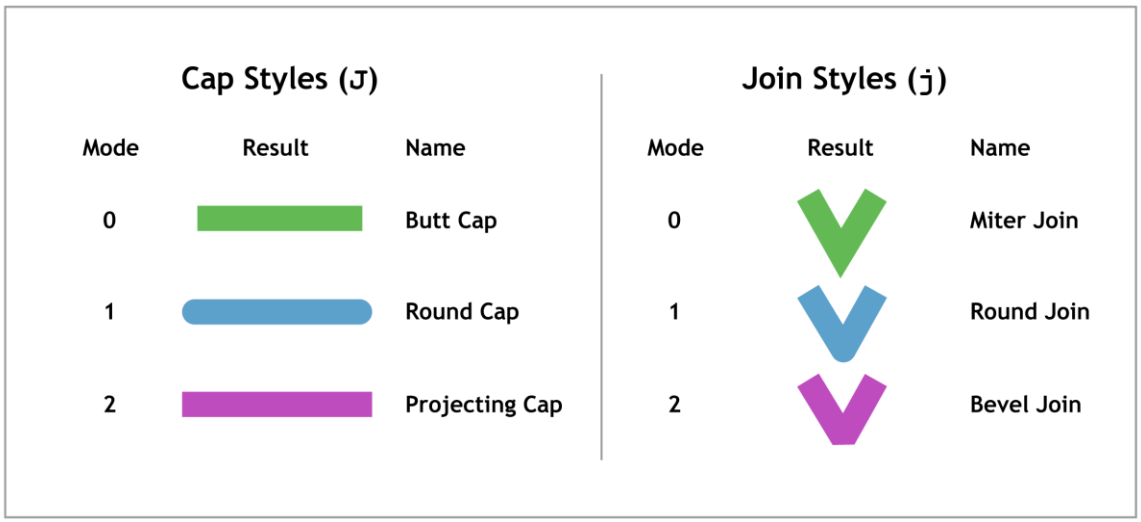

Figure 16: Available modes for line caps and joins

The **M** operator sets the miter limit, which determines when mitered corners turn into bevels. This prevents lines with thick strokes from having long, sharp corners. Consider the following stream.

```
10 w
5 M
306 396 m
306 594 l
336 500 l
S
```

The **5 M** command turns what would be a mitered corner into a beveled one.

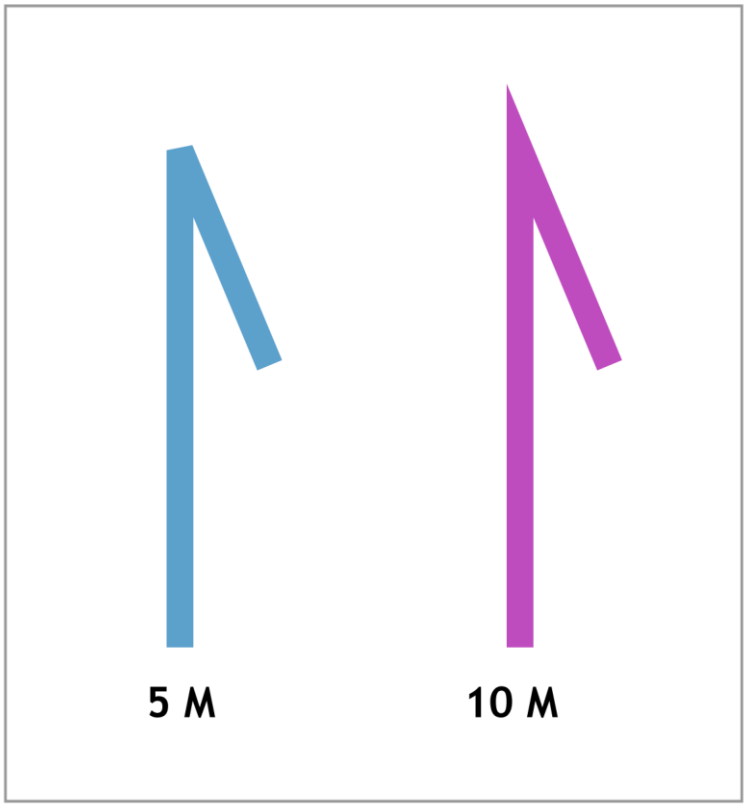

Figure 17: Forcing a beveled corner with ***`5 M`***

Increasing the miter limit from **`5 M`** to **`10 M`** will allow the PDF to display a sharp corner.

The cm operator

Much like the **`Tm`** operator, **`cm`** sets the transformation matrix for everything drawn onto a page. Like the **`Tm`** matrix, it can rotate, scale, and skew graphics. But its most common usage is to change the origin of the page:

```
1 0 0 1 306 396 cm
10 w
5 M
0 0 m
0 198 l
30 104 l
S
```

This stream starts by moving the origin to the center of the page (instead of the lower-left corner). Then it draws the exact same graphic as the previous section, but using coordinates that are relative to the new origin.

The q and Q operators

Complex graphics are often built up from smaller graphics that all have their own state. It's possible to separate elements from each other by placing operators in a **q**/**Q** block:

```
q
    1 0 0 1 306 396 cm
    10 w
    0 0 m
    0 198 l
    30 104 l
    S
Q
BT
    /F0 36 Tf
    (I'm in the corner!) Tj
ET
```

Everything between the **q** and **Q** operators happens in an isolated environment. As soon as **Q** is called, the **cm** operator is forgotten, and the origin returns to the bottom-left corner.

The RG, rg, K, and k operators

While colors aren't technically considered graphics state operators, they do determine the color of all future drawing operators, so this is a logical place to introduce them.

PDFs can represent several color spaces, the most common of which are RGB and CMYK. In addition, stroke color and fill color can be selected independently. This gives us four operators for selecting colors:

- **RG**: Change the stroke color space to RGB and set the stroke color.
- **rg**: Change the fill color space to RGB and set the fill color.
- **K**: Change the stroke color space to CMYK and set the stroke color.
- **k**: Change the fill color space to CMYK and set the fill color.

RGB colors are defined as a percentage between 0 and 1 for the red, green, and blue components, respectively. For example, the following defines a red stroke with a blue fill.

```
0.75 0 0 RG
0 0 0.75 rg
10 w
306 396 m
306 594 l
336 500 l
B
```

Likewise, the CMYK operators take four percentages, one each for cyan, magenta, yellow, and black. The previous stream makes use of the **B** operator, which strokes *and* fills the path.

Path construction operators

Setting the graphics state is like choosing a paintbrush and loading it with paint. The next step is to draw the graphics onto the page. However, instead of a putting a physical paintbrush to the page, we must represent graphics as numerical paths.

PDF path capabilities are surprisingly few:

- **<x> <y> m**: Move the cursor to the specified point.
- **<x> <y> l**: Draw a line from the current position to the specified point.
- **<x1> <y1> <x2> <y2> <x3> <y3> c**: Append a cubic Bézier curve to the current path.
- **h**: Close the current path with a line segment from the current position to the start of the path.

The m operator

The **m** operator moves the graphics cursor (the "paintbrush") to the specified location on the page. This is a very important operation—without it, all path segments would be connected and would begin at the origin.

The l (lowercase L) operator

The **l** operator draws a line from the current point to another point. We've seen this many times in previous sections.

Remember that PDF is a low-level representation of text and graphics, so there is no "underlined text" in a PDF document. There is only text, and lines (as entirely independent entities). Underlining text must be performed manually.

```
.5 w
174 727 m
224 727 l
S
BT
    50 730 Td
    /F0 12 Tf
    (There is no such thing as underlined text!) Tj
ET
```

The c operator

This operator creates a cubic Bézier curve, which is one of the most common ways to represent complex vector graphics. A cubic Bézier curve is defined by four points:

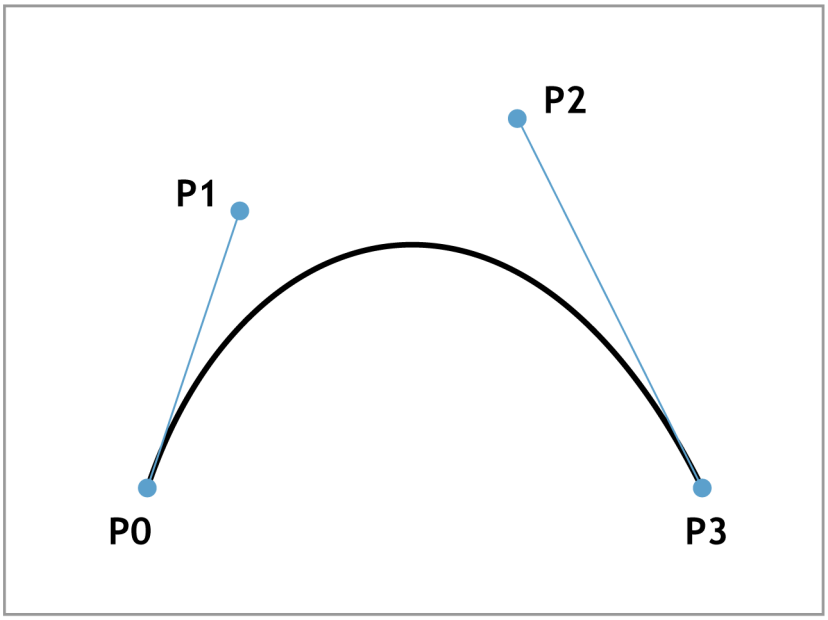

Figure 18: An exemplary Bézier curve

If you've ever used the pen tool in Adobe Illustrator, you should be familiar with Bézier curves. The curve shown in the previous figure can be created in a PDF with the following stream.

```
3 w
250 250 m
300 400 450 450 550 250 c
S
```

The first anchor point is the current position (250, 250), the first control point is (300, 400), the second control point is (450, 450), and the final anchor is (550, 250).

The h operator

The **h** operator closes the current path using a line segment from the current point to the beginning of the path. It takes no arguments. This operator can often be omitted, since many painting operators will automatically close the current path before painting it.

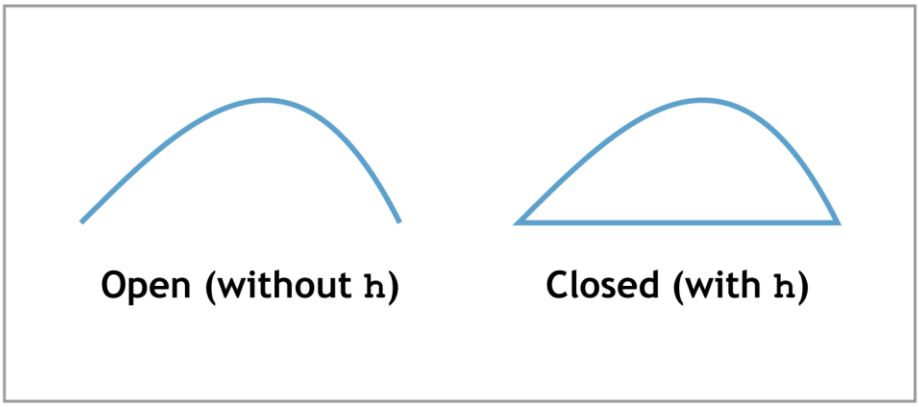

Figure 19: Closing the constructed path with the **h** *operator*

m, **l**, **c**, and **h** are the four main construction operators in a PDF. Again, there are no "shape" operators in the PDF specification—you cannot create a "circle" or a "triangle." However, all shapes can be approximated as a series of lines, Bézier curves, or both. It is up to the PDF editor application to make higher-level shapes available to document authors and to transform them into a sequence of these simple construction operations.

Graphics painting operators

Once you're done constructing a path, you must explicitly draw it with one of the painting operators. This is similar to the **Tj** operator for drawing text. After a painting operator is applied, the constructed path is *finished*—no more painting operators can be applied to it, and another call to a construction operator will begin a new path.

The S and s operators

The **S** and **s** operators paint the stroke of the constructed path using the stroke width set by **w** and the stroke color set by **RG** or **K**. Before applying a stroke to the path, the lowercase version closes the current path with a line segment. This is the exact same behavior as **h S**.

The f operator

The **f** operator fills the constructed path with the current fill color set by **rg** or **k**. The current path *must* be closed before painting the fill, so there is no equivalent to the capital **S** for painting strokes. The following stream creates a blue triangle.

```
0 0 0.75 rg
10 w
306 396 m
306 594 l
```

```
400 500 l
f
```

Remember that painting a path completes the current path. This means the sequence **f** **S** will only fill the path—the **S** applies to a *new* path that has not been constructed yet. To fill *and* stroke a path, we need a dedicated operator.

The B and b operators

The **B** and **b** operators paint *and* stroke the current path. Like **s**, the lowercase **b** closes the path before painting it. However, since filling a path implicitly closes it, the distinction between **B** and **b** can only be seen in the stroke as shown in the following figure.

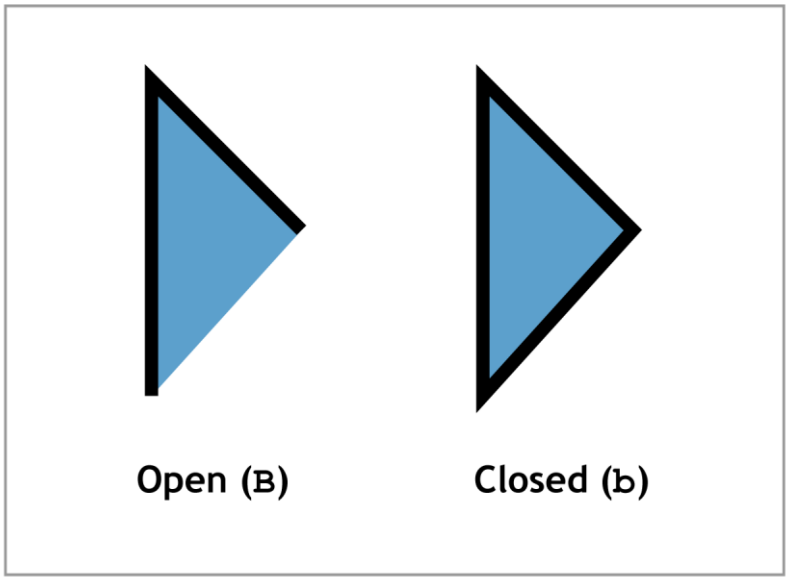

Figure 20: Deciding to open or close a path via a painting operator

The * (asterisk) operators

The fill behavior of **f**, **B**, and **b** are relatively straightforward for simple shapes. Painting fills becomes more complicated when you start working with paths that intersect themselves. For example, consider the following:

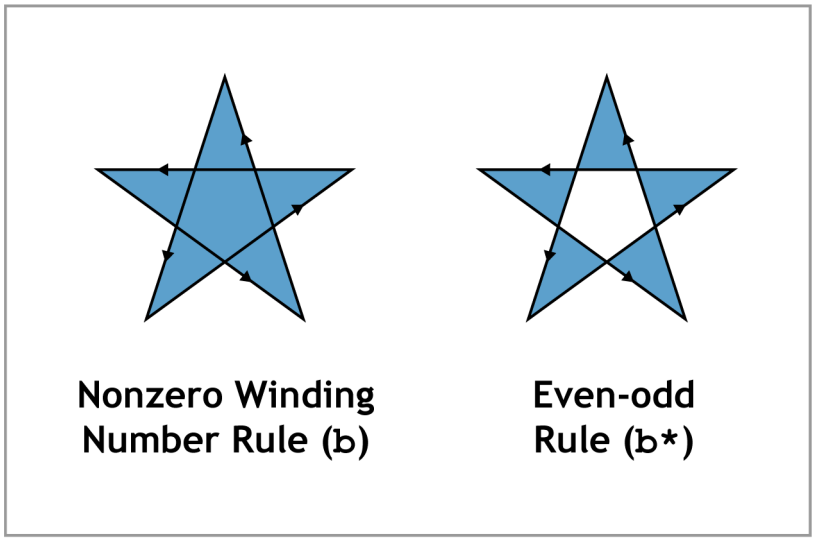

Figure 21: PDF's fill algorithms

As you can see, such a path can be filled using two different methods: the *nonzero winding number rule* or the *even-odd rule*. The technical details of these algorithms are outside the scope of this book, but their effect is readily apparent in the previous diagram.

The fill operators we've seen thus far use the nonzero winding number rule. PDFs have dedicated operators for even-odd rule fills: **`f*`**, **`B*`**, and **`b*`**. Aside from the fill algorithm, these operators work the exact same as their un-asterisked counterparts.

Summary

PDFs were initially designed to be a digital representation of physical paper and ink. The graphics operators presented in this chapter make it possible to represent arbitrary paths as a sequence of lines and curves.

Like their textual counterparts, graphics operators are *procedural*. They mimic the actions an artist would take to draw the same image. This can be intuitive if you're *creating* graphics from scratch, but can become quite complicated if you're trying to manually *edit* an image. For example, it's easy to say something like, "Draw a line from here to there," but it's much harder to say, "Move this box two inches to the left." Once again, this task is left up to PDF editor applications.

Chapter 5 Navigation and Annotations

We've seen how PDFs can accurately represent a physical document in a digital file, but they also provide powerful features that take advantage of their medium. Whereas interactive navigation and editable comments are not possible with a physical book, PDFs make it easy to take notes, share them with others, and bookmark important locations.

This chapter explores the three most important types of user interaction: the document outline, hyperlinks, and text annotations.

Preparations

Before exploring the internal navigation scheme of a PDF, we need a document long enough to demonstrate these interactive features. For our example, all we need to do is add another page. This will also serve as a relevant review of the core PDF objects.

Let's start by adding the page to the document root. The only change here is to add **`6 0 R`** to the **`/Kids`** entry.

```
1 0 obj
<< /Type /Pages
   /Kids [2 0 R 6 0 R]
   /Count 2
>>
endobj
```

Next, we need to create the page object and give it an ID of **`6 0`**. Objects can occur in any order, so you can put this anywhere in the document body.

```
6 0 obj
<< /Type /Page
   /MediaBox [0 0 612 792]
   /Resources 3 0 R
   /Parent 1 0 R
   /Contents [7 0 R]
>>
endobj
```

This looks exactly like our other page (**`2 0 obj`**), but it points to a different content stream (**`7 0 R`**). This page will contain a little bit of textual data.

```
7 0 obj
<< >>
stream
1 0 0 1 50 706 cm
BT
    24 TL
    /F0 36 Tf
    (Page Two) Tj T*
    /F0 12 Tf
    (This is the second page of our document.) Tj
ET
endstream
endobj
```

And that's all we have to do to create another page.

The document outline

Complex PDFs usually come with an interactive table of contents for user-friendly navigation. Internally, this is called a **document outline**. PDF readers typically present this outline as a nested tree that the user can open and close.

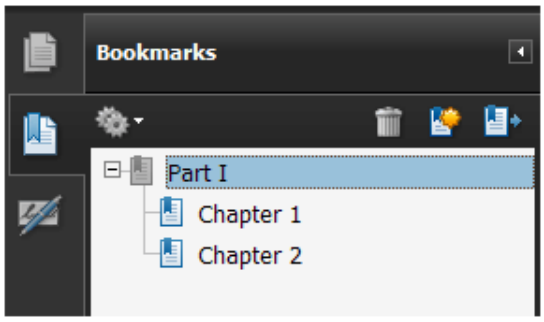

Figure 22: Screenshot of a document outline in Adobe Acrobat Pro

The structure of such a tree is maintained separately from the page objects and content streams of the document. But, like these components, a document outline begins in the catalog object. Add an **`/Outlines`** entry to our existing catalog.

```
5 0 obj
<< /Type /Catalog
   /Pages 1 0 R
   /Outlines 8 0 R
>>
```

```
endobj
```

This points to the root of the document outline. We're going to create a very simple outline that looks exactly like the one shown in the previous figure. It contains a single root node.

```
8 0 obj
<< /First 9 0 R
   /Last 9 0 R
>>
endobj
```

The **/First** and **/Last** entries are a reference to the only top-level node in the outline. In the real world, a PDF would probably have more than one top-level node, but you get the idea. Next, we need to create the following node.

```
9 0 obj
<< /Parent 8 0 R
   /Title (Part I)
   /First 10 0 R
   /Last 11 0 R
   /Dest [2 0 R /Fit]
>>
endobj
```

/Parent points back to the document root. **/Title** is a string literal containing the section title displayed by the PDF reader. **/First** and **/Last** are the same as in the **8 0 obj**—they point to this node's first and last children. Since this node will have two children, **/First** and **/Last** are different.

Finally, the **/Dest** entry defines the destination of the navigation item. A destination is a specific location in the document, specified as a page number, position on the page, and magnification. In this case, we want to display the first page (**2 0 R**) and zoom to fit the entire page in the reader's window (no position can be specified when a page is zoomed to fit). There are several keywords besides **/Fit** that can be used for fine-grained control over a user's interaction with the document. A few of these will be covered shortly.

Next, we need to add the two child nodes to "Part I". The first one will navigate to the top of the second page.

```
10 0 obj
<< /Parent 9 0 R
   /Title (Chapter 1)
   /Next 11 0 R
   /Dest [6 0 R /FitH 792]
>>
endobj
```

This looks very similar to its parent node, but it has no sub-nodes, so **/First** and **/Last** can be omitted. Instead, it needs a **/Next** entry to point to its sibling. The **/FitH** keyword instructs the PDF reader to zoom just enough to make the width of the page fill the width of the window. After **/FitH** is the vertical coordinate to display at the top of the window. Since we wanted to navigate to the top of the page, we specified the height of the page; however, passing a lower value would let you scroll partway down the page. There is a corresponding **/FitV** keyword that fills vertically and offsets from the left of the page.

Finally, we arrive at the last navigation item. This one will point to a destination halfway down the second page.

```
11 0 obj
<< /Parent 9 0 R
   /Title (Chapter 2)
   /Prev 10 0 R
   /Dest [6 0 R /XYZ 0 396 2]
>>
endobj
```

Again, this is just like the previous node, except it has a **/Prev** pointing back to its previous sibling. And, instead of zooming to fit, we manually specified a location (0, 396) and a magnification (2) using the **/XYZ** keyword.

You should now be able to compile your PDF with pdftk and see the document outline (you may need to open the bookmarks panel to see it). You'll notice that the "Part I" node is always closed by default. If you'd like to open it, add a **/Count 2** entry to the top-level node (**9 0 obj**). The **/Count** entry contains the number of visible child nodes. Omitting it hides all child nodes.

To summarize, the document outline consists of a series of navigation items. The **/First**, **/Last**, **/Next**, **/Prev**, and **/Parent** dictionary entries relate items to each other and define the structure of the outline as a whole. Each item also contains a destination to navigate to, which is defined as a page, location, and magnification.

The initial destination

In addition to defining a user-controlled navigation tree, the catalog object can control the initial page to display. This can be accomplished by passing a destination to the **`/OpenAction`** entry in the catalog object.

```
5 0 obj
<< /Type /Catalog
   /Pages 1 0 R
   /Outlines 8 0 R
   /OpenAction [6 0 R /Fit]
>>
endobj
```

Now, when you open the document, the second page (**`6 0 obj`**) will be displayed and the viewer will zoom to fit the entire page.

Hyperlinks

It's also possible to create hyperlinks *within* the document to jump to another destination. PDF hyperlinks aren't like HTML links where the link is directly connected with the text—they are merely rectangular areas placed on top of the page, much like a graphic. They work more like buttons than true hyperlinks.

Hyperlinks are one of many types of *annotations*. Annotations are extra information associated with a particular page. Pages cannot share annotations. The second most common type of annotation is a comment, which we'll look at in a moment.

Annotations are stored in an array under the **`/Annots`** entry in a page object. Our link will be on the second page (**`6 0 obj`**):

```
6 0 obj
<< /Type /Page
   /MediaBox [0 0 612 792]
   /Resources 3 0 R
   /Parent 1 0 R
   /Contents [7 0 R]
   /Annots [12 0 R]
>>
endobj
```

Next we need to create the annotation.

```
12 0 obj
<< /Type /Annot
   /Subtype /Link
   /Dest [2 0 R /Fit]
   /Rect [195 695 248 677]
>>
endobj
```

The **/Subtype** entry tells the PDF reader that this is a hyperlink and not a comment, or one of the other kinds of annotations. Like navigation items, **/Dest** is the destination to jump to when the user clicks the link. And finally, **/Rect** is a rectangle defining the area of the hyperlink. Again, links are not directly associated with the text—they are just an area on the page.

If you don't like the visible border around the hyperlink rectangle, you can get rid of it with: **/Border [0 0 0]**.

Text annotations

Text annotations are user-defined comments associated with a location on a page. They are commonly displayed as "sticky notes" that the user can open and close.

Like hyperlinks, text annotations reside in the **/Annots** array of the page object to which they belong. First, add another object to the **/Annots** array of the second page:

```
6 0 obj
<< /Type /Page
   /MediaBox [0 0 612 792]
   /Resources 3 0 R
   /Parent 1 0 R
   /Contents [7 0 R]
   /Annots [12 0 R 13 0 R]
>>
endobj
```

Then, create the annotation.

```
13 0 obj
<< /Type /Annot
   /Subtype /Text
   /Contents (Hey look! A comment!)
   /Rect [570 0 0 700]
>>
endobj
```

Again, **/Subtype** defines the type of annotation. **/Contents** is the textual content of the annotation, and **/Rect** is the location. This rectangle should place the comment in the upper-right margin of the second page.

Text annotations have a few additional properties that give you more control over their appearance. For example, you can add an **/Open** entry with the value of **true** to the annotation object to make it open by default. You can also change the icon displayed with **/Name /Help**. Other supported icons are: **/Insert**, **/Key**, **/NewParagraph**, **/Note**, and **/Paragraph**.

Aside from **/Link** and **/Text**, there are many other forms of annotations. Some, like **/Line** annotations, are simply more advanced versions of text annotations. But others, like **/Movie** annotations, can associate arbitrary media with a page.

Summary

This chapter presented document outlines, hyperlinks, and text annotations, but this is only a small fraction of the interactive features available in a PDF document. The specification includes more than 20 types of annotations, including everything from printer's marks to file attachments. The complete list of annotations can be found in chapter 8 of Adobe's *PDF Reference*.

Chapter 6 Creating PDFs in C#

Dynamically generating PDFs from a C# program has many potential applications. For example, you could automatically generate custom invoices for a company. Instead of hand-coding all the information, you could pull the company name, price, and items from a database and use them to generate a content stream.

While this could be accomplished by manually outputting each object in the native PDF language, a third-party library makes things much easier. Instead of manipulating strings and saving them to a file, a PDF library lets you interact with a PDF document in an object-oriented fashion. The iTextSharp library is the most popular open-source PDF library for C#.

This chapter will introduce the basics of iTextSharp. It assumes you have a working .NET SDK, which provides the `csc` command for compiling C# code. This chapter is *not* meant to be a comprehensive guide to iTextSharp or dynamic PDF document generation. It merely serves as a quick-start guide to the core features of iTextSharp.

Disclaimer

The iTextSharp library is a C# port of the iText Java library (available from the same website). Some features and aspects of the API were lost during the port to C#. To add to the confusion, these changes are not documented. In fact, as of this writing, there is no iTextSharp documentation to speak of. The only available online API reference is for the Java version, which can be found at http://api.itextpdf.com/itext.

Despite some inconsistencies, this documentation still proves to be a useful reference for C# developers.

Installation

The latest version of iTextSharp can be downloaded from SourceForge. Extract **itextsharp-dll-core-5.2.1.zip** from the package. It should contain a file called **itextsharp.dll**. Any C# projects that require iTextSharp functionality must be compiled against this library.

The basics

Let's start with a simple "Hello, World" example. Add the following to a new file called **HelloWorld.cs** (this file is available at https://bitbucket.org/syncfusion/pdf-succinctly).

```
using System;
using System.IO;
using iTextSharp.text;
using iTextSharp.text.pdf;

public class HelloWorld {
    public static void Main() {
        Document doc = new Document();
        PdfWriter.GetInstance(doc,
            new FileStream("itext.pdf", FileMode.Create)
        );
        doc.Open();
        Paragraph p = new Paragraph("Look! My First PDF!");
        doc.Add(p);
        doc.Close();
        Console.WriteLine("Created a PDF!");
    }
}
```

First, we declare the necessary namespaces for our program. The vast majority of useful classes reside in **`iTextSharp.text`**. The only thing we need from **`iTextSharp.text.pdf`** is **`PdfWriter`**, which will be discussed in a moment.

Then, we create a C# representation of a PDF document with **`new Document()`**. After that, we register the document with an instance of **`PdfWriter`**. This makes sure content we add to the document object is written to the file **itext.pdf**. Next, we open the document with the **`Open()`** method. This *must* be called before any content is added to **`doc`**.

Now that the document is open, we can create a **`Paragraph`** instance and attach some text to it. This may seem odd, considering the fact that PDF does not know what a paragraph is—it only knows **`Tj`**, **`Td`**, and **`T*`**. Fortunately, iTextSharp provides higher-level elements like paragraphs, headers, and lists. This makes it much, much easier to produce dynamic PDFs that include arbitrary runs of text.

Finally, we close the document and output a short success message.

Compiling

Remember, this program needs to be compiled against the iTextSharp library. So, move **itextsharp.dll** into the same folder as **HelloWorld.cs** (alternatively, you can place it in your `PATH`). Then, run the following command to compile the program.

```
csc /reference:itextsharp.dll HelloWorld.cs
```

This creates an executable called **HelloWorld.exe**. When you run it, this program will create a PDF document called **itext.pdf** with the text “Look! My First PDF!” at the top. The internals of this document should have a familiar syntax; however, the streams are compressed and thus unreadable. In addition, several line feeds have been removed, making it harder to orient yourself.

We can make it easier to inspect **itext.pdf** by uncompressing it with pdftk. Its syntax is somewhat unintuitive:

```
pdftk itext.pdf output itext-pretty.pdf uncompress
```

Now, if you open up **itext-pretty.pdf**, you should find a PDF document that is much easier to understand. The trailer dictionary points to the catalog, which should be **`5 0 obj`**. The catalog points to the root page tree (**`3 0 obj`**), and you should be able to find your way from there.

However, iTextSharp added a few new objects to the output file. The trailer has an **`/Info`** entry, which we haven’t seen before. It points to an information dictionary that contains metadata about the document. If you scroll up to **`7 0 obj`**, you’ll see that iTextSharp records itself as the producer application, along with a modification date and a creation date. In addition to these, the PDF specification defines **`/Title`**, **`/Author`**, **`/Subject`**, and **`/Keywords`**. All of these values should be string literals.

There is also a new **`/ID`** entry in the trailer dictionary. It uniquely identifies this version of the document, which can be useful for integrating the document into a workflow.

iTextSharp text objects

As we’ve seen, iTextSharp works on a higher level than PDF text objects. It uses three levels of text objects: chunks, phrases, and paragraphs. These core text objects, along with most of the other available elements, reside in the **`iTextSharp.text`** namespace.

Chunks

Chunks are the smallest unit of text you can work with. They’re useful when you need to style a specific word or phrase differently than the rest of the line. Check out the following code sample.

```
Chunk before = new Chunk("Look! My ");
Chunk middle = new Chunk("First");
middle.SetUnderline(0.5f, -1.5f);
Chunk after = new Chunk(" PDF!");
Paragraph p = new Paragraph();
p.Add(before);
p.Add(middle);
p.Add(after);
doc.Add(p);
```

As you would expect from a higher-level library, you are able to directly associate an underline with a span of text. The **`SetUnderline()`** method takes two arguments: the width of the underline stroke, and the offset distance from the baseline.

However, chunks don't provide much functionality outside of basic styles. They can underline and italicize text, but they can't perform line-level operations. For this, we need the **`Phrase`** class.

Phrases

A **`Phrase`** is a group of **`Chunk`** objects, along with some line-aware functionality. It can wrap lines to a certain length, space lines according to a given leading, and cascade formatting to its chunks. We can see this in action if we change our **`Paragraph`** to a **`Phrase`** and add a few more chunks to it.

```
Phrase p = new Phrase();
p.Add(before);
p.Add(middle);
p.Add(after);
p.Add(new Chunk(" Note that Phrases, unlike Chunks, are line-
aware."));
p.Add(new Chunk(" They can wrap lines to the width of a
document."));
```

Paragraphs

The **`Paragraph`** is iTextSharp*'s* block-level text object. Since it is made up of phrases, it has all the same line-related functionality. In addition, paragraphs have indentation and text-alignment capabilities. We'll take a closer look at these properties in a moment.

Lists

On top of these basic text objects, iTextSharp provides common typesetting elements, including lists. We can create a list with the aptly named **`List`** class, and add items to it by creating **`ListItem`**s.

```
List list = new List(List.UNORDERED);
list.Add(new ListItem("Apples"));
list.Add(new ListItem("Oranges"));
list.Add(new ListItem("Bananas"));
doc.Add(list);
```

By default, lists use a hyphen to denote each item. But, this can be changed with the **`SetListSymbol()`** method.

```
list.SetListSymbol("\u2022");
```

Now, the list uses the Unicode bullet character. Note that this method must be called *before* adding any items to the list.

Formatting a document

Anything you can do in the native PDF language, you should be able to do in iTextSharp. In this section, we'll look at iTextSharp's version of media boxes, colors, fonts, and paragraph formatting.

Document dimensions

The **`/MediaBox`** entry in a page object defines the size of that page. In iTextSharp, this is accomplished by passing a **`Rectangle`** containing the dimensions to the **`Document`** constructor.

```
Rectangle mediaBox = new Rectangle(300, 400);
Document doc = new Document(mediaBox);
```

Now, if you open the document created by **HelloWorld.cs**, you'll find a **`/MediaBox`** entry of **`[0 0 300 400]`**. iTextSharp also ships with a useful **`PageSize`** class that contains constants for common paper dimensions. For example, **`PageSize.A5`** is a 420 × 595 **`Rectangle`**.

The **Document** constructor also accepts four numerical parameters defining the left, right, top, and bottom margins of the page, respectively. For example, a document created by the following sample will have no page margins at all.

```
Document doc = new Document(mediaBox, 0, 0, 0, 0);
```

Colors

iTextSharp uses the **BaseColor** class to represent colors as RGB components. We can define the background color of a page by assigning an instance of **BaseColor** to the **BackgroundColor** field of **Rectangle**.

```
mediaBox.BackgroundColor = new BaseColor(255, 255, 240);
```

Of course, iTextSharp also supports other common color spaces, including: **CMYKColor**, **GrayColor**, and **SpotColor**. There is even a **PatternColor** that can use a bitmap to paint a fill.

Selecting fonts

The **Font** class combines a font face, size, and color into a reusable style. However, working with the **Font** class directly can be a bit complicated, so iTextSharp provides a factory method for creating a **Font** instance.

```
BaseColor red = new BaseColor(255, 0, 0);
Font apple_font = FontFactory.GetFont(
    FontFactory.HELVETICA_OBLIQUE, 12, red
);
```

This creates a red, oblique Helvetica font that is 12 points high. Now we can pass this font instance to any text object. For example, we'll turn the "Apple" text in our list red.

```
list.Add(new ListItem("Apples", apple_font));
```

All of the other text objects (chunks, phrases, and paragraphs) also take a font as their second parameter. However, the **List** object has some unexpected behavior—*all* the bullets are red.

Figure 23: The ***apple_font*** *applied to all bullets*

We can fix this by explicitly setting the font of the bullet. First, we'll create the font.

```
BaseColor black = new BaseColor(0, 0, 0);
Font symbol_font = FontFactory.GetFont(
    FontFactory.HELVETICA, 12, black
);
```

Then we'll use this font in a new **Chunk** and assign it to the list symbol.

```
list.ListSymbol = new Chunk("\u2022", symbol_font);
```

Notice that we had to use the **ListSymbol** field instead of the **SetListSymbol()** method. While **SetListSymbol()** is documented to accept either a string or a **Chunk**, the C# port seems to have lost this functionality. Instead, the **ListSymbol** field accepts a **Chunk**. This is a common "idiom" used by the developers porting the library.

Custom fonts

Using the same method, you can also embed fonts into the document from your hard drive. The ability to embed fonts is a big reason behind the prevalence of PDFs. Since they don't rely on client-side resources, they can ensure that content appears the same on *any* computer, regardless of their installed fonts.

Fonts can be embedded by directly accessing the font file as follows.

```
Font chunk_font = FontFactory.GetFont(
    "C:\\WINDOWS\\Fonts\\Chunkfive.otf",
    BaseFont.WINANSI,
    BaseFont.EMBEDDED
);
```

This embeds a font residing at **`C:\Windows\Fonts\Chunkfive.otf`**, which is a copy of Chunk Five Roman obtained from FontSquirrel.com. **`WINANSI`** is the font's encoding, and **`EMBEDDED`** is a Boolean flag that tells the PDF to embed the font.

Formatting text blocks

Now that we have font selection under our belts, we can move on to block-level formatting. This includes: leading, spacing before and after, left and right indentation, and alignment:

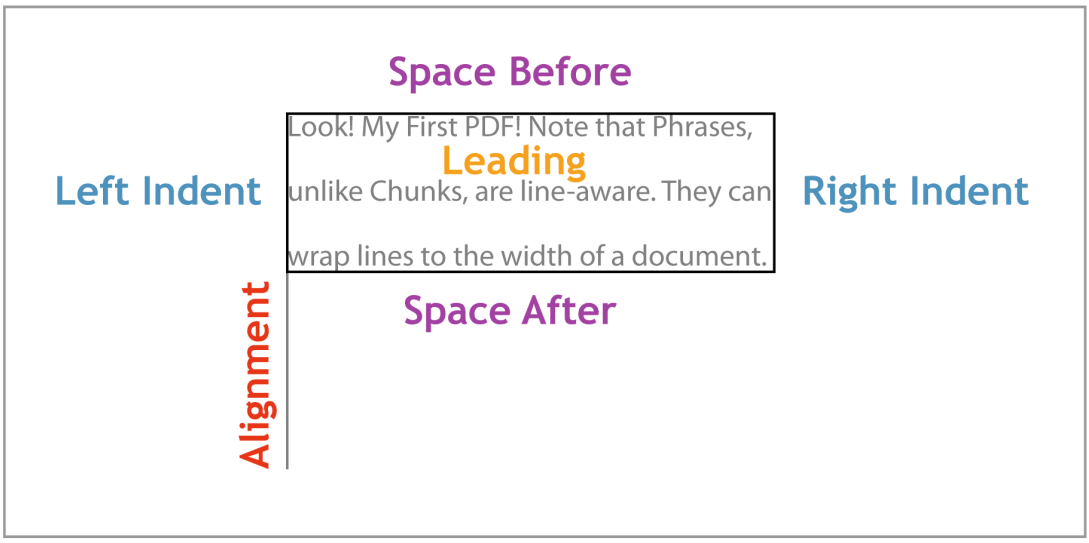

Figure 24: iTextSharp's block-level formatting fields

Let's jump right in. Change the **`Phrase`** back to a **`Paragraph`**, then set all four of the following fields.

```
Paragraph p = new Paragraph();
p.Leading = 26f;
p.SpacingAfter = 10f;
p.IndentationLeft = 10f;
p.Alignment = Element.ALIGN_CENTER;
```

Leading, spacing, and indentation all accept a float, but alignment requires a constant defined in the **`Element`** class. Other alignment options are: **`ALIGN_LEFT`**, **`ALIGN_RIGHT`**, and **`ALIGN_JUSTIFIED`**. If you justify the text, uncompress the file, and sift through the content stream, you'll discover that iTextSharp justifies text using the same `Tc` and `Tw` method discussed near the beginning of the book.

Summary

This chapter provided a brief tour of the iTextSharp library. You should be well on your way to dynamically creating PDF documents from C# and leveraging them against the .NET framework. I leave the rest of iTextSharp for you to explore. On a related note, Syncfusion also offers a commercial PDF library that provides a high-level object model to create, modify, convert, and view PDF files from .NET applications. Online samples of Syncfusion's PDF library can be found here.

Conclusion

This book introduced the fundamental components for the digital representation of paged documents: text, graphics, and navigation. Hopefully, you're now more than comfortable directly editing these elements in PDF files. We were even able to explore the iTextSharp library which gave us a high-level API for the low-level PDF format.

We've covered dozens of objects and operators, but this is merely the core of PDF's capabilities. PDF documents also boast Form XObjects (reusable symbols), digital signatures, native 3-D artwork, and extensive document interchange capabilities, including arbitrary metadata, private application data, and tagged content for easier data extraction and dynamic reflowing of pages.

All of these topics are thoroughly covered in Adobe's *PDF Reference, Sixth Edition, version 1.7*. This indispensable reference will prove to be a trustworthy guide should you choose to continue your journey through the native PDF language.