Contents

List of Figures

List of Tables

List of Examples

Chapter 1: Introduction

1.1 PDF Anatomy

Normally, a PDF file contains a header, a body, a cross-reference (Xref) table, and a trailer. The body part contains a series of object descriptions – each object has an ID and its byte offsets is recorded in the Xref table so that a viewer application can find it quickly. The relationships among these objects are saved in the properties by referring to their IDs, which are unique integers. Data for text, graphics, images, fonts, etc., are appended to the objects as so-called streams. Please refer to the Adobe site at http://partners.adobe.com/asn/developer for more information about the structure of PDF files and PDF syntax.

A software generating PDF output must conform to the strict syntax of PDF as a page description language, and a tool library would be of great help, because of the complexity involved in content encoding, object offset calculation, compression, page markup commands, graphical operations, and the manipulation of a variety of files that could be used to import data from: images, fonts, and vector graphics alike.

When reading an arbitrary PDF file, the parser program must be able to isolate correct data blocks based on the information contained in the Xref tables. A so-called "linearized" file is specially rearranged to make accessing the first page as quick as possible, while an "incrementally updated" file contain new definitions for existing objects and new Xref tables. Both added complexity to the file's structure.

1.2 PDF Everywhere Approach

PDF Everywhere uses an object-oriented approach for the task of generating PDF content in memory. This approach is good for reusability and extensibility. Many of the classes are created for corresponding object types (semantically – since most objects are just "dictionaries" that are key/value pairs) in a PDF file.

PDF Everywhere views a PDF document as a collection of objects. The document consists of varying number of pages, each containing text blocks and graphics **content**, as well as optional, interactive **component**s, i.e. annotations (and form fields). A content object or a component normally requires some **resource**s, such as font, image, and texture. The document's **logical** and navigational information is recorded in some data structures known as catalog, bookmark tree, pages tree, security dictionary, etc., which have no visible representation. In addition, some objects are dependent of other objects, storing information required by the latter. In summary, PDF Everywhere classifies the modules into the following five categories: logical, content, resource, component, and **other**.

Instances of the classes are organized into a hierarchy with a single root node. All classes for the tree nodes are derived from a same class, `PDFTreeNode`, although some classes are not its direct derivation. The inheritance relationships are shown in Figure 1-1, although some other classes are omitted.

A `PDFDoc` represents a fresh PDF file created in memory. It is the root of the object tree, and is responsible of document-level tasks such as printing, encrypting, and importing pages from another file. Each node on the tree is responsible of generating its own PDF code, processing its own data (compression, encryption, etc.), and creating XML representation of itself.

A `PDFFile` (not shown) object would represent an existing PDF file. It is responsible for identifying the object blocks and has the ability to traverse all these objects to perform repetitive tasks such as printing and encrypting. Each object block is eventually parsed and turned into a specific data type, namely name, string, dictionary, array, etc. Parsing of the data is done recursively.

1.3 Creating a PDF File

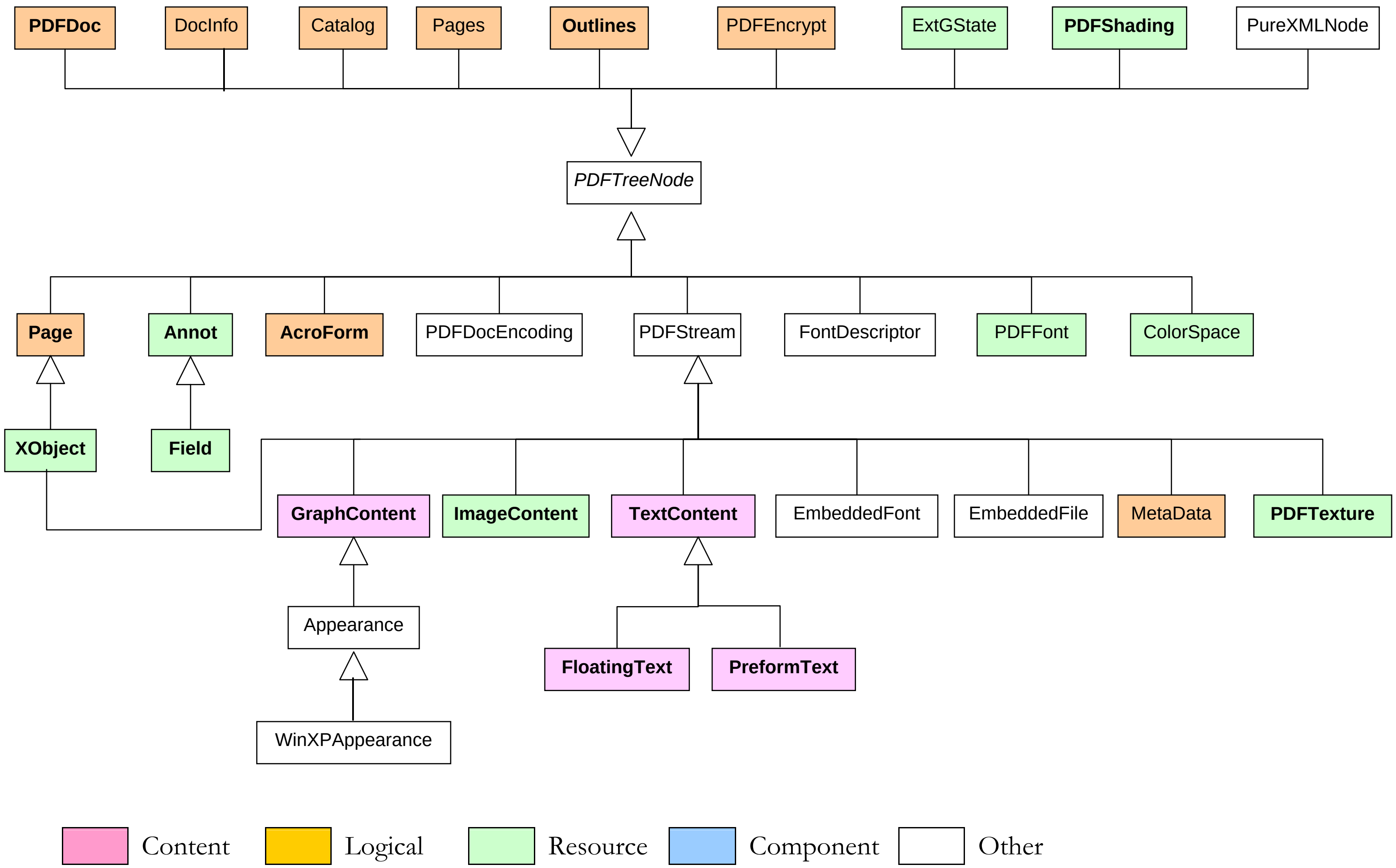
- **The Old and the New Way**

If you are a new user of PDF Everywhere toolkit, skip this section. The version 2.x way of creating a PDF file goes like this:

```
require "pdfever.pl";
init( 1 ); # No input from keyboard
# Set output file name; encrypt file and allow printing
startPDF( { FileName => 'sample.pdf', Print => 1 } );
new Page;
# Page content objects are created here
writePDF( ); # Print to STDOUT
```

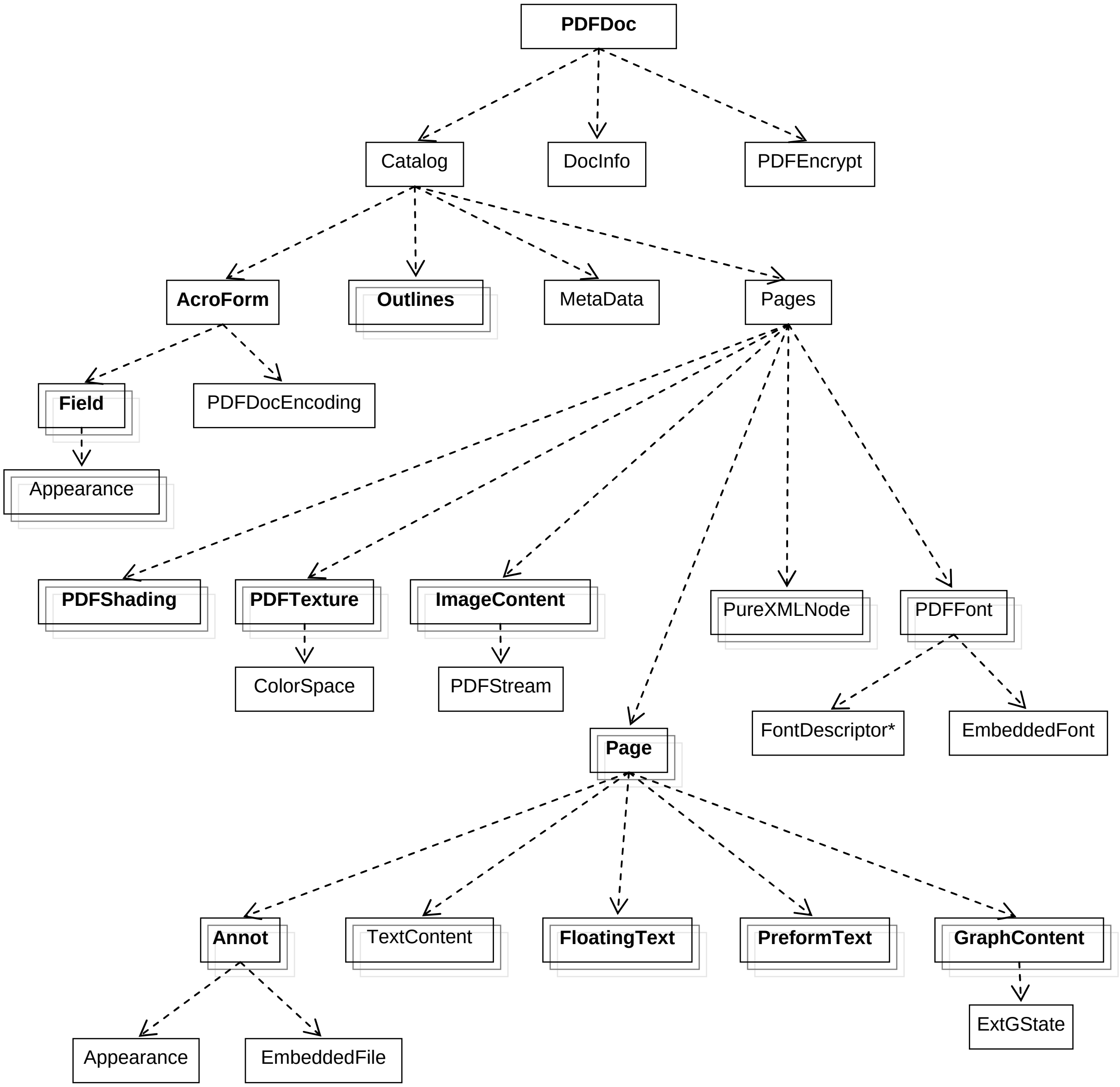
The function `init` does some initiation, while `startPDF` and `writePDF` begins and writes a PDF file. This approach has these drawbacks: (1) It pollutes the namespace of the surrounding program; (2) There can be only one, global PDF document; (3) The document can be written only once; (4) Output file name, encryption setting, document information, etc., can not be changed once set. Fortunately, all have been avoided in version 3.0. The new way is like this:

```
use PDF;
my $doc = new PDFDoc( );
my $pg = $doc->newPage( );
# Page content objects are created here
$doc->writePDF( 'sample.pdf' ); # Print to a disk file
$doc->setTitle( 'Some title' ); # Set document title
$doc->setSecurity( { Print => 1 } ); # Set security
$doc->writePDF( 'sample2.pdf' ); # Print to another file
```



Font face: italic – abstract class; bold – concrete, public classes; normal – concrete, private classes.

Figure 1-1. Classes derived from PDFTreeNode



Legends: *(Class name with a * indicates a private class)*

Class Multiple instances may be present under the parent node.

Class Singletons; at most one instance can be present under the parent node.

- - - -> Dependency.

Figure 1-2. A typical PDF document object tree

A PDF file should have a skeleton that contains a “Root”, an “Info” block, and a file ID. The “Root” is the “Catalog” of the document, which contains an “Outlines” entry as the root for the bookmark (outlines) tree, a “Pages” entry as the root for the page tree. The “Info” block contains information about document title, creator application, creation time, etc. If the document is protected via password encryption, the security information is contained in a data block referred to via the “Encrypt” entry in the trailer. A `PDFDoc` object in this library automatically creates these document-level objects, which used to the responsibility of the function `startPDF` in older versions. These objects are created and managed **automatically**, so you must not create them manually.

The `PDFDoc` object also acts as an object factory. For example, you would call:

```
my $pg = $doc->newPage( );
```

to create a new page, rather than the v2.x approach:

```
my $pg = new Page( );      # It is actually still valid
```

This signifies an important improvement: in version 3.0: a program can operate on more than one PDF document simultaneously, whereas in the older versions it is assumed there is a single global document to work on. For backward compatibility, the old way is still supported: a global variable `$PDF::root` points to current active `PDFDoc` object.

The difference between the above approaches is that the v2.x way links the newly created objects to **currently active document**. When multiple documents are being worked on, you can use function `choose` to switch among them (thus changing current document to insert objects into):

```
my $doc2 = new PDFDoc( );
new Page( );   # Linked to $doc2
PDF::choose( $doc );
new Page( );   # Linked to $doc
```

To create your first PDF file, please start with the example on page 2 and insert more code that creates the actual page contents.

1. Create a new page by using the method `newPage`;
2. Create text blocks by calling `newTextBox` with the following parameters: a `Rect` object to define the box, a string as the text content, and a hash reference to define properties;
3. Finally, call the `writePDF` method with a file name as the argument to generate a PDF file.

For example, a new text box can be created this way:

```
my $txtbox = $doc->newTextBox( new Rect( 20, 20, 300, 300 ), "Sample
text", { FontSize => 28, VerticalAlign => 'Middle', TextJustify =>
'Center', Color => 'green' } );
```

which is equivalent to the following XML code:

```
<Text Rect="20, 20, 300, 300" FontSize="28" VerticalAlign="Middle"
TextJustify="Center" Color="green"><![CDATA[Sample text]]></Text>
```

You can compose the text content before creating this box. The available properties can be found in later chapters.

To show an image, you need to create a graphics layer using newGraph method, and then place the image inside this layer. The coordinates are relative to the origin of this layer; that is, they are measured from the bottom-left corner of the hosting layer if the layer isn't aligned to the page origin:

```
my $graph = $doc->newGraph( );
$graph->showImage( $doc->newImage( 'mylogo.jpg' ), 20, 30 );
```

▪ XML Authoring

XML is applied as an authoring tool, not an alternative method of representing the content and structure of a PDF file. The XML code used to define a PDF document must be embraced by a pair of `<PDF>` tag (which is where the document properties such as title, page size, viewer mode, etc. are set), and graphics layer, text box, and annotations, etc. must be defined within the tags of the hosting page. For example:

```
<?xml version="1.0" standalone="yes"?>
<PDF PageSize="Letter" Print="1">
     <Page>
     <!–Page content tags here -->
     </Page>
</PDF>
```

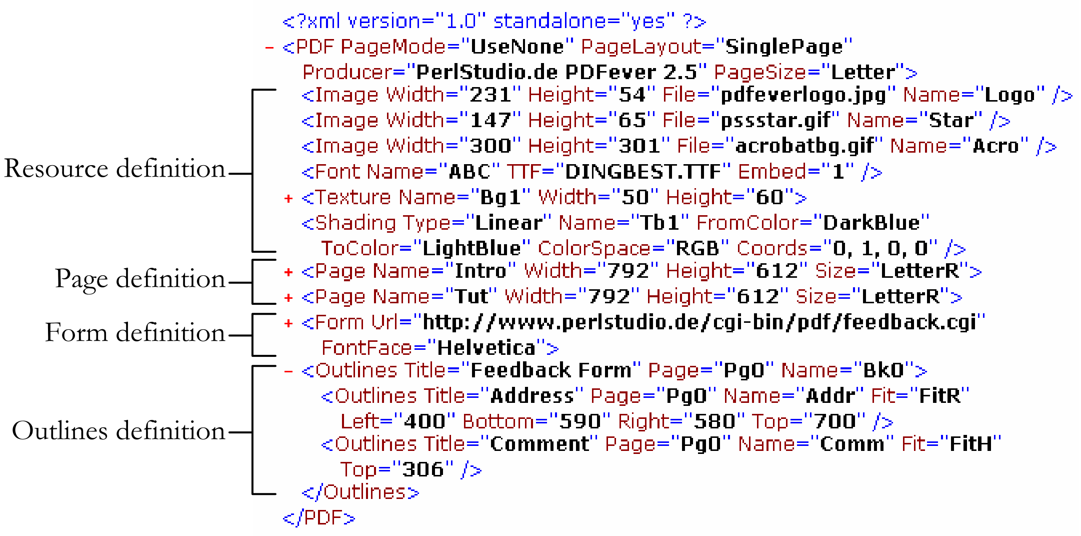
Most functionality can be accessed via an XML tag with appropriate properties. Later in this documentation, we'll list the corresponding XML tags for objects and functions. To convert the document to PDF, the version 2.x way is:

```
require "pdfever.pl";
importXML( );
writePDF( );
```

The new version 3.0 way is:

```
use PDF;
PDFDoc->importXML( 'sample.xml' )->writePDF( 'sample.pdf' );
```

Objects may refer to each other by name, such as when showing an image: a single image can be shown on many pages, and the program needs a name to find an image. In XML, to guarantee a requested name has been defined, groups of tags must be arranged in order, see Figure 1-3. This is strengthened by PDF Everywhere in two ways: first, any object must be assigned a name; second, the document object tree is prioritized so that the output of tags is always in correct order.



```
  <?xml version="1.0" standalone="yes" ?>
- <PDF PageMode="UseNone" PageLayout="SinglePage"
    Producer="PerlStudio.de PDFever 2.5" PageSize="Letter">
    <Image Width="231" Height="54" File="pdfeverlogo.jpg" Name="Logo" />
    <Image Width="147" Height="65" File="pssstar.gif" Name="Star" />
    <Image Width="300" Height="301" File="acrobatbg.gif" Name="Acro" />
    <Font Name="ABC" TTF="DINGBEST.TTF" Embed="1" />
  + <Texture Name="Bg1" Width="50" Height="60">
    <Shading Type="Linear" Name="Tb1" FromColor="DarkBlue"
      ToColor="LightBlue" ColorSpace="RGB" Coords="0, 1, 0, 0" />
  + <Page Name="Intro" Width="792" Height="612" Size="LetterR">
  + <Page Name="Tut" Width="792" Height="612" Size="LetterR">
  + <Form Url="http://www.perlstudio.de/cgi-bin/pdf/feedback.cgi"
      FontFace="Helvetica">
  - <Outlines Title="Feedback Form" Page="Pg0" Name="Bk0">
      <Outlines Title="Address" Page="Pg0" Name="Addr" Fit="FitR"
        Left="400" Bottom="590" Right="580" Top="700" />
      <Outlines Title="Comment" Page="Pg0" Name="Comm" Fit="FitH"
        Top="306" />
    </Outlines>
  </PDF>
```

Figure 1-3. A sample XML document for generating PDF.

As shown above, the XML document used by PDF Everywhere should be organized in a fixed order:

- **Resource definition**: Images, fonts, etc.
- **Page definition**: One block for each page, which may contain graphics, text boxes, and annotations (not necessarily in that order).
- **Form definition**: If your PDF file contains a form, the definition of form fields are shown here, even though the fields may be distributed on different pages.
- **Outlines (bookmarks) definition**: All outline entries are stored in this last section.

- **PDF Importing: Single Page**

To import one page from an existing PDF file, provided that the file is not encrypted, you need to create a `PDFFile` object to represent the file, and then instantiate a `Page` by specifying the page number (starting from 0):

```
use PDF;
my $doc = new PDFDoc( );
my $pdf = new PDFFile('samp1.pdf');
my  $pg  =  $doc->newPage(  {  ImportSource  =>  $pdf,  ImportPage  =>
7 } ); # Import page 8
$doc->writePDF( );
```

You cannot import from any encrypted file. In that case, you must decrypt the file using its own password (if not defined, use user password in stead; if neither password is defined, use empty string).

As a restriction, form fields, articles, and named destinations are among the objects that are ***not*** imported. However, the page contents are imported as-is.

Pages imported from a same `PDFFile` object may use same resources (images, fonts, etc.). PDF Everywhere avoid duplicate copies of such resources by keeping track of all the objects referenced before, thus your file doesn't grow out of sense when multiple pages are imported this way. However, this mechanism indicates that a same page ***cannot*** be imported more than once. To do that, you need to create a new `PDFFile` from the same disk file, but this way the duplication of identical resources will be inevitable.

The imported page is not locked; you can add more contents to that page. By default, newly added contents will show up on top of the original page. To avoid blocking the original contents, for example when you just want to add a background design via a template (see below), you can turn on the "Always on top" property as shown in the code section above.

▪ PDF Importing: Multiple Pages

Multiple pages from an existing file can be imported at once, but these pages are locked and inaccessible via the `Page` class.

```
my @pgs = 2..4;
$doc->importPages( $pdf, @pgs );# Import pages 3 through 5
```

The first argument to the function `importPages` is a `PDFFile` object, the other arguments are treated as zero-based page number values. Note that the range ***cannot*** contain duplicate page numbers, or the PDF output will not be correct.

▪ Template and XObject

By "template" we refer to two things. Firstly, a set of content-type objects (graphics content, text content, etc.) maintained for a PDF document that are automatically duplicated when a new page is created. Any content can be marked as part of the template collection at creation time, or the object can be added later via a function call. Objects can be dropped from the template as well. This template is called "internal template".

Secondly, the template collection can be built via a piece of XML code, referred to as an "external template", as the XML definition can be stored in a separate XML file. The XML code can also be assigned to a scalar variable, and then used for "initializing" new pages, in which process the "internal template" is modified.

Each page can be loaded with a different set of template objects or a different template definition in XML form. The change remains in effect until a new page overwrites it. The tutorial document contains step-by-step examples of templates.

There is also another concept of "template": a PDF file that can be loaded dynamically to format a button's appearance. This is Adobe's definition, which could better be called "invisible pages", and it differs from what we mean here.

Chapter 3 has more information about using templates.

`XObject` extends the concept of template. In PDF specification, the term "`XObject`" is used to refer to images, form field appearances, embedded PostScript code, etc. Here we mean a page that is not counted as a regular page, but as an "image" that can be painted like any other images. The only difference is that a regular image is pixel-based, but an `XObject` is vector-based thus is truly scalable.

An `XObject` can be modeled after an already-created page, or be forged from an external XML template.

- **Table-based layout design**

Version 3 introduces layout design by defining tables (grids) on a page. A page is visualized as a piece of paper, whose physical size is defined in its "`MediaBox`" attribute; PDF Everywhere uses the term "margin" to define the trim (crop) box, and the term padding to define the art box (for definitions of media, trim, and art boxes, please refer to PDF Reference, Adobe). There is no correlation between the latter two.

The art box (default to be the entire page) is the only cell of a default 1 × 1 table. You can draw tables inside this cell, and more nested tables inside any cell or across multiple cells. There isn't a row-by-row iteration mechanism, so you are free to refer to any cells in any order. The table-based layout design favors XML authoring.

For example, you can first create a 3 × 2 table, and then create a textbox using the right-most column, anchored to the top-right (northeast) corner of this column. Text placed inside this box wraps in the column.

In a Perl script, you can get the coordinates of the rectangular region by calling `getCell`:

```
my $pg = $doc->newPage( 'A4R' );
my $graph = $doc->newGraph( );
# Now create a 3x3 table that measures 300x400 pixels
$graph->drawGrid( 300, 400, 0, { 'XGrids' => 3, 'YGrids' => 3 };
my $str = q{the quick brown fox};
# Now create a box at row 1, cell 2, row span 1, col span 2
my $text = $doc->newText( $doc->getCell( 1, 2, 1, 2 ), $str );
```

Tables can be nested: when a new table is created, current table is disabled (a stack is used to manage these tables). In addition, tables are defined for current working page only. When a new page is created, the table definitions are reset.

1.4 Images, Fonts, and Colors

Images, fonts, and colors are resources, referenced by name. In fact, each and everything `PDFTreeNode` object is assigned a name automatically, if a name is not defined.

PDF Everywhere is able to parse and convert a variety of different image formats, including JPEG (regular JFIF and digital camera Exif), GIF, PNG, BMP, WMF, PCX, TGA and TIF. Certain variations of these formats are not supported, notably those having multiple images (and animations), indexed images with non-standard color depth, or interlacing. An image can be assigned with a name.

This library can use a TrueType or Type 1 font to format text. For a Type 1 font, both PFB (Printer Font Format, B) and AFM (Adobe Font Metrics) or PFM (Printer Font Metrics) files must be supplied. For a TrueType font, only the TTF file is needed. Note that the program must read font data from these disk files; it cannot interact with host operating system to get font data, since when the fonts are embedded the original font files are required.

The library provides support for all 14 built-in typefaces that comes with Acrobat. Arial is treated as Helvetica, Times New Roman as Times, and Courier New as Courier. A custom font can be assigned a name, or the name can be retrieved from font file.

To embed a custom font, you call function `useTTF` for TrueType fonts and `usePFB` for Type 1 fonts. Both fonts return the font name found in the font file (after stripping non-word characters). For example:

```
my $f1 = $doc->useTTF( 'abc.ttf', 1 );   # Embed this font
$doc->newTextBox( $pg->getArtBox( ), $str, { FontFace => $f1 } );   # Use the art box of the page
my $f2 = $doc->usePFB( 'abc.pfb', 'abc.afm', 1 );
$graph->showText( '2in', '2in', $str, { FontFace => $f2 } );
```

PDF Everywhere can recognize all HTML color names (letter case is insignificant). Besides, it supports three types of color space: `RGB`, `CMYK`, and `Gray`. A global color space is set for a document. Custom names are supported.

Each font instance (an object is created for a font) and custom color name is managed by a PDF document it is defined for. For example, if you want to define a color named "`Pantone 820`", you must define it for each `PDFDoc` object you create in a script. Details of doing so will be addressed later.

Color-related functions are packaged in class `Color`, which is not used to create as instances but to provide these functions. However, to register a new color, you need to call function `registerColor` on a `PDFDoc` object:

```
$graph->setColor( 'red' );   # Set as foreground color
$doc->registerColor( 'mycolor', '6699CC', 'CMYK' );
$graph->setColor( 'mycolor', 1 ); # Set as background color
$graph->drawCircle( 100, 1 ); # Filled circle, r = 100
```

1.5 Coordination System and Shapes

Position and size of various objects, such as text blocks, vector graphics, images, and form fields, are defined in a coordination system with its origin set to the bottom-left corner of the page. The X-axis extends to the right and Y-axis to the top, as shown in the illustration in Figure 1-4.

In Acrobat, images (and XObjects) are internally treated as a single pixel that is placed onto a page at given coordinates then stretched to specified width and height. Therefore, the coordinates specify the bottom-left corner of the image. However, a text block starts at the top-left corner of the first character, and then extends downwards.

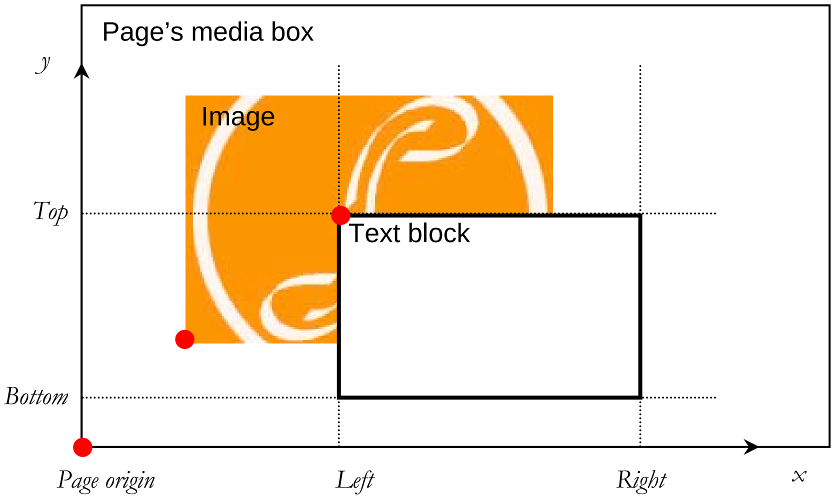


Figure 1-4. Coordination system (red dots shows the origin of the object).

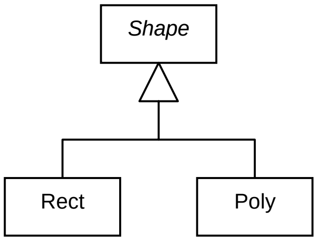


Figure 1-5. Class diagram for shapes.

Unit for measurements is “point”, which is 1/72 or an inch. For you convenience, dimensional values with units as postfixes are understood, such as “0.5in” and “3.8cm”.

The following units are recognized: in, cm, mm, px (pixel), pt (point). Pixel is the same as point, although in other software they may be treated differently.

Rectangles are frequently used in many of the modules. Most objects require a specific region in the page coordinate system to show the contents and graphics. A Rect can be resized or moved after creation. Such operations are typically used by TextContent and Annot objects to get properly rendered.

Polygon is another geometry defined in PDF Everywhere, and it is used for text boxes to confine text flow. Both `Rect` and `Poly` are derived from a common parent class called `Shape`, as shown in Figure 1-5.

Both `Rect` and `Poly` have the following properties: `Left`, `Bottom`, `Right`, `Top`. You can access the properties directly. The values returns by a `Poly` would those of the bounding `Rect` of that polygon.

1.6 Layers and Transparency

Each text box or graphics block introduces a new layer, if we think the page as a stack of such layers overlapped with each other. Each layer is defined in the rectangular region passed to the class constructor. For graphics layers, the default, non-argument constructor creates a layer covering the entire page.

A layer is transparent unless some drawings are made: line drawing, area filling, text showing, and image placing. The areas that are actually covered by these graphics operations can be assigned with different transparency (opacity) and blending mode settings, as supported in Acrobat 5.x. Opacity is set on a scale of 1 to 100 as percentage values. Blending mode affects how the current layer blends with the pixels already painted onto the page media. The blending modes are the same as those defined in PhotoShop and other Adobe-brand graphics applications.

1.7 Limitations

PDF Everywhere doesn't support Unicode and CJK (fareast) character sets. Besides, there is no native support from import from HTML or RTF (Rich Type Format) files.

1.8 Conventions

Code and class/attribute/variable names presented in this document are displayed in fixed-pitch font. Function prototypes are specified in the form of

```
Classname::funcitonName( $param1: type, $param2: type ): type
```

Square brackets `[ ]` are used to denote optional parameters or arrays. Types include `Integer`, `Float`, `String`, `Size`, `Color`, `Font`, `ARRAY`, `HASH`, etc. `ARRAY` and `HASH` means references to an array or a hash. `Size`, `Color`, `Font`, and `Name` are special strings.

Most predefined names are case sensitive, such as font names and texture types.

Chapter 2: PDF Document

2.1 Document Object Tree

As mentioned in Chapter 1, the objects that make up a PDF document are organized into a tree hierarchy (see Figure 2-1), thus all these objects have a same ancestor: `PDFTreeNode`[1]. The abstract class `PDFTreeNode` defines the following properties:

Table 2-1. Attributes common to all document object tree nodes.

Attributes	Type	Explanation
`ObjId`	Integer	A unique integer for identification.
`next`	Reference	Reference to the next sibling node
`son`	Reference	Reference to the first child node
`last`	Reference	Reference to the last child node
`par`	Reference	Reference to the parent node
`Name`	String	A unique name of the object
`EncKey`	String	A fixed-length (32 bytes) string used for encryption

Objects imported from external PDF files are duplicates of original objects, and they are managed by the `PDFDoc` object but are **NOT** linked to the document object tree. All native and imported objects are assigned consecutive ID numbers, starting from 1, when the document is being printed (the numbering sequence is reset when it is printed again).

- **Accessor methods**

```
PDFTreeNode::firstChild( ): PDFTreeNode
PDFTreeNode::lastChild( ): PDFTreeNode
PDFTreeNode::prevSibling( ): PDFTreeNode
PDFTreeNode::nextSibling( ): PDFTreeNode
```

These functions return the value of the fields `first`, `last`, `prev`, and `next`, respectively.

- **Addition and removal**

```
PDFTreeNode::appendChild( $node: PDFTreeNode ): PDFTreeNode
PDFTreeNode::appendSibling( $node: PDFTreeNode ): PDFTreeNode
PDFTreeNode::prependSibling( $node: PDFTreeNode ): PDFTreeNode
PDFTreeNode::deleteChild( $node: PDFTreeNode ): PDFTreeNode
PDFTreeNode::children( ): PDFTreeNode[]
```

[1] In version 2.x, the generalization was done by pushing certain methods into the UNIVERSAL package. This approach has been discarded.

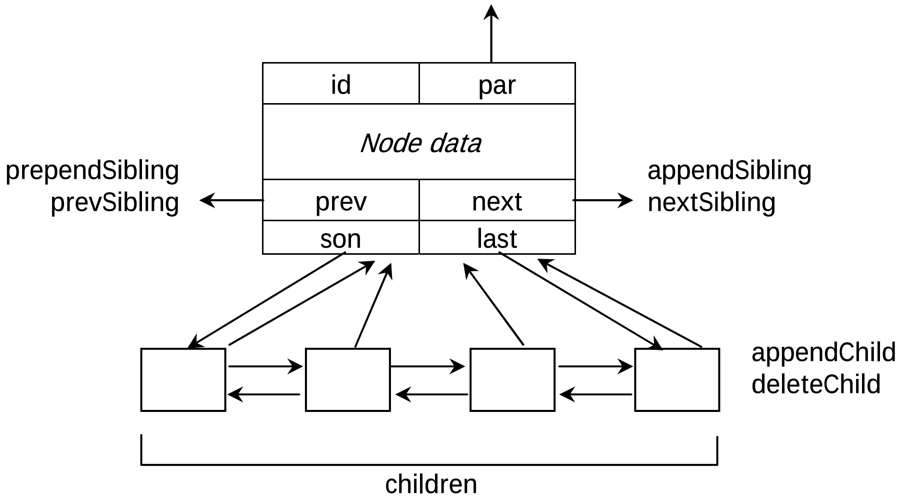


Figure 2-1. Tree node in the document.

These functions add or remove a node to the tree then return that node, except for function `children` that returns the children nodes as a list.

When a node is added to another node as a child, it is inserted into the prioritized linked list of children. Priorities are assigned to classes, so that document-level objects and resources always show up at the front side of the list. This arrangement is for the purpose of correct XML output, since when the XML is converted to PDF these objects must be already defined before other nodes, such as contents and outlines, are created, as they need references to these nodes.

- **Name and ID**

```
PDFTreeNode::getName( ): String
PDFTreeNode::setName( $name: String ): String
PDFTreeNode::getObjId( ): Integer
```

Each node must have a unique name. If the name is not specified in your program, they are named automatically. XML code requires a by-name referencing mechanism, which is also supported in Perl scripts.

When a name is set for a node, the actual name used is determined by the PDF document tree the node is added to, which means the name might be changed (by incrementing the string literal). This name is returned from the function call `setName`.

Function `getObjId`, by default, returns the value of the `ObjId` field, but it is not always the case. When pages and outlines are imported from an external PDF file, corresponding `Page` and `Outlines` objects are created but they act as delegates; that is, they are linked to the document tree but the actual code output and actual IDs are determined by the imported objects they delegate. Therefore, the function call returns the IDs of the imported objects. Note that the IDs are NOT defined until the document is generating PDF output.

- **Encryption and output**

```
PDFTreeNode::encrypt( )
PDFTreeNode::makeCode( )
PDFTreeNode::cleanup( )
PDFTreeNode::startXML( )
PDFTreeNode::endXML( )
PDFTreeNode::newFromXML( ): PDFTreeNode
```

These functions by default have empty implementations. Actual implementations vary in derived classes. Function `encrypt` performs encryption on the strings and stream data, if any, of a node. The encryption key depends on the object ID, document file ID, passwords, and permission settings, and it varies each time the document is printed. That is, the key is not repeated.

Function `makeCode`, `startXML`, and `endXML` deal with PDF and XML output. When the document is being printed, the program traverses the tree to visit each node in depth-first order, calling their `makeCode` functions to generate PDF output, or calling `startXML` when first visiting a node and `endXML` when revisiting the node after finishing up with the children nodes. When a piece of XML code is converted into PDF, function `newFromXML` corresponding to a given tag (class) is called to convert the tag into an object.

The following function traverses the document object tree and simply lists the nodes:

```
PDFDoc::showTreeStruct( )
```

2.2 Creating PDF: the Old Way

For historical reasons, the following functions are still provided:

```
main::init( )
main::startPDF( ): PDFDoc
main::writePDF( )
main::importXML( $xml: String ): PDFDoc
main::exportXML( )
```

In versions 2.x, function `init` does some initialization such as guessing the execution environment, understanding command line arguments and form data posted via a web browser, manipulating default settings, and setting output file names. These operations are **removed** from version 3.0 as its behavior introduces more restrictions than convenience. Incompatibilities will result if your scripts built on PDFever 2.x depend on function `init` to read from `STDIN` and to manipulate default settings, or if they rely on global variables placed into the `main` namespace by PDFever.

Function `startPDF` builds certain document-level internal objects, mostly singletons. In the new version, the `PDFDoc` object takes these responsibilities. The other three functions are now modified and moved to `PDFDoc` package: `writePDF` and `exportXML` print the document in PDF and XML forms, respectively, and `importXML` builds the document from XML.

2.3 Create PDF: the New Way

An instance of `PDFDoc` is created by calling the constructor. Document properties can also be set, via a reference to a hash (HASH) containing the any collection of the keys listed in Table 2-2. For convenience, if you want to use default values for the properties, you can omit the argument or pass a page size name in stead. The prototype of the constructor is as follows:

```
PDFDoc::new( ): PDFDoc
PDFDoc::new( $pagesize: String ): PDFDoc
PDFDoc::new( $attr: HASH ): PDFDoc

<PDF Size="pagesize" <!-- More attributes --> >
    <!-- More tags to compose page contents etc. -->
</PDF>
```

Table 2-2. Available document properties set during instantiation.

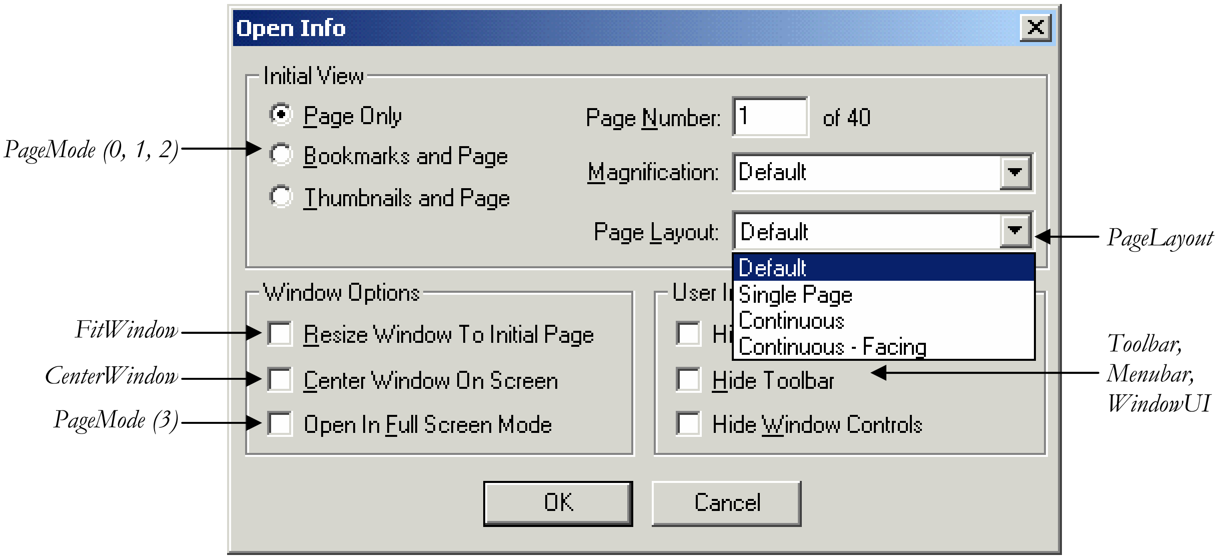
	Key	**Type**	**Explanation**	**Default**
Document info	`Author`	String	Document author	
	`Title`	String	Document title	
	`Subject`	String	Document subject	
	`Producer`	String	Producer application name	
Viewer preferences	`Toolbar`	Bool	If defined and set to 0, Acrobat will hide toolbar when opening this document.	
	`Menubar`	Bool	If defined and set to 0, Acrobat will hide menus.	
	`WindowUI`	Bool	If defined and set to 0, Acrobat will hide windows UI components.	
	`FitWindow`	Bool	If set, Acrobat will resize the window to fir the page.	
	`CenterWindow`	Bool	If set, Acrobat will center the window on screen.	
	`PageMode`	String / integer	Page open mode in viewer application. Must be one of `UseNone`, `UseOutlines`, `UseThumbs`, and `FullScreen`, or 0, 1, 2, 3, correspondingly.	
	`PageLayout`	String / integer	Page layout mode in viewer application. Must be `SinglePage`, `OneColumn`, `TwoColumnLeft`, `TwoColumnRight` or 0, 1, 2, 3, correspondingly.	
Page label	`PageLabel`	String	Expression defining page labels, see section 2.5	
Default page settings	`PageSize`	String	Page size. Must be `Letter`, `Legal`, `A4`, `A3`, `B5`, `B4`, `Env10`, `Postcard`, or `LetterR`, `LegalR`, `A4R`, `A3R`, `B5R`, or `B4R`.	`Letter`
	`Box`	`Rect`	Used to define the size of the pages.	
	`Margins`	String / ARRAY	A list of four size expressions as a string, or an array of four such elements. Used to define crop box.	
	`Paddings`	String / ARRAY	Similar to above, but is used to define art box.	
Color	`ColorSpace`	String	Color space, must be `RGB`, `CMYK`, or `Gray`.	`RGB`

Font and text	`FontFace`	String	Font face for text blocks, must be one of the predefined font names.	`Times-Roman`
	`FontSize`	Size	Font size for text blocks.	`12`
	`Leading`	Size	Line height for text blocks.	`12`
	`FontEncoding`	String	Font encoding for all fonts, if present, must be `WinAnsiEncoding`, `StandardEncoding`, or `MacRomanEncoding`.	`WinAnsiEncoding`
	`TextJustify`	String	Text justification for text blocks. Must be `Left`, `Center`, `Right`, or `Uniform`	`Left`
	`VerticalAlign`	String	Vertical alignment for text blocks. Must be `Top`, `Middle`, or `Bottom`	`Top`
Security	`OwnerPwd`	String	Owner password to change permissions.	
	`UserPwd`	String	User password to read the document.	
	`Print`	Bool	If set, user can print the document.	
	`Change`	Bool	If set, user can change the content using Acrobat.	
	`ChangeAll`	Bool	If set, user can change annotations and fill forms.	
	`Select`	Bool	If set, user can select and copy text and graphics.	
	`StrongEnc`	Bool	If set, program will use strong encryption (128-bit)	
	`LockForm`	Bool	If set, form fills are not fillable in any case	
	`NoAccess`	Bool	If set, accessibility features are disabled	
	`NoAssembly`	Bool	Set to prevent inserting or deleting pages, and creating bookmarks and thumbnails.	
	`PrintAsImage`	Bool	If set, file can only be printed at low resolution.	
Text preprocessing	`FindBIU`	Bool	Replace words enclosed by *, -, _ to bold, italic, and underline format.	
	`FindHyperlink`	Bool	Find URI when reformatting text.	
	`ReplaceEntities`	Bool	Replace special character entities into characters.	
Stream data	`UseASCII85`	Bool	Use ASCII85 encoding if set to 1.	
	`UseCompress`	Bool	Use Zlib/Deflate compression if set to 1.	
	`UseMetaData`	Bool	Create XMP metadata according to Acrobat 5.0	
Template	`NoMultiRefer`	Bool	If set, template objects will be duplicated for a new page, otherwise they are merely referred to by the new page.	

The constructor also creates one instance of each of the following: `DocInfo`, `Catalog`, `Pages`, and `Outlines`. If either or both passwords are defined (even with empty string values), it also creates an instance of `PDFEnrypt`.

- **Document information**

These properties are used to set the document information. A `DocInfo` object is created to store the information, while a `MetaData` object is also created to do the similar if the property `UseMetaData` is defined with a non-zero value. Adobe XMP framework specification defines XML formats for wrapping the information.



Open Info

Initial View

● Page Only

○ Bookmarks and Page

○ Thumbnails and Page

Page Number: 1 of 40

Magnification: Default

Page Layout: Default

Window Options

□ Resize Window To Initial Page

□ Center Window On Screen

□ Open In Full Screen Mode

□ Hi

□ Hide Toolbar

□ Hide Window Controls

Figure 2-2. Document open settings in Acrobat 4.

The `DocInfo` object can be accessed via the function call `getDocInfo`:

```
PDFDoc::getDocInfo( ): DocInfo
```

- **Viewer preferences**

These settings affect how Acrobat Reader arranges user interface when the document is opened, which can be changed with Acrobat, as shown in Figure 2-2. The corresponding properties are marked in this screenshot. Later we will describe how to make the document open at an arbitrary page other than the first and how to label pages.

The viewer preferences can be set by assigning corresponding attributes to 1 in Table 2-2, or use the following function anytime before the PDF file is created (using the same set of keys for the referenced hash):

```
PDFDoc::setViewerPref( $attr: HASH )
```

- **Default page settings**

You can specify a default page size for the document (each page can overwrite this setting with a different size and even rotation angle). If a custom size is needed, please define a rectangle extending from (0, 0) to desired width and height.

PDF specification defines five separate page boundaries: **media box** as the physical boundary of the paper, **crop box** as the region to which the page should be clipped when the page is viewed or printed (which has no intended use at all), **bleed box** as region to which the page content is clipped in production, **trim box** as the actual print-out after trimming, and **art box** as region that defines the extent of the page's meaningful content.

The default regions of the art box, crop box, and media box are the same. PDF Everywhere defines the crop box as the region to which a page is to be clipped when placed into another page as an XObject, and art box as the region within which all artwork should be confined although not enforced at all. Crop box is defined by `Margins`, and art box by `Paddings`. Both are measured from the physical boundary of the page medium. This indicates the crop box may well be defined inside the art box, as illustrated in Figure 2-3.

In Figure 2-3, a crop box is defined to contain only part of the page. The entire page is actually displayed and printed when viewing in Acrobat Reader. However, when the page is placed into another page by instantiating an XObject modeled after itself, only the region defined in the crop box is visible, although the origin of the XObject is still that of the page's media box. This behavior is not affected by transitions applied to the object, such as resizing and rotation.

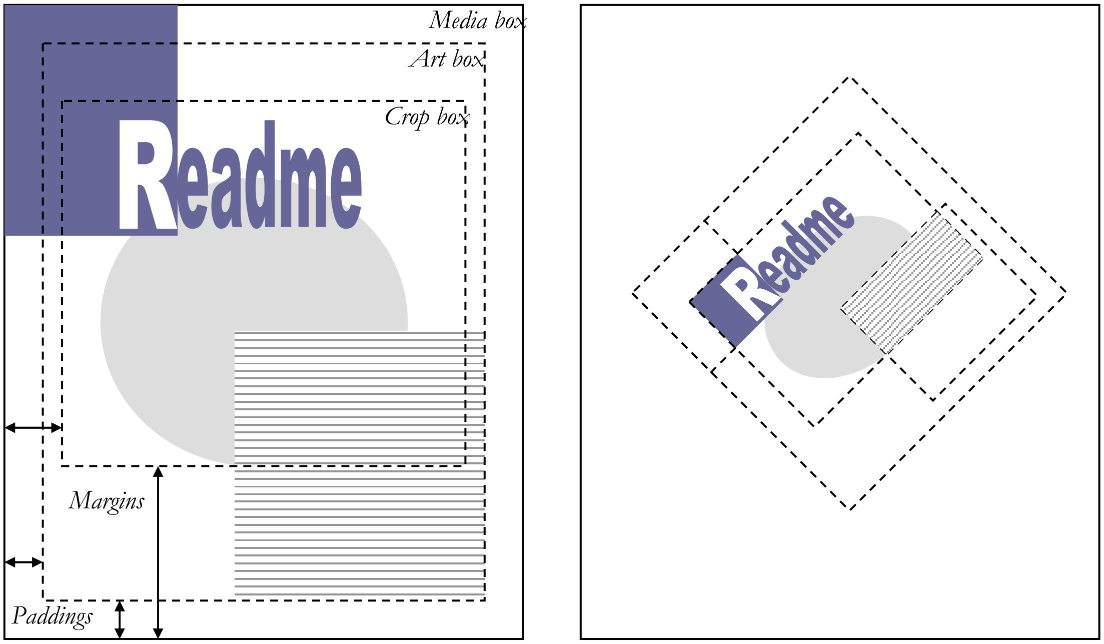


Figure 2-3. Crop box, art box, and the effect on XObject.

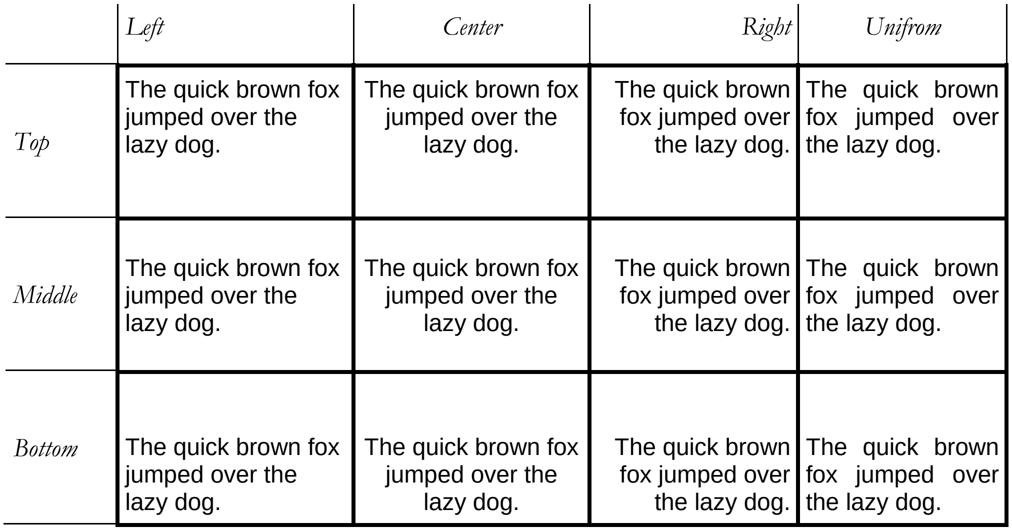
Margins and paddings are specified by either a string containing a list of *up to* four size values separated by commas, or an array of *up to* four such values, in the order of **left, bottom, right, top** (positive values only). If a value is omitted, symmetric margins or paddings are created. For example,

```
new PDFDoc ( { Paddings => '45, 50, 45, 50' } );
new PDFDoc( { Paddings => '45, 50' } ); # Same as above
new PDFDoc( { Margins => [ '1.25in' ] } );
```

- **Colors and fonts**

Colors and fonts will be discussed later in detail. The settings of `ColorSpace` and `FontEncoding` affects the appearance of the document, therefore should not be changed afterwards, or color shifting and out-of-place characters are likely to occur. Please see chapter 4 for more information about color and chapter 5 for details of using fonts.

Font size and leading (line height) are size values with optional unit as postfix. They are used for text boxes (see chapter 5 for detail). Horizontal and vertical alignments (`TextJustify` and `VerticalAlign`) are specified for the actual text contained inside a text box, with respect to the boundaries of the box. Figure 2-4 illustrates the various combinations of alignment schemes.



	Left	*Center*	*Right*	*Unifrom*
Top	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.
Middle	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.
Bottom	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.	The quick brown fox jumped over the lazy dog.

Figure 2-4. Illustrations of various text alignment schemes.

- **Text preprocessing**

Text fed to text boxes can contain inline markers (to be described later). A function `reformatText` defined in `TextContent` as a class method will do some processing on these markers. In addition, it converts certain text into bold, italic, or underline style by discovering the patterns defined in Table 2-2, and converts email addresses and Internet hyperlinks into active links (and the text to blue color), it `FindBIU` and `FindHyperlink` are set to non-zero values. For example, *abc*, -abc-, and _abc_ each becomes **abc**, *abc*, and abc.

In version 2.x, there is another option `ReplaceLine` that replace a line consisting of only consecutive dashes with a real line draw on the page. This feature has been discarded.

- **Stream data encoding**

PDF Everywhere can compress and/or ASCII85-encode all stream data, but it makes no sense to use ASCII85 encoding when encryption is used (all data become 8-bit). Compression is highly recommended.

If `UseMetaData` is set, the program would insert XML code stream into the PDF output, as described below.

2.4 Document Information (DocInfo) and MetaData

Document information can actually be set at any time, by the `DocInfo` object as returned by the `PDFDoc` object's `getDocInfo` method, or via the `PDFDoc` object.

```
PDFDoc::setTitle( $title: String )
PDFDoc::setSubject( $subject: String )
PDFDoc::setAuthor( $author: String )
PDFDoc::setProducer( $author: String )
PDFDoc::setKeywords( @keywords: String[] )
DocInfo::setFileID( )
```

The method `setFileID` is called internally to create file ID. PDF specification defines a permanent ID fixed at creation and a variable one that changes each time the file is modified. PDF Everywhere also uses the same ID for both, which is calculated by applying MD5 algorithm on author, subject, title, producer, current Greenwich mean time, and a random number.

Class `MetaData` is used for creating Adobe XMP-compliant XML packet[2]. A single instance is automatically created for the document. A triplet in different namespaces is generated, as listed in Table 2-3. Note that `dc:creator` is the same as `xap:Author`, and `dc:subject` is actually `xap:keywords`, both being a `rdf:Bag`. Also, `xap:Title` is a `rdf:Alt`. The entire packet, as a stream of bytes, is contained in `<?xpacket?>` tags, which has a fixed ID, and the length of the stream data is stored there as attribute.

The constructor of `MetaData` takes an optional argument, which is a hash reference. This hash can present the following keys: `PDF`, `XMP`, and `DC`, all of which have Boolean values. If a key is absent or is associated with a non-zero value, the corresponding XML tag is generated. If exists, and the value is set to 0, the tag is skipped. See the next section about `Catalog` for more information.

Table 2-3. Information stored in XML metadata output.

	PDF	XMP	DC
Author	`pdf:Author`	`xap:Author`	`dc:creator`
Title	`pdf:Title`	`xap:Title`	`dc:title`
Subject	`pdf:Subject`		
Keywords	`pdf:Keywords`	`xap:Keywords`	`dc:subject`
Producer	`pdf:Producer`		

[2] eXtensible Metadata Platform (XMP). http://www.adobe.com/products/xmp/

Creator	`pdf:Creator`	
Creation date	`pdf:CreationDate`	`xap:CreateDate`
Version	`pdf:PDFVersion`	
Metadata date		`xap:MetadataDate`

Current implementation of this feature is incomplete as doesn't use Unicode.

2.5 Catalog

Like the `DocInfo`, the instance of `Catalog` can be accessed by calling `getCatalog`:

```
PDFDoc::getCatalog( ): Catalog
```

These functions of class `Catalog` can be called (internal methods for managing the children nodes are not described here):

```
Catalog::labelPages( $labels: String )
Catalog::makeMetaData( $attr: HASH )
Catalog::setPageMode( $mode: String / integer )
Catalog::setPageLayout( $layout: String / integer )
Catalog::setViewerPref( $attr: HASH )
Catalog::setOpenPage( $page: Integer / Page / String )
```

Functions `setPageMode` and `setPageLayout` each takes an argument the same as the `PageMode` and `PageLayout` attrtibute defined in Table 2-2. Function `setViewerPref` set viewer preferences, and it takes a hash reference as the argument, which can contain keys `Toolbar`, `Menubar`, `WindowUI`, `FitWindow`, and `CenterWindow`, as defined in Table 2-2. In fact, the `PDFDoc` object just pass the attributes to these functions.

Function `makeMetaData` is called to create a `MetaData` object to generate an XMP packet. Hash reference `$attr` can contain three keys: `PDF`, `XMP`, and `DC`. You may want to call this function if the `UseMetaData` attribute in Table 2-2 is not defined when creating the `PDFDoc`, as otherwise it would have been done automatically. The output is not static; the `MetaData` object builds the stream data on the fly each time the `PDFDoc` is asked to print out PDF code.

- **Page label**

The method `labelPages` can be called to label all the pages in a document. For example, pass an expression "`1-2, 5-8`" would make the pages be labeled as 1, 2, 5, 6, 7, 8. The same expression can also be passed to the constructor of `PDFDoc`, assigned to the attribute `PageLabel`, as listed in Table 2-2. The labels are visible when you drag the scroll bar, and they are shown in thumbnail region in Acrobat.

Roman numbers and characters are recognized, too. For example, "`i-iv, 1-`" results in page labels i, ii, iii, iv, 1, 2, 3, ... Use comma to separate ranges, and dash to separate the lower and upper limits. The right-most characters in each pair are considered ordering numbers, and the prefix is preserved. For example, "`Page 6-7, Appendix A-D`" results in labels Page 6, Page 7, Appendix A, Appendix B, Appendix C, Appendix D.

Comma and dash can also be part of the prefix: just insert a backslash to the left of these signs, for example, “`Preface\-Revised 1-2, iv-`” would give you a set of page labels like “Preface-Revised 1, Preface-Revised 2, iv, v, vi, vii, viii, ix, ...”.

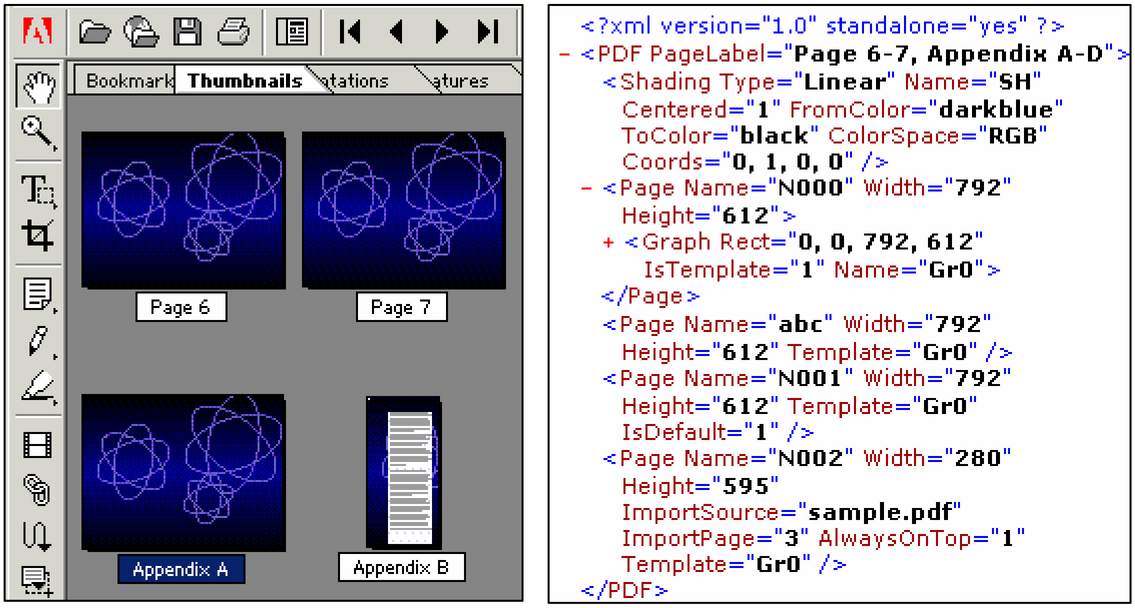
- ## Open page

The first (physical) page doesn’t have to be first one to be displayed when the document is opened in Acrobat. You can call function `setOpenPage` with one argument:

- If a string, it is treated as a name of a page object;
- If an integer, it is considered the page sequence number (starting from 0);
- If a `Page` object, it is used directly.

Example 2-1. Setting open page and page labels.

```
use PDF;
# Create page labels
$doc = new PDFDoc( { PageLabel => 'Page 6-7, Appendix A-D' } );
# or do the following:
$doc->getCatalog( )->labelPages( 'Page 6-7, Appendix A-D' );
# Build a page from template (it remains in effect for new pages)
$p1 = $doc->newPage( { Template => 'atom.tpl' } );
$p2 = $doc->newPage( { Name => 'abc' } ); # Assign a name
$p3 = new Page( { IsDefault => 1 } );  # Set as open page
# Import a page from a PDF file
$p4 = new Page( { ImportSource => new PDFFile( 'sample.pdf' ),
ImportPage => 3, AlwaysOnTop => 1 } );
$doc->getCatalog( )->setOpenPage( 'abc' ); # Set open page to page 2
$doc->getCatalog( )->setOpenPage( 2 );     # Set open page to page 3
$doc->getCatalog( )->setOpenPage( $p3 );   # Same as above
$doc->writePDF( 'abc.pdf' );
```



```
<?xml version="1.0" standalone="yes" ?>
- <PDF PageLabel="Page 6-7, Appendix A-D">
    <Shading Type="Linear" Name="SH"
      Centered="1" FromColor="darkblue"
      ToColor="black" ColorSpace="RGB"
      Coords="0, 1, 0, 0" />
  - <Page Name="N000" Width="792"
      Height="612">
    + <Graph Rect="0, 0, 792, 612"
        IsTemplate="1" Name="Gr0">
    </Page>
    <Page Name="abc" Width="792"
      Height="612" Template="Gr0" />
    <Page Name="N001" Width="792"
      Height="612" Template="Gr0"
      IsDefault="1" />
    <Page Name="N002" Width="280"
      Height="595"
      ImportSource="sample.pdf"
      ImportPage="3" AlwaysOnTop="1"
      Template="Gr0" />
  </PDF>
```

Figure 2-5. PDF and XML output of Example 2-1.

Equivalently, you can set a page as the default open page when it is created, by setting its `IsDefault` property. See later chapters for detail. Example 2-1 shows how to label pages and set open page:

The PDF output (and XML, after replacing `writePDF` with `exportXML`) is shown in Figure 2-5. As we can see, the pages are correctly labeled. For the last page, which contains contents from both the template and the imported page, the dimension is changed to that of the latter, while the template objects are placed below the imported ones because of the `AlwaysOnTop` setting.

In XML output, the page label expression is placed in `<PDF>` tag, which is handed to `PDFDoc` when the XML is converted to PDF. Certain objects, namely the `Shading` tag and the `Graph` tag (which is folded), are created by converting the original template objects. When converted to PDF, the “`IsTemplate`” property of the `Graph` tag will add it into the template object collection. For more information on template use, please see later chapters.

2.6 Output

PDF Everywhere uses temporary files created in the directory specified through environment variable `TMPDIR`, `TEMP`, `TMP`, or from where the executable is started up (by parsing `$^X`), trying in that order. When printing PDF output, it first creates such a temporary file, named after the pattern “`PEV`*`hhhhhhhh`*.TMP”, where *`h`* being a character in the class [0-9A-F], then redirects `STDOUT` to this file. It needs to do so because it must build the table of byte offsets for all objects and record the offset of that table in trailer. Next it traverses the nodes to print code, calling the function `makeCode` on each node, followed by `cleanUp` when revisiting the same node after completing the depth-first traversal to remove temporary changes made by certain nodes.

The function to write PDF output is `writePDF`, which takes an output file name, a hash reference to set attributes, and a code reference as call-back function (all optional):

```
PDFDoc::writePDF( $fname: String, $attr: HASH, $callback: CODE )
```

Keys acceptable in the hash include `OwnerPwd`, `UserPwd`, `Print`, `Change`, `Select`, `ChangeAll`, `StrongEnc`, `LockForm`, `NoAccess`, `NoAssembly`, `PrintAsImage`, and `Final`. All except the last one are the same as those listed in Table 2-2, and are used for setting security at output time. The last one, a Boolean value, is used for telling the program (by assigning it a non-zero value) that the document is not used after the output is done, since with this knowledge the program would do skip some steps to speed up the process (and you don’t call `writePDF` on the same `PDFDoc` object again if it contains imported pages).

Note that the callback function must NOT write anything to `STDOUT`.

XML input/output is done by calling this function:

```
PDFDoc::importXML( $arg: String ): PDFDoc
PDFDoc::exportXML( )
```

Table 2-4. Information available for the callback function in writePDF.

Key	Type	Explanation
`ObjId`	Integer	ID of current object being printed
`ObjNum`	Integer	Total number of objects
`CurrLen`	Integer	Current offset before the object is printed

The XML output is always printed to `STDOUT`. It cannot print to a file. The XML code is indented; it starts with `<PDF>` tag and ends with `</PDF>` tag. The following rules are followed:

1. All properties listed in Table 2-2, if defined, are printed as attribute for the `<PDF>` tag.
2. All resource objects are printed at the top of the document; all annotations and bookmarks are printed at the end of the document.
3. If you've used XML templates to format pages, they are expanded into objects as part of current document.

PDF Everywhere tries to produce an XML output that can be converted to PDF, using the function `importXML` that takes one argument, which is either a file name or the XML code itself. If the argument is omitted, it tries the command line argument; if that is also not defined, it tries reading from `STDIN` (this behavior is for backward-compatibility.) It always assumes ISO-8859-1 encoding for the XML.

One last note: the XML output is strictly for authoring via PDF Everywhere's conversion mechanism only. It has nothing to do with the XML representation of a PDF file as created by Acrobat 5 and alike.

2.7 Stream Data

Page contents are drawn on page by a series of graphical operators similar to the PostScript language. PDF specification requires that these commands be stored as chunks of stream data, preferably encoded using one or more filters. Metadata, image data, embedded font program, etc., are all comprised of a dictionary describing properties and dependencies followed by a stream of bytes.

PDF Everywhere extensively uses `PDFStream` and its subclasses. As illustrated in Figure 1-1, there are many subclasses. The following methods are defined in class `PDFStream`:

```
PDFStream::customCode( )
PDFStream::setFilter( $filter: string, $decodeParm: string )
PDFStream::setDiskBased( )
```

Function `customCode` is always overwritten in subclasses to deal with specialization; it is used to build the custom dictionary and to make changes to stream data. Function `setFilter` merely record the fact that a filter has been applied. Function `setDiskBased` transfers the stream data into an internally maintained temporary file, which is deleted when program exits. By doing so the object would consume less memory, especially when the stream data is read from a large disk file, for example an image or font file.

When printing PDF output, stream data are ASCII85-encoded or compressed, depending on the settings `UseASCII85` and `UseCompress` passed to the constructor of `PDFDoc`.

2.8 Other Accessor Methods

The following accessor functions related to names are defined for `PDFDoc`:

```
PDFDoc::setName( $obj: PDFTreeNode, $name: string ): string
PDFDoc::getObjByName( $name: string ): PDFTreeNode
```

Function `setName` is automatically called whenever an object that is a tree node is created. It returns the actually used name after resolving possible name conflictions by incrementing the name literally. `PDFDoc` has a name lookup table that links names to objects, which is queried by `getObjByName` to find the object associated with a given name, or `undef` on failures.

Function all returns a list of all objects of the given class, optionally starting from a branch:

```
PDFDoc::all( $class: string, $branch: PDFTreeNode ): PDFTreeNode[]
```

2.9 Common Utility Functions

The following functions, used to be in the `main` namespace, are defined in package `PDF` (for compatibility with version 2.x, they are exported to the `main` namespace):

```
PDF::MD5( @input:string[] ): byte[16]
PDF::RC4( $key: string, $data: string ): string

PDF::tellSize( $size: string ): number

PDF::min( @numbers: number ): number
PDF::max( @numbers: number ): number

PDF::strToHex( $str: string ): string
PDF::hexToStr( $str: string ): string
```

Functions `MD5` and `RC4` are used for encryption. `MD5` accepts a list of strings and returns a 16-byte message digest in binary form.

Function `tellSize` converts a size value such as “5cm” to a float number.

Functions `min` and `max` finds the minimum or maximum value from a given list.

Functions `strToHex` and `hexToStr` converts between the binary form and hex-coded form of a string.

2.10 Security settings

PDF Everywhere 3 encrypts a document according to Acrobat 4.x method, which is a 40-bit encryption, or Acrobat 5.x method, which is a variable-length encryption but it is fixed to 128-bit here. This library uses MD5 message-digest algorithm as defined in Network Working Group RFC 1321[3], and Arc-four algorithm, thought to be compatible with RC4 used by Acrobat but not guaranteed as RC4 is a proprietary algorithm of RSA Security, Inc. and never enters public domain.

Both passwords are first padded or truncated to 32 bytes. An encrypted owner password value is generated from both the owner and the user passwords. An encrypted user password is generated from the encrypted owner password, the user password, the permission, and the file ID. These are one-way operations – the information cannot be extracted from the encrypted values.

The permission settings correspond to the fields shown in Figure 2-6 and 2-7 that are screenshots of Acrobat dialog boxes. In PDF Everywhere, if security option is enabled, permissions defined for Acrobat 4 security handler are *denied* by default, and you must grant the permissions explicitly, while those defined for Acrobat 5 security handler are *granted* by default, and you need to restrict them explicitly. For example, if you want to restrict printing at low-resolution only, you would need to set both `Print` and `PrintAsImage` to 1 while enabling strong encryption (128-bit).

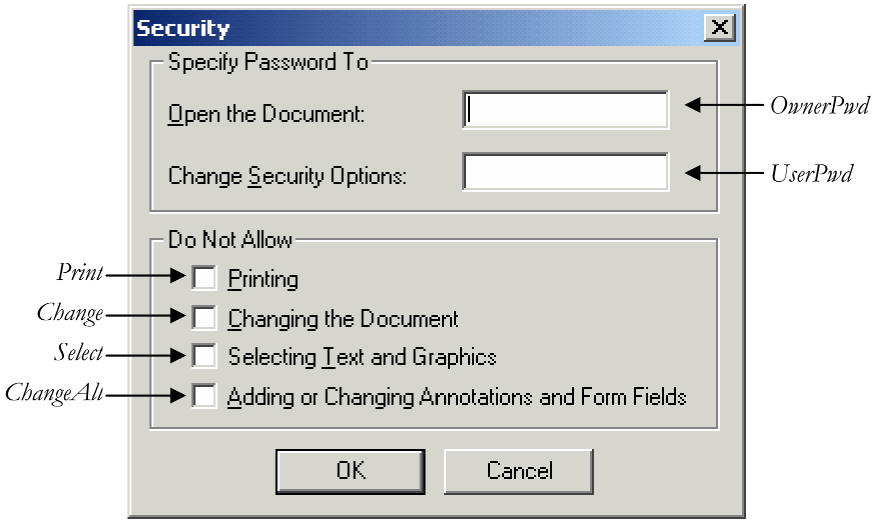


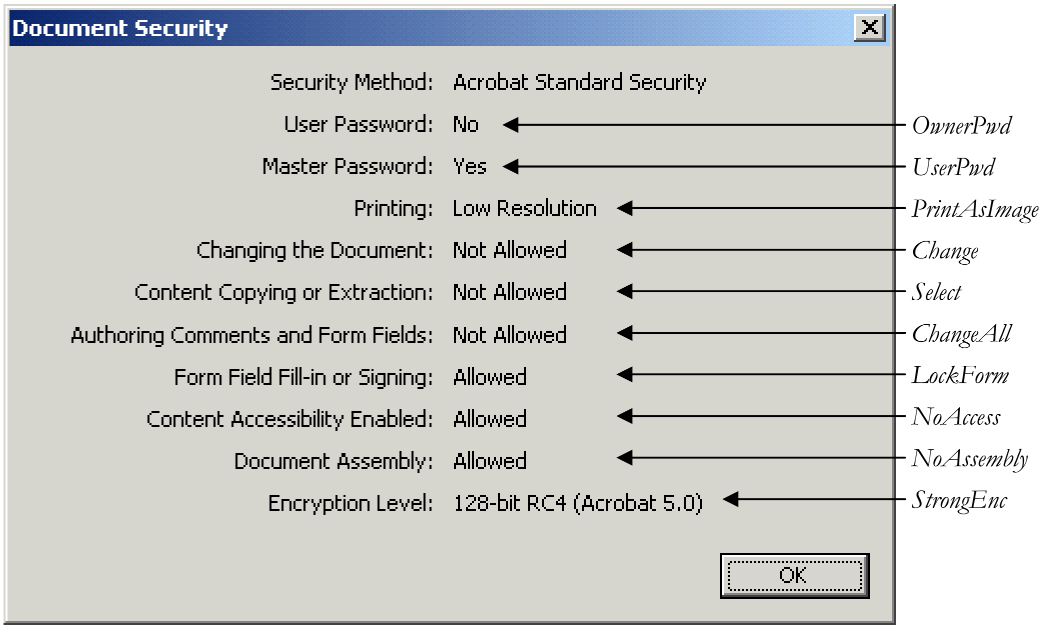
Figure 2-6. Document security settings in Acrobat 4.

As described above, security settings can be applied at the time of document creation (by instantiating a `PDFDoc` object) or PDF generation (by calling function `writePDF`). Security settings can also be applied or removed at any time, using the following two functions defined for `PDFDoc` objects:

[3] Ronald Rivest, MIT Laboratory for Computer Science and RSA Data Security, Inc., RFC 1321: The MD5 Message-Digest Algorithm. April 1992. http://www.phys-iasi.ro/library/rfcs/rfc1321.htm

```
PDFDoc::setSecurity( $attr: HASH ): PDFEncrypt
PDFDoc::removeSecurity( )
```

The argument `$attr` has the same keys for security settings as listed in Table 2-2. Since function `writePDF` can silently apply security settings, you can call `removeSecurity` after printing an encrypted version of a PDF file.



Security Method: Acrobat Standard Security
User Password: No
Master Password: Yes
Printing: Low Resolution
Changing the Document: Not Allowed
Content Copying or Extraction: Not Allowed
Authoring Comments and Form Fields: Not Allowed
Form Field Fill-in or Signing: Allowed
Content Accessibility Enabled: Allowed
Document Assembly: Allowed
Encryption Level: 128-bit RC4 (Acrobat 5.0)

Figure 2-7. Document security settings with the Acrobat 5 method.

If the document contains imported pages, they are encrypted or decrypted as well. Since each time a document is created from a script a new file ID is assigned (even if no changes have been made), no two runs would generate identical PDF output. Note that import objects are encrypted just before output generation then decrypted back immediately after the code is printed. This consumes additional computation time, but is not avoidable because of the change in file ID.

Chapter 3: Pages and Templates

3.1 Pages Tree

In a PDF document, pages are typically organized into a tree structure, with `Page` objects as leaves, and a type of object called `Pages` as tree root and branches. There must be at lease one `Pages` object, that is, the tree root. Its children can be `Page` objects and `Pages` objects, which in turn have their own children. This root `Pages` object is automatically created by `PDFDoc`, and it is returned by the following function:

```
PDFDoc::getPagesRoot( ): Pages
```

Due to the highly dynamic nature of page creation and arrangement, PDF Everywhere uses this single `Pages` object to manage all pages; when a new page is created, it automatically adds itself to the children of the `Pages`.

However, when generating PDF output, the program does some trick: it creates temporary `Pages` objects if necessary, based on the following algorithm: divide all `Page` objects into groups of 10, then create a `Pages` object for each group; next, group the newly created `Pages` objects and ungrouped `Page` objects then create another layer of `Pages`; finally, when the number of a newly layer of `Pages`/`Page` objects is less than 10, they are added to the root `Pages` node. When the output is done, these new objects are *removed*. This approach builds a tree from bottom-up and doesn't make a balanced tree.

- **Retrieve a page**

All pages are stored in the order of creation and managed by the `Pages` root. To get the reference to a page, pass the index (integer, for the first page it is zero) to this function:

```
Pages::getPageByNumber( $index: integer ): Page
```

You may need the following functions to get/set current page object and the total number of pages:

```
PDFDoc::getCurrPage( ): Page
PDFDoc::setCurrPage( $pg: Page )
PDFDoc::getPageCount( ): int
```

By setting a page as "current" all content type objects are linked to that page until another page is created or set as "current page" (default page). The last function returns the total number of pages of the document.

- **Reorganize page order**

```
Pages::reorganizePages( @seq: integer[] )
```

Function **reorganizePages** is used to reorganize the pages. The sequence should contain the indices of all pages, such as "`2, 1, 0, 3`" for a 4-page document.

Sometimes you may want to swap two pages, for example, you build a table of contents after all pages are created then switch it to the first page. The library provides a function to do so; the order of the two pages are interchanged, and the document tree hierarchy is adjusted so that the XML output would be in correct order, too. The function can be called on a `Page` object or the `PDFDoc` object:

```
Page::swapPage( $that: Page )
PDFDoc::swapPage( $this: Page, $that: Page )
```

3.2 Page Object

PDF Everywhere recognizes the names listed in Table 2-2 for `PageSize`: `Letter` (8.5 × 11 in), `Legal` (8.5 × 14 in), `A4` (210 × 297 mm), `A3` (297 × 420 mm), `B5` (182 × 257 mm), `B4` (257 × 364 mm), `Env10` (4.13 × 9.5 in), `Postcard` (4 × 6 in), and landscape orientations of these pages `LetterR`, `LegalR`, `A4R`, etc., except for Env10 and Postcard.

If a page size is not specified for a document, the document defaults it to be `Letter`. Each page can have a different dimension, and custom sizes are supported. The constructor of `Page` objects can take various types of parameters (you can also call `PDFDoc::newPage( )`, which just forwards the call to this function):

```
Page::new( ): Page
Page::new( $size: String ): Page
Page::new( $box: Rect ): Page
Page::new( $attr: HASH ): Page

<Page Name="" <!-- More attributes --> >
    <!-- Content tags -->
</Page>
```

If no argument is given, the new page has a size defined by document default setting. If a string is passed, it is considered one of the predefined page size names (if not found, the parameter is discarded). If a rectangle is passed, the page will be set to the same size as the rectangle. However, the last form could be most useful of all. Like other function calls, a reference to a hash is needed. This hash may contain the following keys:

Table 3-1. Available page properties set during instantiation.

	Key	Type	Explanation	Default
Page media	`Box`	`Rect`	A `Rect` object that defines the page size.	
	`Size`	String	Page size. See the section on `Pages`.	
	`Width`	number	Page width	
	`Height`	number	Page height	
	`Rotation`	Integer	Page rotation. If given, must be 1 – 3.	`0`

Structure	`Template`	String	A list of names, separated by commas, of the objects to be used as template for this page, or an external template file name, or the template code.	
	`Name`	String	A name for the page.	
	`Parms`	HASH	A collection of parameters to be used in an XML template definition.	
	`IsDefault`	Bool	If 1, the page is automatically set as open page.	
	`Bookmark`	String	Used to add a bookmark for the current page; the viewing mode is to fit the entire page in window.	
	`XObject`	String	Forward referencing mechanism; a page can be incorporated into another page via an `XObject` as delegate. This property specifies the name to be assigned to the `XObject`.	
Transition	`Trans`	Integer	Page transition visual effect. If present, must be 1 – 7, as listed in Table 3-3.	
	`TransDur`	Integer	Duration (in seconds) of the transition effect.	
	`Dimension`	Char	Dimension of certain transition modes, must be '`H`' for horizontal or '`V`' for vertical.	`H`
	`Direction`	Integer	Direction of certain transition modes, in degree.	
	`Motion`	Char	Motion of certain transition modes, must be '`O`' for outwards or '`I`' for inwards	`O`
	`Duration`	Integer	Page display duration in seconds before auto-flipping.	`0`
Import	`ImportSource`	`PDFFile`	An existing PDF file.	
	`ImportPage`	Integer	Page number in the PDF file.	
	`AlwaysOnTop`	Bool	If 1, imported contents always remain on top of other page contents.	

The attributes `Box` and `Size` are identical to those seen above, which can nonetheless be passed to the constructor directly.

A page can be rotated by a 90-degree increment. `Rotation` property can be set to 1 (90° counterclockwise), 2 (90° clockwise), or 3 (180°). The page is displayed after the contents of the entire page are built; it doesn't affect the page coordinate system. A letter-size page still measures 8.5 × 11 inches even if it rotates by 90 degree.

3.3 Page Template

As described in section 1.3, content objects can be added into a document-level template collection; when a new page is created, these objects are duplicated or simply referenced by this page, depending on the value of the `NoMultiRef` setting for the `PDFDoc` document, as shown in Table 2-2.

The value for `Template` attribute in above table overwrites the current setting of templates. It can be used to do the following:

- If its value is a list of object names, these objects would become the new definition of document template collection;
- If a piece of XML code that defines a template, then new objects are created from XML and added to template collection, and they remain effective until another page reset the template collection;
- If a file name, then the template code stored in that file is retrieved, and what happens next is exactly the same thing above.

A template file is a bona fide XML document but they must have a file name extension “`tpl`” or “`xml`” to distinguish them from regular XML files used for PDF conversion. The major difference is that in a template file there can be only ***one*** page, since a template is used to format a single page anyway. There is no `PDF` or `Page` tag in a template file; rather, a pair of `<Template>` tags hosts the page content definition, as illustrated in Example 3-1.

A `<Template>` tag must specify the `PageSize` or use `Width` and `Height` attributes. Beside, it should have a unique `Name`:

```
<Template Name="string" Width="size" Height="size">
    <!-- Content tags -->
</Template>
```

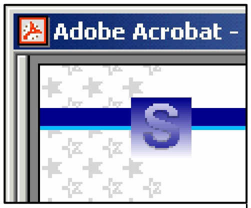
Why? Because this name acts as a “namespace” for the resource objects defined in the template. A `PDFDoc` object keeps track of all templates used in the document, thus avoiding duplicates of resource objects when a same template is explicitly applied more than once (through the steps shown above). If the template doesn’t have a name, the names of resource objects are placed into a default “namespace”, risking name confliction between objects defined in different templates. If such confliction occur, the PDF output wouldn’t be correct.

Then how about content objects? If a template is applied to different pages explicitly, the content objects always get duplicated. The reason is as follows. Once a template is parsed, the content objects are created and the internal template collection immediately gets updated. Therefore, if the next page doesn’t use a template (either by using a template file or by specifying the content objects it wants), all objects in current template collection are referenced in this new page. On the other hand, if the page *does* use a template file but it is the same one as used before, the original template collection has already been reset to empty, and the program always create new instances of these objects.

PDF Everywhere cannot directly generate XML code to be used as template. You must make some changes to the XML output: select an appropriate `<Page>` tag, retain the content tags with it, and then copy the resources tags such as images.

In Example 3-1, we first create a texture named “`Star`”, and then in a graphics layer called “`Gr0`” we use the texture to fill a rectangular region. Two shadings are created and are used for rendering the monograph.

Example 3-1. A sample template XML document.



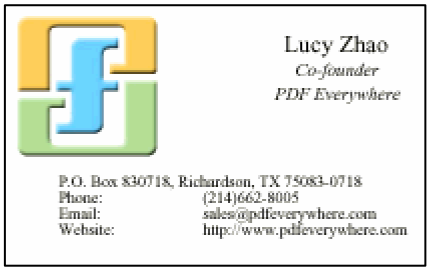
```
<Template PageSize="LetterR">
   <!-- Resources tags -->
   <Texture Name="Star" Width="40" Height="40"
      ColorSpace="RGB" Uncolored="1">
      <MoveTo X="10" Y="30"/>
      <DrawPolygon Sides="5" Radius="10"
         Rotation="18" Tilt="72" Fill="1"/>
      <MoveTo X="30" Y="10"/>
      <DrawPolygon Sides="5" Radius="10"
         Rotation="18" Tilt="72"/>
   </Texture>
   <Shading Name="Grad1" FromColor="Darkblue" ToColor="White"
      Dir="T->B"/>
   <Shading Name="Grad2" FromColor="Darkblue" ToColor="White"
      Dir="B->T"/>
   <!-- Contents tags -->
   <Graph Rect="0, 0, 120, 612" Name="Gr0">
      <Set Texture="Star" Color="Lightgrey" Ground="1" />
      <DrawRect Width="120" Height="612" Fill="1" />
      <MoveTo X="0" Y="550"/>
      <Set Color="DarkBlue" Ground="1"/>
      <DrawRect Width="792" Height="20" Fill="1" />
      <Set Color="DeepSkyBlue" LineWidth="4"/>
      <MoveTo X="0" Y="550"/>
      <LineTo X="792" Y="550"/>
      <Exec Cmd="NewPath"/>
      <Fill Rect="90, 520, 150, 580" Shading="Grad1"/>
      <ShowText Text="S" FontFace="Helvetica-Bold" FontSize="60"
         X="100" Y="590" Shading="Grad2" BorderColor="Black"/>
   </Graph>
</Template>
```

Template can actually contain parameters, to be replaced with custom values when the template is used for creating a new page. To work with the parameters, you need to do the following:

1. Insert a section `<Parms>` as the first thing after the `<Template>`;
2. Inside the `<Parms>` section, list each parameter in a `<Parm>` tag, which has a parameter `Name` and default `Value`;
3. In your program, load the template to preformat a new `Page` or an `XObject`, while passing the parameters in an hash (you don't have to provide new values for all parameters);
4. To do the same as step 3 in an XML document, you need to add these two attributes to the tag `<Page>`: Template with a value as the template file name, and Parms with a value of a string, listing the param/value pairs in the format of "`parm1:value1; parm2: value2; ...`" If '`:`', '`;`', and '`"`' are present in the values, you can use the hex code of these characters, prefixed by a "%" sign similar to HTML form data, for example in "`Name:Daisy%89Anna;Email:da`".

Example 3-2 actually creates a whole sheet a business card, by instantiating an XObject (as we mentioned before, it acts like a page – it actually derives from the `Page` class – but is painted like an image) with a template then showing it for multiple times. Note that only some parameters are supplied; other template parameters take their default values. The text box contains inline markers, to be introduced later.

Example 3-2. A parameterized template document and its use in XObject.



Lucy Zhao
Co-founder
PDF Everywhere

P.O. Box 830718, Richardson, TX 75083-0718
Phone: (214)662-8005
Email: sales@pdfeverywhere.com
Website: http://www.pdfeverywhere.com

```
<!-- Listing 1: mycard.tpl -->
<Template Name = "Card" Width="252"
   Height="144">
<Parms>
   <Parm Name="Name" Value="Joey Li"/>
   <Parm Name="Title" Value="Co-
founder"/>
   <Parm Name="Prog" Value="PDF
Everywhere"/>
   <Parm Name="Phone" Value="(214)662-
8005"/>
   <Parm Name="Email" Value="info"/>
</Parms>
<Image Width="500" Height="500" File="pdflogonew.jpg" Name="Im0"/>
<Graph Rect="0, 0, 252, 144" Name="Gr0">
   <ShowImage Name="Im0" Left="15" Bottom="50" Scale="0.16" />
</Graph>
<Text Name="info" Rect="40, 15, 200, 60" FontSize="8"
   VerticalAlign="Bottom"><![CDATA[P.O. Box 830718, Richardson, TX
   75083-0718 ^n Phone: ^t %Phone% ^n Email: ^t
   %Email%@pdfeverywhere.com ^n Website: ^t
   http://www.pdfeverywhere.com]]>
</Text>
<Text Name="name" Rect="120, 80, 240, 120" TextJustify="Center"
   VerticalAlign="Middle"><![CDATA[%Name% ^n ^f:size=9 ^i %Title% ^n
   %Prog%]]>
</Text>
</Template>


# Listing 2: mycard.pl
use PDF;

$doc = new PDFDoc( { UseCompress => 1} );
$card = $doc->newXObject( { Template => 'mycard.tpl', Params => {
   Name => 'Lucy Zhao', Email => 'sales' } } );
$doc->newPage( );
$layout = $doc->newGraph( );
$layout->moveTo( '0.7in', '0.3in' );
@cells = $layout->drawGrid( '7in', '10in', 0, { 'XGrids'=>2,
   'YGrids'=>5, 'NoDraw'=>1 } );
for $cell ( @cells ){
   $layout->showXObject( $card, $cell->left( ), $cell->bottom( ),
   {'Width'=>$cell->width( ), 'Height'=>$cell->height( )} );
}
$doc->writePDF( 'mybuscard.pdf' );
```

3.4 Internal Template

As introduced before, the internal template collection is a list of objects that are to show up on subsequent new pages. This collection contains only content-type objects, including `TextContent`, `GraphContent`, `FloatingText`, and `PreformText`. When creating objects of these classes, you can put it into the template collection by setting the attribute `IsTemplate` to true, or at any time later by calling function `addToTemplate`:

```
PDFDoc::addToTemplate( @objs: PDFTreeNode[] )
PDFDoc::dropFromTemplate( @objs: PDFTreeNode[] )
PDFDoc::resetTemplate( )
```

For example, when creating a document of 8 pages, where pages 4 through 7 bears a same watermark, we can create a `GraphContent` object when manipulating page 4 and call `addToTemplate` to add the object. When creating pages 5, 6, and 7, the system will automatically add this object to the active page. Before creating page 8, we call `dropFromTemplate` or `resetTemplate` to prevent it from showing up on the last page.

In XML, this is done by an attribute `IsTemplate="1"` for the tags, such as:

```
<Graph Name="footer" IsTemplate="1"><!-- More tags here --></Graph>
```

However, the XML exported by PDFever doesn't contain this attribute. The reason is that the other two template-manipulation functions have no corresponding tags. The solution is, a `<Page>` tag may contain an attribute `Template` listing the *names* of the objects shown on that page, such as:

```
<Page Name="form1" Template="footer, logo"/>
```

In summary, here's the approach of defining and reusing a template object:

1. During the construction of any page, assign `IsTemplate` attribute to a content object:

   ```
   $g1 = new GraphContent( {'Name'=>'BgDesign', 'IsTempalte'=>1} );
   ```

2. Or manipulate template manually (Subsequent new page objects will add the current template objects to their content):

   ```
   $g2 = new GraphContent( {'Name'=>'newdesign'} );
   $t1 = new FloatingText( new Rect( 100, 200, 500, 400 ), "Title",
         {'Name'=>'title'} );
   PDF::dropFromTemplate( $g1 );
   PDF::addToTemplate( $g2, $t1 );
   ```

3. Or, you can select just those objects you want:

   ```
   new Page( {'Name'=>'P3', 'Template'=>'title, newdesign'} );
   ```

4. Alternatively, you can load a template from a disk file:

   ```
   new Page( {'Name'=>'P4', 'Template'=>'tpl/star.tpl'} );
   ```

5. With XML, you can do the same as:

```
   <Page>
   <Graph Name="newdesign" IsTemplate="1"><!-- more tags --></Graph>
   <Text Name="title" IsTemplate="1" Rect="100, 200, 500,
400">Title</Text>
   </Page>
   <Page Name="P3" Template="title" />
   <Page Name="P4" Template="tpl/star.tpl" />
```

3.5 Page Snapshot as an XObject

A page can be incorporated into another page as if it were a regular image, via the XObject mechanism. If fact, this is the mechanism how form buttons are drown. It makes it easy to place and transform a template on a page, even with different sizes.

An XObject is a special "page": it can be initiated from a template, and it can take the parameters `Box`, `Size`, `Name`, `Template`, `ImportSource`, `ImportPage`, and `AlwaysOnTop` as listed in Table 3-1. Another way to build it is to use a "copy constructor"; the snapshot is created with *current* appearance of the page. Example 3-3 shows both schemes.

Example 3-3. Two ways to place an XObject.

```
use PDF;
my $doc = new PDFDoc( );
$card = new XObject( { Template => 'brm.tpl' } );
# Page 1 shows an XObject build from template
$p = $doc->newPage( new Rect( 0, 0, '8.5in', '8.5in' ) );
$g = $doc->newGraph( );
$g->setColor( 'Orange', 1 );
$g->moveTo( 10, 10 );
$g->drawRectTo( 300, 420, 1 );
$g->setColor( 'DarkBlue', 1 );
$g->moveTo( 10, 420 );
$g->drawRectTo( 600, 600, 1 );
$g->setColor( 'Black', 1 );      # Affects text color, too.
$g->showXObject( 300, 10, $card, { 'Height'=> 300, 'Flip'=>1,
   'Href'=>'http://www.pdfeverywhere.com' } );
$g->showText( 30, 560, 'PDF Everywhere', {'FontSize'=>60,
   'FontFace'=>'Times-Bold', 'Color'=>'Orange'} );
$g->showImage( new ImageContent( 'acrobatbg.gif' ), 30, 30 );
# Page 2 shows a snapshot of page 1
$doc->newPage( );
$g = $doc->newGraph( );
$g->showXObject( 100, 100, new XObject( $p ), {'Scale'=>0.6} );
$doc->writePDF( );
```

To display an XObject, you need to call the function `showXObject` on a `GraphContent` object, which takes as arguments a pair of coordinates, the XObject, and a hash reference for attributes.

```
XObject::new( ): XObject
XObject::new( $size: String ): XObject
XObject::new( $box: Rect ): XObject
XObject::new( $attr: HASH ): XObject

< XObject Name="string" <!-- More attributes --> >
    <!-- Content tags -->
</ XObject>
```

The above functions and XML tag are just similar to those described for `Page` objects.

The third way to create an XObject is to pass a name to the `XObject` attribute when creating a new page (this property would be exported in XML). There is a subtle difference between this approach and creating an `XObject` from an existing `Page`: the appearance of the XObject will not be finalized until the PDF code is being generated. This indicates that two pages cannot mutually incorporate each other. To access this XObject, use the function `getObjByName` defined in class `PDFDoc`.

```
XObject::new( $page: Page ): XObject
```

```
< XObject Name="string" Page="pagename"/>
```

XObjects built from templates are turned into a `XObject` tag in XML output, cutting the link with original template file. The instance built from another page becomes a mere attribute of that page because it is completely dependent on the later. Example 3-4 shows the difference. What's also indicated by this example is that an `XObject` tag can host graphics and text content tags. You can make use of this fact if you want to build an invisible page without resorting to a separate template file.

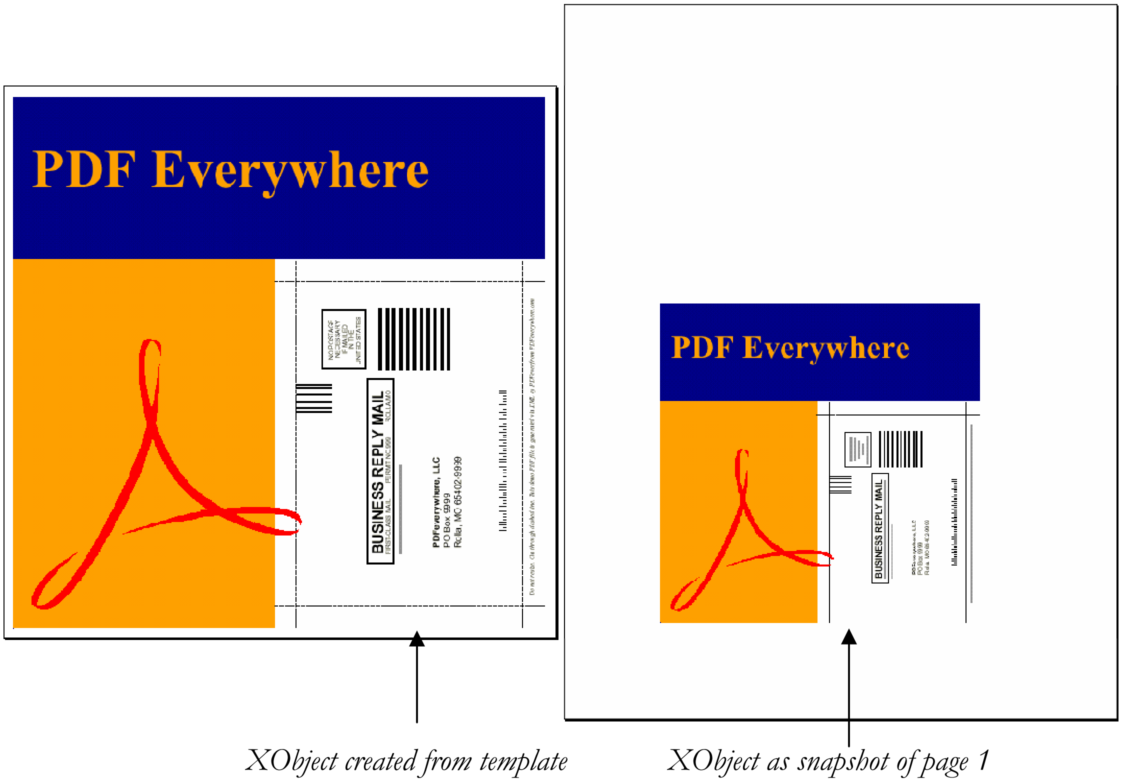


Figure 3-1. Using a page as an XObject (output of Example 3-3).

XObject can also be used to import a page in an existing PDF file, just like a Page does. This means we can write a very short program to create an *n*-up imposition of the file, as shown in Example 3-5 that places 4 pages onto one sheet of paper.

Example 3-5 makes use of the table-based layout model. It places up to 4 pages as XObjects, aligned to the four corners of the art box (defined by the paddings) using the Anchor parameter for the function `showXObject` (and the coordinates are omitted as the positioning is relative). These XObjects are scaled (in proportion to the original dimension) to fit ¼ of the art box, i.e. the default cell in a 1 × 1 table.

Example 3-4. Sample XML code equavalent to script in Example 3-3.

```
<?xml version="1.0" standalone="yes"?>
<PDF PageMode="UseNone" PageLayout="SinglePage" PageSize="Letter">
   <Image Width="300" Height="301" File="acrobatbg.gif" Name="Im0"/>
   <XObject Name="Pg0" Width="481.882" Height="354.325">
      <Graph Rect="0, 0, 612, 792">
         <Set LineWidth="2.5" />
         <MoveTo X="3cm" Y="7cm" />
         <!-- Code Omitted -->
      </Graph>
      <!-- Code Omitted -->
   </XObject>
   <Page Name="Pg1" Width="612" Height="612" XObject="Pg3">
      <Graph Rect="0, 0, 612, 612" Name="Gr0">
      <!-- Code Omitted -->
      </Graph>
   </Page>
   <Page Name="Pg2" Width="612" Height="792">
      <Graph Rect="0, 0, 612, 792" Name="Gr1">
         <ShowXObject Name="Pg3" Left="100" Bottom="100" Scale="0.6"
         />
      </Graph>
   </Page>
</PDF>
```

Example 3-5. Simple 4-up imposition of a PDF file.

```
use PDF;
$doc = new PDFDoc( );
$pdf = new PDFFile( 'review.pdf' );
$count = $pdf->getPageCount( );  # Get total number of pages
@anchors = qw(NorthWest NorthEast SouthWest SouthEast);
$i = 0;
while( $count ){
   my $pg = $doc->newPage( { Size => 'Letter', Padding => '20' } );
   my $gr = $doc->newGraph( );
   for( 0..3 ){
      $gr->showXObject( 0, 0, new XObject( {
         ImportSource => $pdf, ImportPage => $i++ }
         ), { Anchor => $anchors[$_], Height => '5.2in' } );
      last unless --$count;
   }
}
$doc->writePDF( 'impose.pdf' );
```

The approach depends on the ability of the program to decode original stream data stored in that PDF file and the correctness of the original content in terms of PDF graphical operators that draw the content. In PDF, each page is independent of another, in that each maintains a stack of operators, so programs creating the PDF may sometimes leave unbalanced operations (for examples, saving without restoring graphical states) wouldn't cause problem in viewing since the stack is cleared for next page. However, when the XObject is constructed, all the operators are concatenated, so syntax errors may occur in those cases. In addition, some programs may compress the content stream data with LZW or other proprietary methods, which PDF Everywhere cannot decode. However, since Acrobat 4.x and later uses Zlib compression, it normally wouldn't cause this problem.

3.6 Table-based Layout Scheme

As described in section 1.3, this layout scheme is intended to facilitate sizing and position page contents. PDF Everywhere uses a stack to manage the tables, which are modified and accessed by the following functions:

To get the dimensions and boxes of a page:

```
Page::getWidth( ): number
Page::getHeight( ): number
Page::getCropBox( ): Rect
Page::getArtBox( ): Rect
```

To change (cropping changes crop box and padding changes art box) boxes of a page:

```
Page::crop( $marginleft: size, $marginbottom: size,
   $marginright: size, $margintop: size )
Page::pad( $paddingleft: size, $paddingbottom: size,
   $paddingright: size, $paddingtop: size )
```

To get the default graphics layer of a page:

```
Page::getGraphics( ): GraphContent
```

To draw a new table and retrieve a table cell (or cells block):

```
Page::drawTable( $width: size, $height: size, $fill: bool,
   $attr: HASH )
Page::drawTableAt( $rect: Rect, $fill: bool, $attr: HASH )
PDFDoc::getCell( $row: int, $col: int, $rowspan: int,
   $colspan: int ): Rect
```

To disable current page and return to upper level of table on a page:

```
PDFDoc::undefCurrTable( )
```

To disable all tables defined on a page:

```
PDFDoc::resetAllTables( )
```

The above functions do not have XML tags. Section 2.3 has details about the page box models. You would need the page width, height, crop box, and art box as returned by the accessor methods shown above. Although margins and paddings are defined when the `Page` is created, you can change them by using functions crop and pad, passing a list of 1 to 4 values as sizes, in the order of ***left***, ***bottom***, ***right***, and ***top***.

The function `drawTable` draws a table starting from given position. Note that the *current coordinates* specify the bottom-left corner of the table, and the table extends to the right (`width`, x direction) and to the top (`height`, y direction). If `$fill` is set 1, the table cells are filled with current background color. To change current coordinates and to set the background color, you need to get hold of the default page graphics layer by calling `getGraphics` then calling `moveTo` and `setColor`. Function `drawTable` and `drawTableAt` are delegates to the function `drawGrid` and `drawGridAt` defined `GraphContent`, please see §6.5 for detailed information.

`$attr` in function `drawTable` refers to a hash defining the rows, columns, and other properties of the table. Right now it is sufficient to know that you can define the following keys in this hash: one of `XGrids` or `XSpacing`, one of `YGirds` or `YSpacing`. `XGrids` and `YGrids` specify how many columns and rows (evenly distributed) are to be created; `XSpacing` and `YSpacing` specify column width and row height (the program will create just enough columns and rows to fit in the given width and height).

Functions `undefCurrTable` and `resetAllTables` pops and clears the stack of tables. Function `getCell` returns a `Rect` covering the cells as specified by the parameters. As shown in Figure 3-2, the cells are numbered left to right, top to bottom.

The table-based layout scheme favors XML authoring in that the tags can specify relative width and height values and relative positions.

(1, 1)	(1, 2)	(1, 3)
(2, 1)	(2, 2)	(2, 3)
(3, 1)	(3, 2)	(3, 3)

Figure 3-2. Table cell numbering.

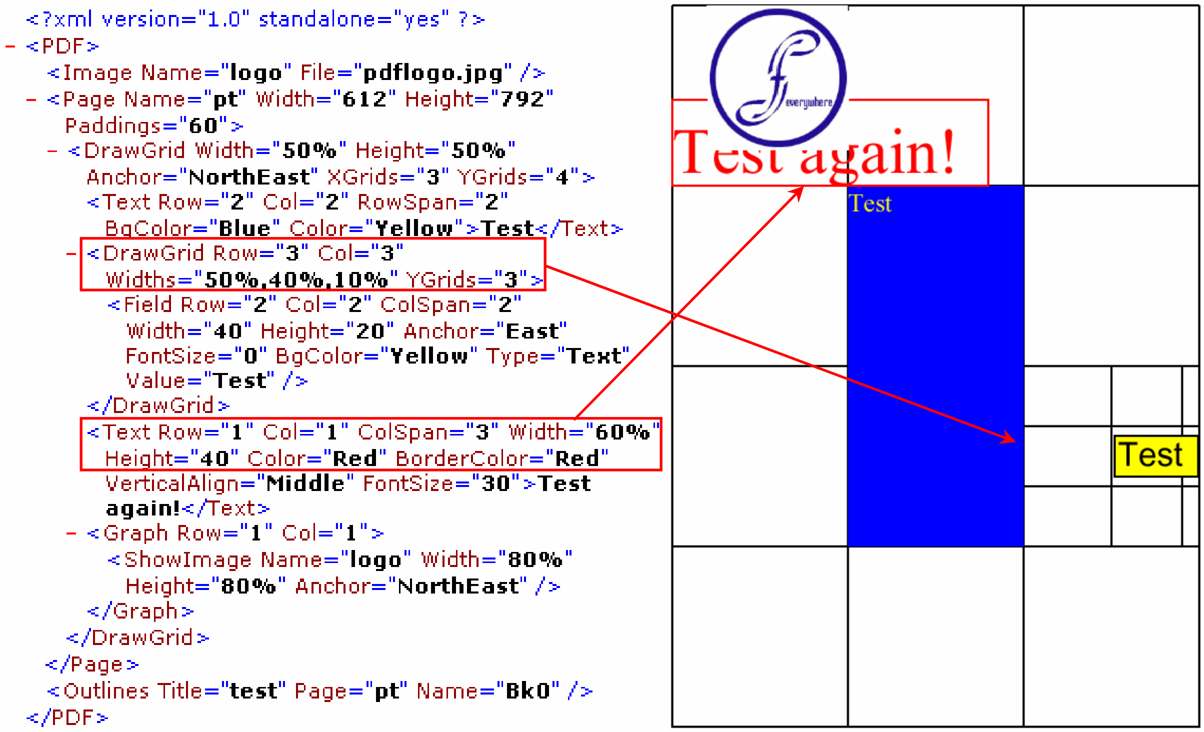
In Figure 3-3, the use of tags `DrawGrid`, `Text`, and `Graph` utilizing the tables are illustrated. A pair of `<DrawGrid></DrawGrid>` creates a table and defines its scope, with attributes similar to those defined in function call `GraphContent::drawGrid` and `Page::drawTable`, but `Width` and `Height` can be set as percentages, relative to page's art box (for top-level `DrawGrid` tags) or to the cell defined using the `Row`, `Col`, `RowSpan` and `ColSpan` (for nested `DrawGrid` tags). `Text` and `Graph` layers can be defined similarly. Tag `showImage` (and `showXObject`) can also use `anchor` to place the image.

In Figure 3-3, the attribute `Padding` in `Page` tag means that the page content would be hosted in the art box with a 60-pixel distance from all sides. This box becomes the first (and the only) top-level cell for this page. The first `DrawGrid` tag is called on the `Page` object, which forward the call to a `GraphContent` object to do the actual work. Note that the and `X`, `Y` position of the table is not defined, but the `Width`, `Height` (in percentage), and `Anchor` are. Therefore, the program would draw a table of ¼ size of the art box, and align it to the top-right corner (`NorthEast`).

Within the `DrawGrid` tag, a `Text` tag creates a text box at row 2, cell 2, spanning over two rows (the blue area at the center). The second `Text` tag creates another text block with respect to the entire first row, but with a width of 60% only and aligned to lower-left corner of the row by default.

The second `DrawGrid` tag creates a 3 × 3 grid in cell 3 or row 3. Then, a form `Field` is created within this nested table, from cell 2 to cell 3 in its row 2. Finally, a `GraphContent` object is created in row 1 cell 1 of the first table, when the second

`DrawGrid` tag has ended and thus its cell definition discarded. The `ShowImage` tag shows a logo with a relative width and height value in percentages but is always anchored to the `NorthEast` corner of this cell.



```
  <?xml version="1.0" standalone="yes" ?>
- <PDF>
    <Image Name="logo" File="pdflogo.jpg" />
  - <Page Name="pt" Width="612" Height="792"
      Paddings="60">
    - <DrawGrid Width="50%" Height="50%"
        Anchor="NorthEast" XGrids="3" YGrids="4">
        <Text Row="2" Col="2" RowSpan="2"
          BgColor="Blue" Color="Yellow">Test</Text>
      - <DrawGrid Row="3" Col="3"
          Widths="50%,40%,10%" YGrids="3">
          <Field Row="2" Col="2" ColSpan="2"
            Width="40" Height="20" Anchor="East"
            FontSize="0" BgColor="Yellow" Type="Text"
            Value="Test" />
        </DrawGrid>
        <Text Row="1" Col="1" ColSpan="3" Width="60%"
          Height="40" Color="Red" BorderColor="Red"
          VerticalAlign="Middle" FontSize="30">Test
          again!</Text>
      - <Graph Row="1" Col="1">
          <ShowImage Name="logo" Width="80%"
            Height="80%" Anchor="NorthEast" />
        </Graph>
      </DrawGrid>
    </Page>
    <Outlines Title="test" Page="pt" Name="Bk0" />
  </PDF>
```

Figure 3-3. XML code illustrating the use of table-based layout scheme.

The aforementioned relative sizes and positions are not directly applicable to object constructors in Perl program. You would have to use `getCell` function to retrieve a rectangle area then size and move it.

Internally, the tables stack is managed by the `PDFDoc` object.

Table 3-2. Attributes for XML tags in the table-based layout scheme.

Attributes	Type	Explanation
`Row`	Integer	Row number.
`Col`	Integer	Column number.
`RowSpan`	Integer	Rows to be spanned over.
`ColSpan`	Integer	Column to be spanned over.
`Width`	Size / percentage	Desired width (absolute or relative).
`Height`	Size / percentage	Desired height (absolute or relative).
`Anchor`	String	Anchor position in the cells block.

The attributes listed in Table 3-2 are prepared for making use of the table cells, and are valid for text box, graphics layer, annotation, form fields created via XML tags `<Text>`, `<Graph>`, <`Annot>`, and `<Form>`. They are also used in functional tags `<DrawGrid>` and `<ShowImage>`.

3.7 Imported Page

When a page is imported from a PDF file, the Page object acts as a delegates and placeholder of the corresponding imported object:

- It reserves the position in the document object tree;
- It reports the object ID of the imported object as if it were its own, although it does have a different ID;
- It doesn't create PDF code, rather, it modifies the data fields of the later to do so;
- It does create XML code.

All original content is imported as-is; the program doesn't even attempt to understand the data. Annotations are imported as well, but they are inactive even if they used to be (since destinations and actions associated with these annotations, if any, are removed). The reason is that the destinations typically links to other pages, which could contain other annotations, and because of the dependencies a large chunks of data would be unnecessarily imported.

When generating PDF code, since these Page objects doesn't produce anything, there will be unused entries in the Xref table just before the file trailer is printed. PDF Everywhere assigns a generation ID of 1 for these so-called "free entries".

Imported page are not locked; new layers of text and graphics, and annotations and form fields can still be added to the page. Sometimes you want to add a watermark to the page; in that case you would assign the property `AlwaysOnTop` a non-zero value when creating an imported page. Please note there are chances that the imported page is opaque with a white-colored background.

Example 3-6 stamps every page of a PDF file with a watermark. The job is simplified by the table-based layout model.

Example 3-6. Applying a watermark to an PDF file.

```
use PDF;
$doc = new PDFDoc( );
$img = $doc->newImage( 'mylogo.jpg' );
$pdf = new PDFFile( 'review.pdf' );
for( 1..$pdf->getPageCount( ) ){
   my $pg = $doc->newPage( { ImportSource => $pdf, ImportPage =>
      $_-1, AlwaysOnTop => 1 } );
   $pg->getGraphics( )->showImage( 0, 0, $img, { Anchor =>
      'Center' } );
}
$doc->writePDF( 'stamped.pdf' );
```

3.8 Z-index of Page Contents

Each content object is considered occupying a specific layer, and the page printout is considered the merger of these “stacked” layers, with optional blending modes or opacity settings applied.

In this toolkit, each layer is independently manipulated, thus their sequence can be arbitrarily set, without leaving residue effects when their contents are concatenated in the stacking order.

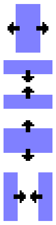
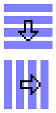
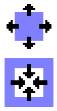
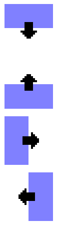
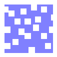
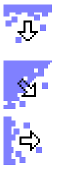
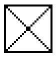
There is an attribute called `ZIndex` that can be used in the instantiation of text and graphics layers, with a default value of 0. This value can be a negative or positive integer. The layers will be displayed in the order of `ZIndex` values, largest on top. If two or more layers have the same `ZIndex` value, then they are displayed in the order of creation with newest on top.

3.9 Page Transition Effects

Adobe defined 7 different types of transition effects for a page. Related attributes are listed in Table 3-1, and the required attributes for each transition type are listed in Table 3-3. Transition duration is optional for all types. By default, a transition effect lasts for 2 seconds.

A page can be auto-flipped after being displayed for a few seconds, determined by the presence and value of the attribute `Duration`. This duration doesn’t include the time elapsed for the transition effect (`TransDur`).

Table 3-3. Attributes required for different transition effects.

Transition values		1	2	3	4	5	6	7
Key	**Type**	*Split*	*Blinds*	*Box*	*Wipe*	*Dissolve*	*Glitter*	*(No effect)*
`TransDur`	Bool	•	•	•	•	•	•	
`Direction`	Bool				•		•	
`Dimension`	Bool	•	•					
`Motion`	Bool	•		•				
Illustrations								

3.10 Bookmarks (Outlines)

A bookmark, or an `Outlines` entry, is usually linked to specific position in the document at a specific zoom factor. Entries are displayed in Acrobat in a cascade style, (a tree), and their internal data representations are also organized into a tree, with the tree root, an `Outlines` entry, is automatically created by the `PDFDoc` object and retrieve by the function `getOutlinesRoot`:

```
PDFDoc::getOutlinesRoot( ): Outlines
```

To create an entry, use function `newOutlines` defined in `PDFDoc`, which forwards the call to the constructor of the class `Outlines`. The attributes are listed in Table 3-4.

```
PDFDoc::newOutlines( $title: string, $attr: HASH )
```

Table 3-4. Available attributes for an Outlines entry.

Key	**Type**	**Explanation**					
`Page`	Page	Reference or name of destination page, if not current page.					
`Name`	String	Unique name for the object.					
`Parent`	Outlines	Parent `Outlines` object, optional.					
`Fit`	String	How the destination page will be displayer in window. Must be `Fit`, `FitH` (fit horizontally), `FitV` (fit vertically), `FitR` (fit in rectangle), or `XYZ` (align to X-Y at zoom level Z).					
		Required for fit methods?	`Fit`	`FitH`	`FitV`	`FitR`	`XYZ`
`Left`	Size	Left position.			●	●	●
`Top`	Size	Top position.		●		●	●
`Right`	Size	Right position.				●	
`Bottom`	Size	Bottom position.				●	
`Zoom`	Integer	Zoom factor as a percentage.					●

```
<Outlines Title="string" Page="pagename" <!-- More attributes --> >
    <!-- More Outlines tags -->
</Outlines>
```

All newly created entries are added to the root (unless attribute `Parent` is set) in the order of creation. To move them to correct places in the hierarchy, call the function `add` on a desired branch node. The same entry is also the return value. The function is also called `appendEntry`.

```
Outlines::add( $entry: Outlines ): Outlines
```

In XML, all `<Outlines>` tags as a group should be placed at the end of file, just before the ending `</PDF>` tag. The reason is that the referred pages must have been defined.

- **Numbering**

To number all the sub-entries of a given `Outlines`, call the method `numbering`, optionally with a hash reference that contains two keys: `Style` and `Start`. The values for `Style` can be `A`, `a`, `1`, `i`, and `I` (roman system), similar to defining an `<OL>` tags in HTML. The value for `Start` should be a positive integer (default: 1). The call is not recursive (only bookmarks at the same level are numbered).

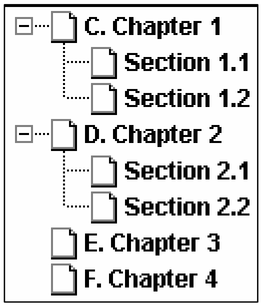
```
Outlines::numbering( $attr: HASH )
```

Example 3-6 shows the creation and organization of bookmarks. You can choose to export PDF or XML output. In the XML output that follows the code, you can see that the hierarchy of the entries is represented in the similar way via nested `<Outlines>` tags. The top level bookmarks are also numbered.

A text box, a graphics layer, or even a page, created by `newTextBox`, `newGraph`, or `newPage` defined in class `PDFDoc` (or equivalently, the new operations defined in classes `FloatingText`, `GraphContent`, and `Page`), can each have an optional attribute "`Bookmark`", whose value is a string, that automatically creates a top-level bookmark entry that when clicked will fit the destination box on screen.

Example 3-7. Creating and organizing bookmark entries.

```
use PDF;
$doc = new PDFDoc( { PageMode => 'UseOutlines' } );
$doc->newPage( );
$doc->newGraph( );
my $oroot = $doc->getOutlinesRoot( );
$ch1 = $oroot->add( new Outlines( 'Chapter 1' ) );
$ch1s1 = $ch1->add( new Outlines( 'Section 1.1' ) );
$ch1s2 = $ch1->add( new Outlines( 'Section 1.2' ) );
$ch2 = $oroot->add( new Outlines( 'Chapter 2' ) );
$ch2s1 = $ch2->add( new Outlines( 'Section 2.1' ) );
$ch2s2 = $ch2->add( new Outlines( 'Section 2.2' ) );
$ch3 = $oroot->add( new Outlines( 'Chapter 3' ) );
$ch4 = $oroot->add( new Outlines( 'Chapter 4' ) );
$oroot->numbering( {'Start'=>3, 'Style'=>'A'} );
$doc->writePDF( 'sample3-7.pdf' );
#$doc->exportXML( );
```



```
⊟ C. Chapter 1
    Section 1.1
    Section 1.2
⊟ D. Chapter 2
    Section 2.1
    Section 2.2
  E. Chapter 3
  F. Chapter 4
```

```
<?xml version="1.0" standalone="yes"?>
<PDF>
   <Page Name="N000" Width="612" Height="792">
      <Graph Rect="0, 0, 612, 792" Name="N001">
      </Graph>
   </Page>
   <Outlines Title="C. Chapter 1" Page="N000" Name="N002">
      <Outlines Title="Section 1.1" Page="N000" Name="N003" Parent="N002"
      />
      <Outlines Title="Section 1.2" Page="N000" Name="N004" Parent="N002"
      />
   </Outlines>
   <Outlines Title="D. Chapter 2" Page="N000" Name="N005">
      <Outlines Title="Section 2.1" Page="N000" Name="N006" Parent="N005"
      />
      <Outlines Title="Section 2.2" Page="N000" Name="N007" Parent="N005"
      />
   </Outlines>
   <Outlines Title="E. Chapter 3" Page="N000" Name="N008" />
   <Outlines Title="F. Chapter 4" Page="N000" Name="N009" />
</PDF>
```

Chapter 4: Shape, Color, and Resources

This chapter describes the assistant classes that create objects used by document tree nodes. Instances of these classes do not participate PDF or XML generation. They are used as data types.

4.1 Rectangle and Polygon

Rectangle (`Rect`) and polygon (`Poly`) are the two shapes defined in PDF Everywhere, both extended from an abstract base `Shape`, which defines the following accessor methods for properties `Left`, `Bottom`, `Right`, `Top`:

```
Shape::left( ): number
Shape::bottom( ): number
Shape::right( ): number
shape::top( ): number
```

A `Rect` is initiated with four values as left, bottom, right, and top positions, or with another `Shape` object (`Rect` or `Poly`). In the latter case, the bounding box of the shape is returned. A `Poly` is created from a list of (x, y) value pairs defining the vertices. The polygon is always closed by connecting the first and the last vertices.

`Rect` is enhanced with the following methods

```
Rect::width( ): number
Rect::height( ): number
Rect::width( $newwidth: size )
Rect::height( $newheight: size )

Rect::moveBy( $offsetx: size, $offsety: size )
Rect::shrink( $offset: size )
Rect::union( $rect: Rect ): Rect
Rect::anchorRect( $width: string, $height: string, $anchor: string,
   $that: Rect ): Rect
```

Function `moveBy` moves the rectangle by given offsets; function `shrink` shrinks all four sides inwards by a given offset (or expand it outward if offset is negative). `Union` is used to create a new `Rect` (without changing anything) that covers both rectangles. Function `anchorRect` is used internally to facilitate the table-based layout scheme by resizing and moving the `Rect` (the 4th argument) as well as returning it to caller: both width and height can be a string specifying a size of a percentage (relative to the `Rect` it is invoked on). Anchor must be one of the following (case significant):

`North`, `NorthEast`, `NorthWest`, `West`, `Center`, `East`, `SouthWest`, `South`, `SouthEast`

Poly defines a method to return all the coordinates in pairs in a list:

```
Poly::getPoints( ): numbers[]
```

4.2 Colors

The default color space is `RGB`, and each color is printed in PDF code as three float numbers between 0 and 1 specifying the weight of color components red, blue, and green. When color space is set to `CMYK`, four values are printed for cyan, magenta, yellow, and black components. For `Gray` color space, only one value is printed.

To set color space at any time, call this function on a PDFDoc object:

```
PDFDoc::setColorSpace( $space: string )
```

The value for $space must be one of RGB, CMYK, and Gray. Note the change of color space doesn't affect operations (e.g. drawings) already performed.

- **Specifying a color**

Many objects and functions accept color values, such as when drawing a circle and setting text color. These values are always strings, which can either be a color name, a string expressing the weight of the color components (ranging from 0 to 1), or a color value in hex form (each component ranges from 00 through FF) with optional prefix '#' as in HTML. For example, "Limegreen", "32CD32", "#32CD32", and "0.196, 0.804, 0.196" express the same color in RGB color space.

PDF Everywhere assumes that you are specifying a color in `CMYK` form if the hex or list form contains four values, or `Gray` form if there is only one value. Internally, the color is always expressed in `RGB` form. The following function is frequently used through out the programs, which returns a list of values for a given color in given color space (this function is exported to the `main` namespace):

```
Color::tellColor( $color: string, $space: string ): number[3]
```

The program replies on the following conversion formula:

RGB → CMYK:

```
 K = min( 1-R, 1-G, 1-B)
    C = (1-R-K)/(1-K)
    M = (1-G-K)/(1-K)
    Y = (1-B-K)/(1-K)
```

CMYK → RGB:

```
R = 1 - min(1,C*(1-K)+K)
G = 1 - min(1,M*(1-K)+K)
B = 1 - min(1,Y*(1-K)+K)
```

RGB → Gray:

```
g = 0.2125 * R + 0.7154 * G + 0.0721 * B
```

- **Register a color name**

PDF Everywhere recognizes all predefined HTML color names (case insignificant), as listed in Table 4-1.

Table 4-1. Predefined color names.

Name	RGB	Name	RGB	Name	RGB
Antiquewhite	FAEBD7	Aqua	00FFFF	Aquamarine	7FFFD4
Azure	F0FFFF	Beige	F5F5DC	Bisque	FFE4C4
Black	000000	Blanchedalmond	FFEBCD	Blue	0000FF
Blueviolet	8A2BE2	Brown	A52A2A	Burlywood	DEB887
Cadetblue	5F9EA0	Chartreuse	7FFF00	Chocolate	D2691E
Coral	FF7F50	Cornflowerblue	6495ED	Cornsilk	FFF8DC
Crimson	DC143C	Cyan	00FFFF	Darkblue	00008B
Darkcyan	008B8B	Darkgoldenrod	B8860B	Darkgray	A9A9A9
Darkgreen	006400	Darkkhaki	BDB76B	Darkmagenta	8B008B
Darkolivegreen	556B2F	Darkorange	FF8C00	Darkorchid	9932CC
Darkred	8B0000	Darksalmon	E9967A	Darkseagreen	8FBC8F
Darkslateblue	483D8B	Darkslategray	2F4F4F	Darkturquoise	00CED1
Darkviolet	9400D3	Deeppink	FF1493	Deepskyblue	00BFFF
Dimgray	696969	Dodgerblue	1E90FF	Firebrick	B22222
Floralwhite	FFFAF0	Forestgreen	228B22	Fuchsia	FF00FF
Gainsboro	DCDCDC	Ghostwhite	F8F8FF	Gold	FFD700
Goldenrod	DAA520	Gray	808080	Green	008000
Greenyellow	ADFF2F	Honeydew	F0FFF0	Hotpink	FF69B4
Indianred	CD5C5C	Indigo	4B0082	Ivory	FFFFF0
Khaki	F0E68C	Lavender	E6E6FA	Lavenderblush	FFF0F5
Lawngreen	7CFC00	Lemonchiffon	FFFACD	Lightblue	ADD8E6
Lightcoral	F08080	Lightcyan	E0FFFF	Lightgoldenrodyellow	FAFAD2
Lightgreen	90EE90	Lightgrey	D3D3D3	Lightpink	FFB6C1
Lightsalmon	FFA07A	Lightseagreen	20B2AA	Lightskyblue	87CEFA
Lightslategray	778899	Lightsteelblue	B0C4DE	Lightyellow	FFFFE0
Lime	00FF00	Limegreen	32CD32	Linen	FAF0E6
Magenta	FF00FF	Maroon	800000	Mediumaquamarine	66CDAA
Mediumblue	0000CD	Mediumorchid	BA55D3	Mediumpurple	9370DB
Mediumseagreen	3CB371	Mediumslateblue	7B68EE	Mediumspringgreen	00FA9A
Mediumturquoise	48D1CC	Mediumvioletred	C71585	Midnightblue	191970
Mintcream	F5FFFA	Mistyrose	FFE4E1	Moccasin	FFE4B5
Navajowhite	FFDEAD	Navy	000080	Oldlace	FDF5E6
Olive	808000	Olivedrab	6B8E23	Orange	FFA500
Orangered	FF4500	Orchid	DA70D6	Palegoldenrod	EEE8AA
Palegreen	98FB98	Paleturquoise	AFEEEE	Palevioletred	DB7093
Papayawhip	FFEFD5	Peachpuff	FFDAB9	Peru	CD853F

Pink	FFC0CB	Plum	DDA0DD	Powderblue	B0E0E6
Purple	800080	Red	FF0000	Rosybrown	BC8F8F
Royalblue	4169E1	Saddlebrown	8B4513	Salmon	FA8072
Sandybrown	F4A460	Seagreen	2E8B57	Seashell	FFF5EE
Sienna	A0522D	Silver	C0C0C0	Skyblue	87CEEB
Slateblue	6A5ACD	Slategray	708090	Snow	FFFAFA
Springgreen	00FF7F	Steelblue	4682B4	Tan	D2B48C
Teal	008080	Thistle	D8BFD8	Tomato	FF6347
Turquoise	40E0D0	Violet	EE82EE	Wheat	F5DEB3
White	FFFFFF	Whitesmoke	F5F5F5	Yellow	FFFF00
Yellowgreen	9ACD32				

To define a custom color name, you can use the following function:

```
PDFDoc::registerColor($name: string, $color: string, $space: string)
```

```
<Color Name="string" Color="color" Space="string"/>
```

The name will become case-insignificant. The second argument can be assigned any value described above. The last argument specifies the preferred color space for that color. After registration, the color name can be used like any other predefined ones. Note the color name must be unique; otherwise an error will be produced.

- **Adjusting a color**

The following functions lighten, darken, or invert a color in RGB color space. They are defined in package Color, but are also exported to the main namespace for the sake of backward-compatibility:

```
Color::lighten( $color: string, $ratio: number ): string
Color::darken( $color: string, $ratio: number ): string
Color::invert( $color: string ): string
```

Here the `$ratio` is a number between 0 and 1. The return values of all three functions are always in `RRGGBB` hex form.

- **Special color space**

When importing an image, sometimes PDF Everywhere creates a type of object of the class `ColorSpace` to express an indexed color space. This process is done automatically; you don't need to create instances of this class.

4.3 Resources Management

PDF specification defines resources such as `XObject`, `Shading`, `Pattern`, `ColorSpace`, `ExtGState`, and Font. PDF Everywhere defines classes `XObject`, `PDFShading`, `PDFPattern`, `ColorSpace`, `ExtGState`, and `PDFFont` for these resources types.

The management is done automatically. The resources are shared by all pages, but in generating PDF code only those actually used on a page are counted, so that later on if you want to import a page from a PDF file created by PDF Everywhere, those resources not used by that page are not imported, saving much space.

Since the native `Page` object and the imported page both have their own resources dictionaries, when creating PDF the program merges these data then clean up the changes when PDF output is done. For the records, the following ***private*** functions are involved in the process:

```
Page::getResources( ): Resources
Page::mergeResources( )
XObject::mergeResources( )
Resources::merge( $that: Resources )
```

PDF Everywhere does extra work for you: it keeps track of the use of such resources in a page; it monitors the resources defined in any external templates (XML) and those imported for external files to avoid duplication; and it merges imported resources with native ones to correctly build PDF. The objective is an optimized output.

Chapter 5: Font and Text

5.1 Introduction

In PDF Everywhere both TrueType and Type 1 fonts can be used and embedded. Fonts are referred to by names, which are case-significant. The default font used for all text boxes is `Times-Roman` at 12 point; the default font for text labels in a graphics layer is `Helvetica` at 12 point. There are 14 Adobe fonts built in Acrobat that are guaranteed to be available, as listed in Table 5-1. A pair of alternative names is provided for the first 12 fonts: the short name is for compatibility with Acrobat PDF form fields, the long name is for compatibility with previous version of PDFEver. The "-" and "," are considered a part of the name, so please do not omit, nor should you include spaces in the names.

Table 5-1. Adobe Acrobat built-in font names.

Font name	Other names	Sample
Courier	Cour / CourierNew	ABC 123
Courier-Bold	CoBo / CourierNew,Bold	**ABC 123**
Courier-BoldOblique	CoBO / CourierNew,BoldItalic	***ABC 123***
Courier-Oblique	CoOb / CourierNew,Italic	*ABC 123*
Helvetica	Helv / Arial	ABC 123
Helvetica-Bold	HeBo / Arial,Bold	**ABC 123**
Helvetica-BoldOblique	HeBO / Arial,BoldItalic	***ABC 123***
Helvetica-Oblique	HeOb / Arial,Italic	*ABC 123*
Times-Roman	TiRo / TimesNewRoman	ABC 123
Times-Bold	TiBo / TimesNewRoman,Bold	**ABC 123**
Times-Italic	TiIt / TimesNewRoman,Italic	*ABC 123*
Times-BoldItalic	TiBI / TimesNewRoman,BoldItalic	***ABC 123***
Symbol		ABX 123
ZapfDingbats	ZaDb	

Version 2.x defines Courier New, Arial, and Times New Roman as different fonts than their equivalents Courier, Helvetica, and Times in version 3.0. Although there really are subtle differences in character shapes and dimensions, Acrobat since version 4 has been using Arial for Helvetica and Times New Roman for Times-Roman, making the differentiation unnecessary.

To use a font, usually you need to set an attribute `FontFace` with the font name, such as when creating a text box. Form fields can only take one of the first 12 fonts in Table 5-1, but other text can be set to any embedded font (see next section).

Each font used in a PDF document is wrapped in a `PDFFont` object, which is created internally. You don't need to worry about the maintenance of these objects.

5.2 Font Embedment

▪ Embedding TrueType and Type 1 fonts

To embed a new TrueType font, use the method `useTTF` on a `PDFDoc` object to parse the TTF font file content and get the PostScript name for this font. This method builds a hash for this font, and registers it to the main program so that you can use the font in your text with the name it returns. Non-word characters contained in the font's native name are filtered out ("_" and "-" are considered word characters).

Likewise, use `usePFB` to embed a Type 1 font in PFB format (sorry, no PFA). You will need to provide the AFM or PFM file for that font, too.

```
PDFDoc::useTTF(  $TTFFile:  String,  [  $embed:  Boolean,  $name:
String ] ): String
PDFDoc::usePFB( $PFBFile: String, $FMFile: String, [ $embed: Boolean,
$name: String ] ): String
```

Argument `$embed` indicates whether you want to embed the font in the PDF output; the last argument is a name you want to use, in stead of that is automatically created (if the name is already defined, you will get an error). The return value is the font name.

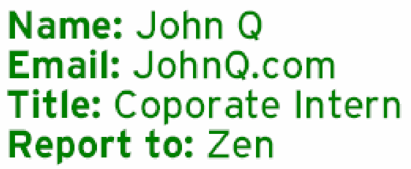
Note the font file must be specified; there is no way to scan the operating system's registry dictionary looking for a font, and the names returned by these functions are very likely not that shown in your system. In particular, each variant of a same font (`Normal`, `Bold`, `Italic`, and `BoldItalic`) is associated with a different font file, and the PDF Everywhere cannot establish this association for you.

In Example 5-1, two styles of a same font, Interstate, are embedded and associated with each other, so the program knows when which font to use when switching from normal to bold text, or from bold to normal. The text box is initiated with the normal style; when a marker "`^b`" is read, the program switches to bold style; when the "`^B`" is read, it switches back to normal. Both "`^b`" and "`^B`" are inline style change markers, like the tags in HTML but much more simple and limited. We will talk more about these markers in this chapter. In Example 5-2, custom names are assigned to embedded fonts. This example also illustrates that the association of fonts are in fact arbitrary.

Example 5-1. Font embedment.

```
use PDF;
my $doc = new PDFDoc( );
# Now embed two fonts (as Normal and Bold style of Interstate font
my $ttfbold = $doc->useTTF( 'Interstate-Bold.ttf', 1 );
my $ttfreg = $doc->useTTF( 'Interstate-Regular.ttf', 1 );
# Associate these two fonts
$doc->setFontVariant( $ttfreg, 'Bold', $ttfbold );
$doc->setFontVariant( $ttfbold, 'Normal', $ttfreg );
$doc->newPage( );
# Now create a text box
```

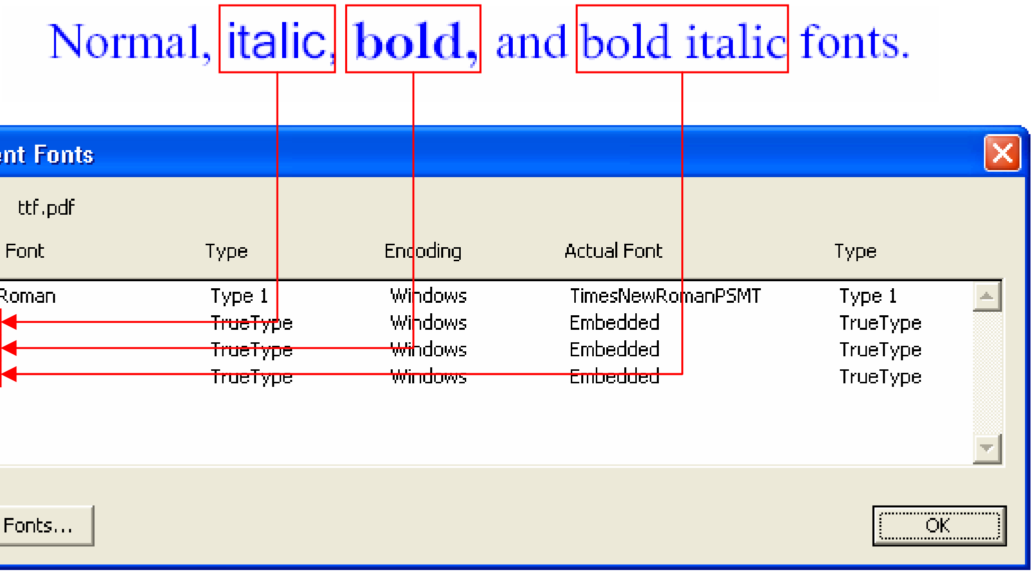
```
$doc->newTextBox(
   new Rect( '2in', '5in', '8in', '9in' ),
   qq{
         ^b Name: ^B  John Q ^n
         ^b Email: ^B JohnQ.com ^n
         ^b Title: ^B Coporate Intern ^n
         ^b Report to: ^B Zen
   }, {
   FontSize    => 14,
   FontColor   => 'green',
   FontFace    => $ttfreg,
   } );
$doc->writePDF( 'ttf.pdf' );
```



Name: John Q
Email: JohnQ.com
Title: Coporate Intern
Report to: Zen

Example 5-2. Font association.

```
use PDF;
my $doc = new PDFDoc( );
my $fa = $doc->useTTF( 'tiffany.ttf', 1, 'FontA' ); # Custom name
my $fb = $doc->useTTF( 'micross.ttf', 1, 'FontB' );
my $fc = $doc->useTTF( 'sylfaen.ttf', 1, 'FontC' );
# Now associate four fonts arbitrarily, although not recommended
$doc->setFontAllVariants( {
   Normal      => 'Times-Roman',
   Bold        => 'FontA',
   Italic      => 'FontB',
   BoldItalic  => 'FontC',
} );
$doc->newPage( );
$doc->newTextBox(
   new Rect( '2in', '5in', '8in', '9in' ),
   qq{Normal, ^i italic, ^I ^b bold, ^B and ^ib bold italic ^IB
fonts.}, { FontSize  => 14, FontColor  => 'blue', } );
$doc->writePDF( 'ttf.pdf' );
```



Normal, italic, bold, and bold italic fonts.

Document Fonts

Fonts in: ttf.pdf

Original Font	Type	Encoding	Actual Font	Type
Times-Roman	Type 1	Windows	TimesNewRomanPSMT	Type 1
FontB	TrueType	Windows	Embedded	TrueType
FontA	TrueType	Windows	Embedded	TrueType
FontC	TrueType	Windows	Embedded	TrueType

- **Font variants**

When embedding multiple fonts, you might want to associate different variants of a same font using the function `setFontVariant` defined in class `PDFDoc`: `$rel` must be `Normal`, `Bold`, `Italic`, or `BoldItalic` denoting the style, `$encoding` must be `WinAnsiEncoding` (for TrueType fonts) or `StandardEncoding` (for Type 1 fonts). The association is one-way only.

```
PDFDoc::setFontVariant( $ToFont: String, $rel: String, $FromFont:
String, [ $encoding: String ] )
PDFDoc::setFontAllVariants( $rels: HASH, [ $encoding: String ] )
```

Function `setFontAllVariants` sets all four variants of a font at the same time via a hash that contains exactly four keys (the four styles), each with a value that is the corresponding font name (see Example 5-2).

- **XML tags**

The following XML tags correspond to the functions described above:

```
<Font                <!-- PDFDoc::useTTF -->
   Embed="1/0"       <!-- Optional; If 1, font will be embedded -->
   Name="name"       <!-- Optional name for the font -->
   TTF="filename"    <!-- TrueType file name -->
/>
<Font                <!-- PDFDoc::usePFB -->
   Embed="1/0"       <!-- Optional; If 1, font will be embedded -->
   Name="name"       <!-- Optional name for the font -->
   PFB="filename"    <!-- Type 1 file name -->
   FM="filename"     <!-- AFM or PFM file name -->

/>

<FontVariant         <!-- PDFDoc::setFontVariant -->
   FromFont="font"
   ToFont="font"
   Rel="Normal|Bold|Italic|BoldItalic"
   Encoding="WinAnsiEncoding|StandardEncoding"
/>
<FontVariants        <!-- PDFDoc::setFontAllVariants -->
   Normal="fontname"
   Bold="fontname"
   Italic="fontname"
   BoldItalic="fontname"
   Encoding="WinAnsiEncoding|StandardEncoding"
/>
```

These XML tags are equivalent to the function calls in Example 5-2:

```
<Font Name="FontA" Embed="1" TTF="tiffany.ttf"/>
<Font Name="FontB" Embed="1" TTF="micross.ttf"/>
<Font Name="FontC" Embed="1" TTF="sylfaen.ttf"/>
<FontVariants Normal="Times-Roman" Bold="FontA" Italic="FontB"
   BoldItalic="FontC" Encoding="WinAnsiEncoding"/>
```

5.3 How Fonts Are Managed

You can skip this part as it talks about some internals that your program should not worry about. The hash `%PDFFont::Fonts` stores data for all 14 built-in fonts. Each font has all possible characters defined, and a character set is built for each encoding scheme that is requested in a program. The width value of each character is needed for correct layout of text.

Each PDF document represented by a `PDFDoc` object manages its own font objects separately. A same font with different encoding schemes is considered as different fonts. The font embedment and variant associations are independent among documents.

When a font is requested, the program first checks its true name (in case it is one of the variable font names listed in Table 5-1), if it is one of the predefined names, it returns the font object (`PDFFont`) or build a new one if it hasn't been constructed (note this is a ***private*** function that you must not call, and the returned `PDFFont` is a ***private*** object that you must not change):

```
PDFDoc::getFont( $FontName: String, $Encoding: String ): PDFFont
```

If the name is not found, it returns a `PDFFont` object representing `Time-Roman`. If the font is one of the 14 built-in fonts, the program constructs the character width array. Otherwise, a TrueType font can only be used with `WinAnsiEncoding`, while an embedded Type 1 font can only be used with `StandardEncoding` – these settings are not affected by the encoding of the document for built-in fonts. This is why the parameter `$encoding` in functions `setFontVariant` and `setFontAllVariants` (and the attribute `Encoding` in the XML tags `FontVariant` and `FontVariants`) takes only one of these two values.

When a font is registered to a document using either `useTTF` or `usePFB`, the document add an entry in its font data, but an actually `PDFFont` object is not created until the font is used somewhere in the document. Therefore, the output PDF may not contain all fonts that are registered.

- **Automatic objects**

A font descriptor accompanies each non-built-in font. This descriptor, wrapped in a `FontDescript` object, is created automatically and stores the information about a font such as character width, stem dimension, bounding box, etc. The width value is a quantity with the unit of 1/1000 of 1 point.

For an embedded font, an additional `EmbeddedFont` object is created to store the font data. It is a specialized `PDFStream` object, with data retrieved from the original font file.

- **Form fonts**

PDF form fields use a special encoding scheme `PDFDocEncoding`. For form fields (text box, drop-down list, etc.), only the first 12 built-in fonts can be used. Check boxes and radio buttons use `ZaDb` for the symbols (captions are not part of the fields). It is impossible to use an embedded font face for a form field.

5.4 Text Box

A text box is a layer of text on the page. The box can actually be a rectangle (`Rect`) or polygon (`Poly`); text will flow with the shape. If the shape can not hold all text to be displayed, you have a choice of whether to stop the text flow there or continue by extending the shape downwards.

You can set text attributes such as font face, size, color, line height, alignment and padding. The box also maintains an array for all the annotations (underline, email address, URL's, etc.) shown in this box, so that when the vertical alignment or text rotation is performed the annotations will be moved correctly. The positions and dimensions of these annotations are calculated by recording the (x, y) coordinates before and after the targeted text segment is displayed. Text justification and vertical alignment are done by adjusting the starting point of the line/block, and the word spacing may be adjusted also. Indentation is made by moving the whole line of text, and padding is made by shrinking the container box or shortening the available line width.

To create such a text box, you can call the following function:

```
PDFDoc::newTextBox( $shape: Rect | Poly, $text: String, $attr:
    HASH) : FloatingText
```

The first argument is either a `Rect` or a `Poly` object, the second is the text to be displayed in the shape, and the third defines the attributes of the text box. The return value is actually an object of type `FloatingText`, which is a subclass of `TextContent` (which is also the base class for `PreformText` used for preformatted text). The XML tag to do so is:

```
<Text
   Rect="x1,y1,x2,y2" <!-- A list of four dimension values -->
   FontFace="fontname"
   <!-- More attributes can be inserted here -->
><![CDATA[<!-- The text to be displayed -->]]></Text>
```

All attributes defined in the `$attr` can be used as attributes in the XML tag. You are advised to give it a `Name`. The text MUST be enclosed in the CDATA section. As a reminder, the text can ONLY be encoded in ISO-8859-1; there is no Unicode support.

- **Text direction and rotation**

Text doesn't have to be displayed left to right, top to bottom. The attribute `TextDir` determines how the text should be displayed, as seen in Figure 5-1. Text alignment (horizontal or vertical) remains in effect. This attribute is overlooked if the box is not a rectangle.

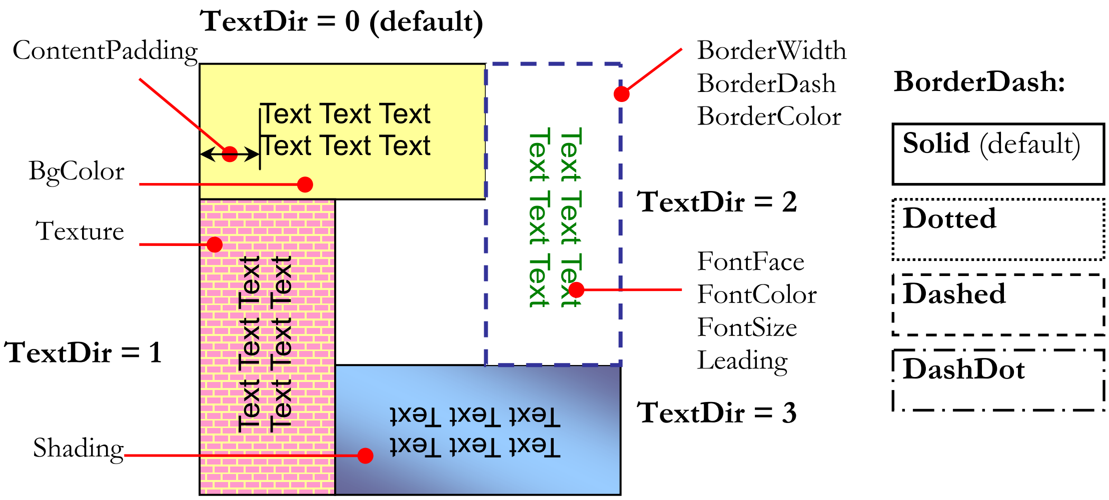


Figure 5-1. Attributes for rectangular text boxes.

If text direction is not specified, you can rotate the entire text box by an arbitrary angle (in degree; counterclockwise is positive), with the attribute `Rotation`. There is much difference between `TextDir` and `Rotation`:

1. With `TextDir` the text always stays inside the box; with `Rotation`, the text is rotated around the top-left corner and is not bounded by the box;
2. With `Rotation`, the shape can be a polygon;
3. With `TextDir`, annotations are created and their positions are adjusted; with `Rotation`, no annotations are created.

- **Text alignment**

Both horizontal and vertical alignments are supported: attribute `TextJustify` defines horizontal alignment, taking a value of `Left` (default), `Center`, `Right`, or `Uniform`, and `VerticalAlign` defines vertical alignment, taking a value of `Top` (default), `Middle`, or `Bottom`. Figure 2-4 illustrates the combination of these settings.

Note: When a polygon is used as the hosting shape, vertical alignment is always `Top`.

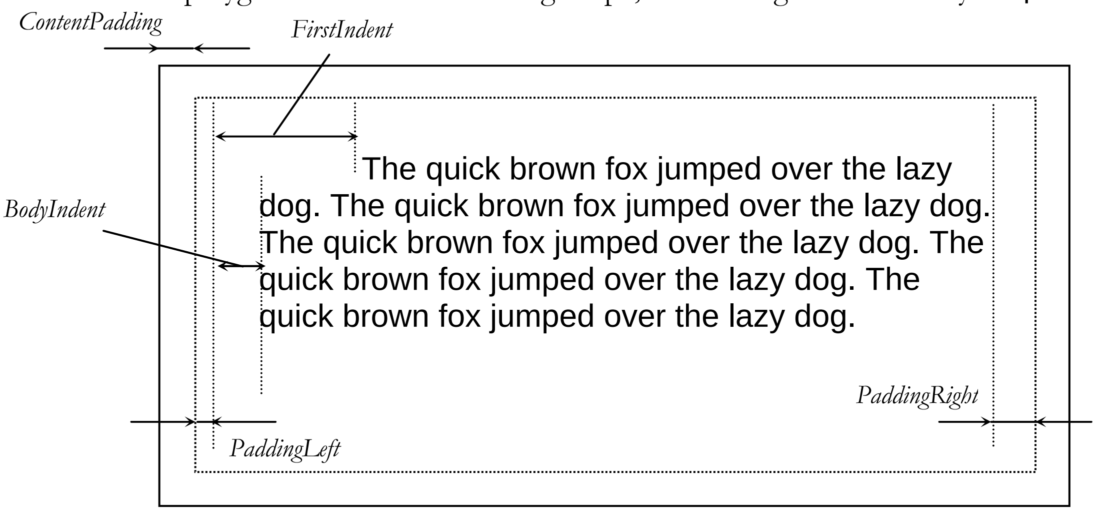

ContentPadding
FirstIndent
BodyIndent
The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.
PaddingRight
PaddingLeft

Figure 5-2. Definition of padding and indentation for a text box.

Table 5-2. Available properties for text boxes during instantiation.

	Key	Type	Explanation	Default
Font settings	FontFace	String	Default font used for text boxes.	Times-Roman
	FontSize	Size	Default font size.	12
	FontColor	Color	Font color.	black
	Leading	Size	Leading (line height).	12
	CharSpacing	Size	Character spacing	0
	WordSpacing	Size	Word spacing (affect space characters only).	0
Text alignment settings	TextDir	Integer	Text direction: 0 (normal), 1 (90° counterclockwise), 2 (90° clockwise), 3 (180°).	0
	Rotation	Float	Rotation angle of the text box (-180 to 180).	0
	TextJustify	String	Horizontal alignment: must be Left, Center, Right, or Uniform.	Left
	VerticalAlign	String	Vertical alignment: must be Top, Middle, or Bottom.	Top
	ContentPadding	Size	Content padding.	0
	PaddingLeft	Size	Left-side padding.	0
	Paddingright	Size	Right-side padding.	0
	BodyIndent	Size	Body text indentation.	0
	FirstIndent	Size	First line indentation.	0
	TabPosition	Size	Default tab position measured from left padding.	1in
	ExtraParaSpacing	Bool	If non-zero, an extra line break will be introduced after each paragraph (hard line break).	
Background and borders	BgColor	Color	Background color.	undef
	Texture	Object	A PDFTexture object defining background texture pattern.	Undef
	Shading	Object	A PDFShading object defining background shading (gradient) pattern.	Undef
	BorderColor	Color	Border color.	Undef
	BorderWidth	Float	Border line width in points.	Undef
	BorderDash	String	Border dash pattern: can be Dashed, Dotted, Solid, or DashDot.	Solid
	Opacity	Integer	Between 0 to 100	100
	BlendMode	String	Blending mode of this text box as a layer.	Normal
Document object manipulation	Name	String	A unique name for this text box object.	(auto)
	IsTemplate	Bool	If non-zero, the object is added to document's internal template collection once created.	Undef
Additional	Bookmark	String	Bookmark title for this text box object.	Undef
	ZIndex	Integer	Stack sequence on the host page.	0
	IsContinued	Bool	If non-zero, the text box is considered the continuation of a previous one.	Undef

- **Text attributes**

  The attributes of the text include font face, size, color, leading (line height), word spacing, and character spacing.

- **Padding and indentation**

  As shown in Figure 5-2, you can set the padding and indentation values for the text box.

- **Bookmark**

  You can add a bookmark for the text box. When the user clicks on the bookmark title, the viewer window will zoomed to fit the text box to the window. See Example 5-3. To change the hierarchy of the bookmarks, use the `getBookmark` method to retrieve the newly created bookmark and use the `add` method on the desired parent bookmark entry.

Example 5-3. Bookmark for a text box.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( );
my $str = "test test test";
my $box = $doc->newTextBox( new Rect( 40, 40, 200, 200 ), $str, {
   TextJustify => 'Center', VerticalAlign => 'Middle',
   Bookmark => 'My sample bookmark', BorderColor => 'Red' } );
# Create a top-level bookmark then make the previous one its child.
my $sect = $doc->newBookmark( 'Section 1' );
$sect->add( $box->getBookmark( ) );
$doc->writePDF( 'sample5-3.pdf' );
```

- **Border and background**

  By default the text box has a transparent background and no borders. The text is shown at full opacity in normal mode – the pixels do not blend into those already rendered. You can define border color, line width, and dash pattern for the text box, as well as background attribute. Three different types of background filling schemes can be used:

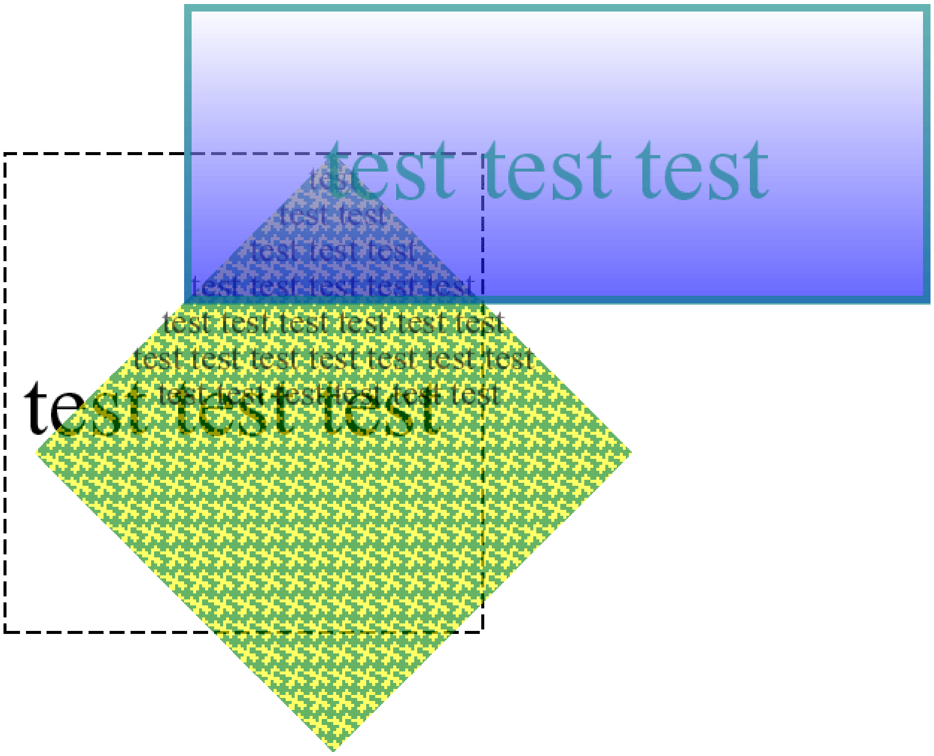
  1. Solid color: Use a color and assign it to the attribute `BgColor`;
  2. `Texture` pattern: A valid `PDFTexture` object that defines a pattern;
  3. `Shading` pattern: A valid `PDFShading` object that defines a shading (gradient).

  In addition, for Acrobat 5.x or higher, opacity (Opacity) and blending mode (BlendMode) can be set for each text box, which is regarded as separate layer. The page is considered a stack of such layers overlapped with each other, in a predefined order, over the page media. In PDF Everywhere, each individual `TextBox` becomes an independent layer; new instance is placed above existing ones.

  In Example 5-4, three text boxes are created, each with different background and border settings. The details of initiating and using texture and shading patterns will be described later. The output of the example is shown on the next page.

Example 5-4. Setting the border, background, and transparency of text boxes.

```
use PDF;
my $doc = new PDFDoc( { TextJustify => 'Center',
   VerticalAlign => 'Middle' } );      # Default alignment settings
$doc->newPage( );
my $str = "test test test ";
# Layer 1
$doc->newTextBox( new Rect( 40, 40, 200, 200 ), $str, {
   BorderDash => 'Dashed',
   FontSize => 30 } );
# Layer 2
$doc->newTextBox( new Poly( 50, 100, 150, 200, 250, 100, 150, 0 ),
   $str x 10, {
   Opacity => 60,
   Texture => new PDFTexture( 'daisies', { FgColor => 'Green',
      BgColor => 'yellow' } ) } );
# Layer 3
$doc->newTextBox( new Rect( 100, 150, 350, 250 ), $str, {
   BorderColor => 'Teal',
   BorderWidth => 5,
   FontSize => 32,
   FontColor => 'Teal',
   Opacity => 60,
   BlendMode => 'Luminocity',
   Shading => new PDFShading( 'Linear', { FromColor => 'blue',
      ToColor => 'white' } ) } );
$doc->writePDF( 'sample5-3.pdf' );
```


test test test
test
test test
test test test
test test test test test
test test test test test test
test test test test test test test
test test test test test test
test test test

Please also check Chapter 3 for examples about adding text boxes to page template.

- **Inline text style change**

Text style can be changed with some special markers in the form of “`^x`”, where `x` can be `a`, `b`, `c`, `e`, `f`, `h`, `i`, `l`, `r`, `s`, `u`, etc. (case sensitive), with optional attribute values in the form of “`:name1=value1&name=value2`”, for example “`^f:Face=Arial&Size=10`”. The available markers and the optional attributes for some markers are listed in Table 5-3 (these markers are `f`, `a`, `H`, `S`, and `U`). For `r`, `e`, `h`, `s`, and `u`, annotations of the corresponding type are automatically created. The starting and ending markers should not be mixed. These markers are a much simplified solution to overcoming the drawback of the constant text style within a text box; no new markers will be introduced in the future.

Table 5-3. Markers used for inline style change in a text box.

Marks	Explanation
`b`	Sets the subsequent text to bold
`i`	Sets the subsequent text to italic
`bi` *or* `ib`	Sets the subsequent text to bold-italic
`r, e`	Creates a hyperlink for the subsequent email address or URL
`u`	Use underline for the subsequent section (may run across multiple lines).
`f`	Changes the font style of subsequent text to denoted `Size`, `Face`, `Color`, and `Leading`. Not all parameters are need in all situations.
`a`	Changes the alignment of subsequent text. Optional attributes include `Align` and `Vert`, denoting horizontal and vertical alignments.
`h`	Begins a highlighted section.
`s`	Begins a strikeout section.
`n, p`	Begin a new line of a new paragraph.
`B`	Marks the end point of the preceding ‘`b`’ marker
`I`	Marks the end point of the preceding ‘`i`’ marker
`BI` *or* `IB`	Marks the end point of the preceding ‘`bi`’ or ‘`ib`’ marker
`U`	Marks the end of underlined section. Can be followed by parameters `Width`, `Color`, and `Dash`.
`R, E`	Marks the end of the email address or URL to be hyperlinked.
`F`	Restores text style
`A`	Restores text alignment since next paragraph.
`L`	Insert a line annotation. Retained for historical reason, but no longer recommended to use.
`H`	Ends the highlighted section. Attributes include `Color`.
`S`	Ends the strikeout section. Attributes include `Color` and `Width`.

Table 5-4. Optional attributes for some of the inline markers.

Parameter	Used By	Explanation
`Color`	`f, U, H, S`	Font color or line color
`Face`	`f`	Font name.
`Size`	`f`	Font size in points.
`L`	`f`	Text leading (line height) in points.
`Align`	`a`	Horizontal alignment. Must be **`Left`**, **`Center`**, **`Right`**, or **`Uniform`**. When text direction is not horizontal, this specifies the alignment along the text direction.
`Vert`	`a`	Vertical alignment. Must be **`Top`**, **`Middle`**, or **`Bottom`**.
`Dash`	`U`	Line dash pattern, can be **`Solid`**, **`Dotted`**, **`Dashed`**, or **`DashDot`**.
`Width`	`U, S`	Line width in points.

See Example 5-1 for the use of such markers.

If the PDF document is created with attributes **`FindBIU`**, **`FindHyperlink`**, and/or **`ReplaceEntities`** set, corresponding activities will be done. Words enclosed by a pair of `*`, `=`, and `_` will be replaced with bold, italic, and underlined text via inline markers `b`…`B`, `i`…`I`, and `u`…`U`. Email address and hyperlinks will be automatically turned into clickable hyperlinks via the inline markers `r`…`R` and `e`…`E`.

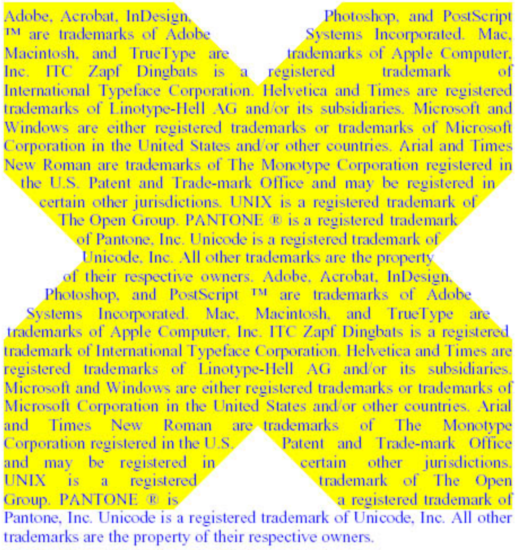
- **Polygon text box**

⚠ The text box can take a polygon as the hosting shape. The interior of the polygon is ruled by line height of the text. If a line passes one or more vertices of the polygon, the program would have trouble determining where the interior is. Therefore, please try to avoid this by using non-integer vertex coordinates.

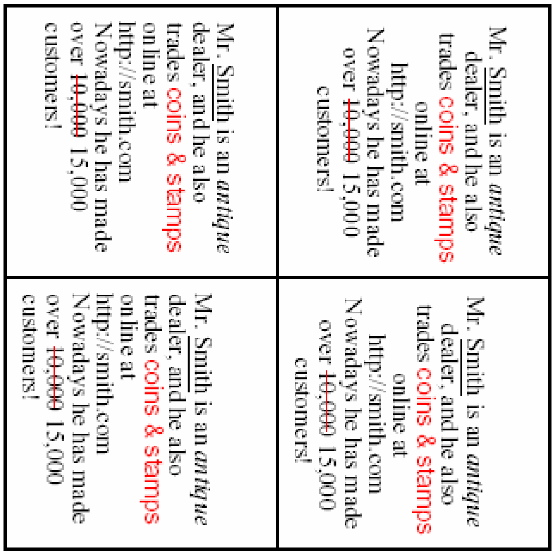
Example 5-5 shows an example of using a polygon. If the polygon can not host all text, the text will flow out of the shape but within the width of the polygon.

Example 5-5. Text box with a polygon container.

```
use PDF;
my $text = q{Adobe, Acrobat &trademark; ...}; # Omitted
my $doc = new PDFDoc( { ReplaceEntities => 1 } );
$doc->newPage( );
$doc->newTextBox( new Poly( 100, 100, 200, 100, 250, 150, 300, 100,
400, 100, 400, 200, 350, 250, 400, 300, 400, 400, 300, 400, 250,
350, 200, 400, 100, 400, 100, 300, 150, 250, 100, 200 ),
    $text, {FontSize=>10, TextJustify=>'Uniform', Color=>'blue',
    Leading=>11, BgColor=>'Yellow'} );
$doc->writePDF( 'sample5-5.pdf' );
```



Adobe, Acrobat, InDesign, Photoshop, and PostScript ™ are trademarks of Adobe Systems Incorporated. Mac, Macintosh, and TrueType are trademarks of Apple Computer, Inc. ITC Zapf Dingbats is a registered trademark of International Typeface Corporation. Helvetica and Times are registered trademarks of Linotype-Hell AG and/or its subsidiaries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Arial and Times New Roman are trademarks of The Monotype Corporation registered in the U.S. Patent and Trade-mark Office and may be registered in certain other jurisdictions. UNIX is a registered trademark of The Open Group. PANTONE ® is a registered trademark of Pantone, Inc. Unicode is a registered trademark of Unicode, Inc. All other trademarks are the property of their respective owners. Adobe, Acrobat, InDesign, Photoshop, and PostScript ™ are trademarks of Adobe Systems Incorporated. Mac, Macintosh, and TrueType are trademarks of Apple Computer, Inc. ITC Zapf Dingbats is a registered trademark of International Typeface Corporation. Helvetica and Times are registered trademarks of Linotype-Hell AG and/or its subsidiaries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Arial and Times New Roman are trademarks of The Monotype Corporation registered in the U.S. Patent and Trade-mark Office and may be registered in certain other jurisdictions. UNIX is a registered trademark of The Open Group. PANTONE ® is a registered trademark of Pantone, Inc. Unicode is a registered trademark of Unicode, Inc. All other trademarks are the property of their respective owners.



Mr. Smith is an *antique* dealer, and he also trades coins & stamps online at http://smith.com Nowadays he has made over ~~10,000~~ 15,000 customers!

Mr. Smith is an *antique* dealer, and he also trades coins & stamps online at http://smith.com Nowadays he has made over ~~10,000~~ 15,000 customers!

Mr. Smith is an *antique* dealer, and he also trades coins & stamps online at http://smith.com Nowadays he has made over ~~10,000~~ 15,000 customers!

Mr. Smith is an *antique* dealer, and he also trades coins & stamps online at http://smith.com Nowadays he has made over ~~10,000~~ 15,000 customers!

Example 5-6 shows the automatic creation of underlines and strikeouts. It creates four pages of 3 × 4 grids filled with text with different alignment and text direction. In any case, this program is able to align and rotate the annotations correctly.

Please note the appearance of highlight annotations look ugly in Acrobat 5, although it was fine in Acrobat 4.

Example 5-6. Automatically annotations in any text direction and alignment.

```
use PDF; my $doc = new PDFDoc( { FindHyperlink => 1 } );
my @horiz = qw(Left Center Right Uniform);
my @vert = qw(Top Middle Bottom);
my $text = "Mr. ^u Smith ^U is an ^i antique ^I dealer, and he also
trades ^f:Face=Arial&Color=red coins & stamps ^F online at
http://smith.com Nowadays he has made over ^s 10,000 ^S:Color=red
15,000 customers!";
for my $tdir ( 0..3 ){
   $doc->newPage( );
   my $g = $doc->newGraph( );
   $g->moveTo( 100, 300 );
   my @cells = $g->drawGrid( 400, 300, 0, {XGrids=>4, YGrids=>3} );
   for $i ( 0..2 ){
      for $j ( 0..3 ) {
         $doc->newTextBox( $cells[$i*4 + $j], $text, {
            FontFace=>'Times-Roman', FontSize=>9, TextDir=>$tdir,
            ContentPadding=>'2mm', TextJustify=>$horiz[$j],
            VerticalAlign=>$vert[$i] } );
      }
   }
}
$doc->writePDF( 'sample5-6.pdf' );
```

- **Special characters**

Special characters such as ® and ™ cannot be directly typed. You need to use a form similar to HTML entities, for example, `&register;` and `&trademark;` for these two symbols. The recognizable entity names are listed in Table 5-5. They will be replaced in the text of a text box, if the document's `ReplaceEntities` attribute is set.

Please note they are PDF entity names therefore *not* exactly the same as those defined in HTML, and they are case-significant. The semicolon must be typed, or the program won't convert it. In the rare case when you want to avoid an entity name be converted, just place a slash in front of "`&`", as in "`\&Euro;`".

Table 5-5. Recognizable entities for special characters in a text box.

Entity	Char	&ae;	æ	&multiple;	×
&AE;	Æ	&aacute;	á	&ntilde;	ñ
&Aacute;	Á	&acircumflex;	â	&oe;	œ
&Acircumflex;	Â	&adieresis;	ä	&ograve;	ò
&Adieresis;	Ä	&agrave;	à	&onehalf;	½
&Agrave;	À	&ampersand;	&	&onequarter	¼
&Aring;	Å	&aring;	å	&oslash;	ø
&Atilde;	Ã	&atilde;	ã	&otilde;	õ
&Ccedilla;	Ç	&ccedilla;	ç	&perthousand;	‰
&Eacute;	É	&cent;	¢	&plusminus;	±
&Ecircumflex;	Ê	&copyright;	©	&questiondown;	¿
&Edieresis;	Ë	&degree;	°	&quotedblleft;	“
&Egrave;	È	&dagger;	†	&quotedblright;	”
&Eth;	Đ	&daggerdbl;	‡	&quoteleft;	‘
&Euro;	€	&dotlessi;	ı	&quoteright;	’
&Oacute;	Ó	&eacute;	é	&registered;	®
&Ocircumflex;	Ô	&ecircumflex;	ê	&ring;	˚
&Odieresis;	Ö	&edieresis;	ë	&scaron;	š
&OE;	Œ	&egrave;	è	&section;	§
&Ograve;	Ò	&ellipse;	…	&sterling;	£
&Oslash;	Ø	&emdash;	—	&thorn;	þ
&Otilde;	Õ	&eth;	ð	&threequarters;	¾
&Scaron;	Š	&germandbls;	ß	&trademark;	™
&Thorn;	Þ	&guillemotleft	«	&uacute;	ú
&Uacute;	Ú	&guillemotright	»	&ucircumflex;	û
&Ucircumflex;	Û	&iacute;	í	&udieresis;	ü
&Udieresis;	Ü	&icircumflex;	î	&ugrave;	ù
&Ugrave;	Ù	&idieresis;	ï	&yacute;	ý
&Yacute;	Ý	&igrave;	ì	&ydieresis;	ÿ
&Ydieresis;	Ÿ	&logicalnot;	¬	&yen;	¥
&Zcaron;	Ž	&lslash;	ł	&zcaron;	ž

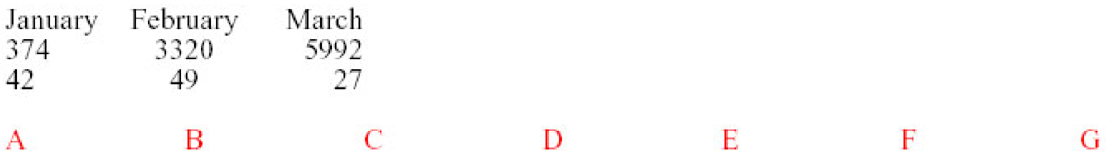
- **Tab stop**

A special marker `^t` in original text is treated as a tab, and the subsequent text is moved to next tab position (the default interval for tab positions is 1 inch, to change, set the attribute `TabPosition` when creating the text box). This behavior is meaningful *only* when the horizontal alignment (`TextJusify`) is set to `Left`.

The marker ^t can have two attributes: `Align`, to specify the alignment subsequent text (until the first line-break) at the next tab position, and `Pos`, to designate the position of next tab. In the next example, alignment is set for some tabs to create columns of data.

Example 5-7. Tabs and new lines in a text box.

```
use PDF;
my $doc = new PDFDoc( );
my $text = q{
January ^t:Align=Center February ^t:Align=Right March ^n
374 ^t:Align=Center 3320 ^t:Align=Right 5992 ^n
42 ^t:Align=Center 49 ^t:Align=Right 27 ^n
^n ^f:Color=Red
A ^t B ^t C ^t D ^t E ^t F ^t G ^F ^n
};
$doc->newPage( );
my $g = $doc->newGraph( );
$doc->newTextBox( new Rect( 100, 100, 400, 500 ), $text );
$doc->writePDF( 'sample5-5.pdf' );
```



```
January   February     March
374         3320        5992
42           49           27

A           B           C           D           E           F           G
```

- **New line and new paragraph**

To insert a new line or being a new paragraph within text, use inline markers `^n` and `^p`.

In original input text, two or more consecutive line-breaks begins a new paragraph, which single line-breaks are just discarded. In addition, multiple instances of white spaces are treated as a space.

- **Table-based layout scheme**

In XML, you can use the attributes listed in Table 3-2 for the `<Text>` tag to position and size the text box. To the similar in Perl, the function `PDFDoc::getCell` would be needed, as well as the rectangle-manipulation functions defined in class Rect and introduced in §4.1.

Same also applies to preformatted text box, described below.

5.5 Preformatted Text Box

The previous text box cannot display preformatted text such as program code. To do so, you can use the `PreformText` object. This object requires public-domain modules `Text::Tabs` and `Text::Wrap`.

```
PDFDoc::newPreformText( $shape: Rect, $text: String, $attr: HASH) :
    PreformText
```

The font face is always `Courier`, and the horizontal alignment is always `Left`. The container can only be a rectangle. Most of the properties listed in Table 5-2 are applicable here, with the exception of `FontFace`, `CharSpacing`, `WordSpacing`, and all text alignment settings other than `VerticalAlign` and `ContentPadding`. In addition, the following three properties are defined for the treatment of preformatted text.

Table 5-6. Additional properties for preformatted text boxes.

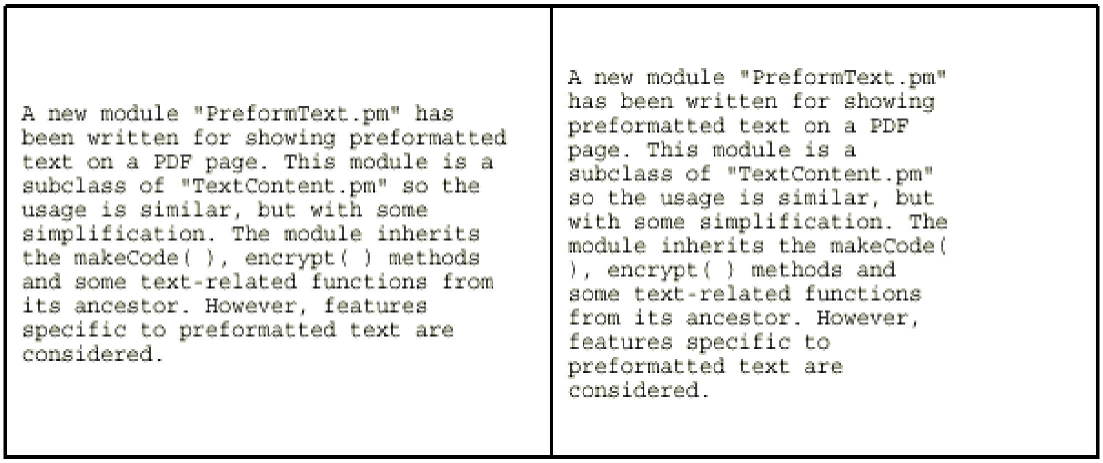
Key	Type	Explanation	Default
`AutoWrap`	Bool	Text is auto-wrapped at the container rectangle boundary or at the specified number of columns (see next row).	`0`
`Columns`	Integer	The number of characters per line. Used for auto wrapping.	`Undef`
`TabStop`	Integer	The maximum number of spaces to be extended from a tab.	`8`

The constructor automatically wraps the text and extends the text. `Columns` value is set to the maximum number of characters allowed if it is not passed to the constructor. In the following example, you can see the effect of this parameter.

Example 5-8. Different wrapping method for preformatted text boxes.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( );
$oGraph = $doc->newGraph ( );
$oGraph->moveTo( 100, 400 );
$text = "blah blah blah ..."; # Replace with actual text here
@cells = $oGraph->drawGrid( 400, 150, 0, {XGrids=>2, YGrids=>1} );
$doc->newPreformText( $cells[0], $text, {
   AutoWrap => 1,
   FontSize => 8, ContentPadding => 3, VerticalAlign => 'Middle'} );
new PreformText( $cells[1], $text, {
   AutoWrap => 1,
   Columns => 30,
   FontSize => 8, ContentPadding => 3, VerticalAlign =>'Middle' } );
$doc->writePDF( 'sample5-8.pdf' );
```

The output of this example is shown below.



```
A new module "PreformText.pm" has
been written for showing preformatted
text on a PDF page. This module is a
subclass of "TextContent.pm" so the
usage is similar, but with some
simplification. The module inherits
the makeCode( ), encrypt( ) methods
and some text-related functions from
its ancestor. However, features
specific to preformatted text are
considered.
```

```
A new module "PreformText.pm"
has been written for showing
preformatted text on a PDF
page. This module is a
subclass of "TextContent.pm"
so the usage is similar, but
with some simplification. The
module inherits the makeCode(
), encrypt( ) methods and
some text-related functions
from its ancestor. However,
features specific to
preformatted text are
considered.
```

In XML, the tag is also `Text` but there must be an attribute called `Preform` with the value 1:

```
<Text
   Rect="x1,y1,x2,y2" <!-- A list of four dimension values -->
   Preform="1"
   Name="string"
   FontFace="fontname"
   <!-- More attributes can be inserted here -->
><![CDATA[<!-- The text to be displayed -->]]></Text>
```

Add the following line to the above example to see what it produce:

```
$doc->exportXML( );
```

5.6 Some Clarifications

- **Free-text Annotation**

Such annotations appear as editable text boxes that can be dragged and resized. These interactive annotations are topics of future chapters and should be differed from the floating text boxes we've been discussing above, which are part of page content.

- **Embedded Font**

When a font is embedded, an object of the type `EmbeddedFont` is created. This is extended from the type `PDFStream`.

The font file is embedded as-is. For a Type-1 PFB font, the font file is analyzed to calculate the byte length values of the leading and trailing ASCII portion and the binary portion that defines character shapes.

Chapter 6: Vector Graphics

In a PDF file, there is no such a thing as a specific graphic content stream. Graphic and text operations are mixed together in a language similar to PostScript. In this toolkit, the concepts for graphics and text are separated. One or more graphics layer can be created to host vector graphics and raster images, and blocks of text can be hosted in the text boxes described in previous chapter, while short, rich-effected text arts can be shown within the graphics layers. Such a layer is represented by a `GraphContent` object, which is created by the method `newGraph` defined in class `PDFDoc` representing the PDF document to be created.

This toolkit provides a relatively large and definitely powerful collection of convenient functions to vector graphics operations, much beyond simply interfacing the primitive operators available in PDF language. These functions are classified into five categories:

- **General** – Provide the interface for primitive operators.
- **Positioning** – Move the current position, a reference point that can be regarded as a virtual cursor. The program keeps track of this position to ensure the viewer application will render the graphics in the same way it was created.
- **Setting** – Set graphic properties such as color, line width, fill pattern, etc.
- **Drawing** – Include basic line, rectangle and curves, plus circle, arc, oval, polygon, polyline, grid, etc.
- **Utility** - Provide utilities to fill shapes with gradient or texture, to show an image and perform transformations, to show text with leading layout facilities,

To render a page, the viewer application maintains a stack to save and restore workspace status (graphic state). Therefore, settings are reset to default values when entering a new graphic state but are recalled when exiting from that state. This fact should be kept in mind.

In addition, the application would maintain a clipping path, with default coverage of the entire page. There is only one active path and replaced ones are discarded. Clipping path is not affected by the graphic state's changes, and its existence only defines the regions that should be rendered while the path itself is invisible. In summary, graphic state and clipping path are independently maintained. It should be noted that the building of complex paths and repeated path changes can make the resultant PDF file subject to logical errors due to incorrect arrangement or missing of operators. Also please keep in mind that page origin and the origin of a `GraphContent` object is always the bottom-left corner.

6.1 Constructing a Graphics Layer

To create such a graphics layer, call the following function:

```
PDFDoc::newGraph( ) : GraphContent
PDFDoc::newGraph( $shape: Rect ) : GraphContent
PDFDoc::newGraph( $attr: HASH) : GraphContent
PDFDoc::newGraph( $shape: Rect, $attr: HASH) : GraphContent
PDFDoc::newGraph( $obj: GraphContent ) : GraphContent
```

That is, the parameters for the constructor could be one of the following:

1. Nothing: In this case, the layer is the same size as the entire art box of the page.
2. Rectangle: The rectangle defines the position and size of the layer.
3. Hash reference: The hash contains key/value pairs for the properties of the layer.
4. Rectangle and hash reference: Cases 2 and 3 combined.
5. Another graphics layer: The content of that layer is copied. Beyond this point, the two layers are independent to each other.

▪ Properties of the graphics layer

The following attributes can be used in the hash reference mentioned above. They can also be used in XML creation.

Table 6-1. Properties for a graphics layer.

Key	Type	Explanation	Default
`Rect`	Rect	Layer's box in the page coordinate system	
`Name`	String	A unique name for this layer.	
`IsTemplate`	Bool	If true, the object will be added to document's internal template collection.	
`InternalUse`	Bool	Do not create PDF code. (Not used in XML.)	
`Opacity`	Integer	Between 0 to 100	`100`
`BlendMode`	String	Blending mode of this text box as a layer.	`Normal`
`Bookmark`	String	Bookmark title for this text box object.	`Undef`
`ZIndex`	Integer	Stack sequence on the host page.	`0`

Below we will describe the various graphics-drawing functions. Most of the functions need coordinate values, which are measured with respect to the origin of the layer, rather than that of the page.

A layer cannot be moved or transformed. The coordinates are converted to values measured in page coordinate system.

To create the graphics layer in an XML document, use the tag `<Graph>` and put XML instructions for graphical operations between the opening and closing tags:

```
<Graph
   Rect="x1,y1,x2,y2" <!-- A list of four dimension values -->
   Name="string" ZIndex="int" <!-- More attributes here -->
>
   <!-- Put XML tags for graphics operations here -->
</Graph>
```

In XML, you can also use the attributes listed in Table 3-2 for the <Graph> tag to position and size the graphics layer.

6.2 General a Graphics Operations

These methods are simple interfaces to the corresponding PDF graphic state operators, with an exception that newPath also begins a new graphic state, thus drawing settings such as line width, fill color, dash pattern, etc., are reset. The other method, closePath, is what PDF defined as "new path" operator. The equivalency is shown in the table below. For detailed description of these operators, please refer to the PDF Reference Manual available on Adobe's technical support site.

Table 6-2. Graphic state operations.

Function	Parameter	PDF operator	Explanation
saveGState		q	Save graphic state
restoreGState		Q	Restore graphic state
stroke		S	Stroke path
fill	$eof	f, f*	Fill path
strokeFill	$eof	b, b*	Stroke and fill path
closeSubPath		H	Close a sub path
closePath		N	Close a path
intersect	$eof	W, W*	Intersect with existing clipping path

Parameter $eof is an optional Boolean value. If true, then even-odd filling rule is used. If false (0), non-zero winding rule is used.

The rule of thumb is to always ensure that intersect is called before any one of the functions stroke, fill, and strokeFill. The path is built between these two calls by using moveTo or setPos first (see next section), followed by drawing functions. If multiple clipping paths are needed, use the combination newPath and intersect to begin a new path. If there is only one path, intersect can be omitted.

- **XML tag**

A single tag, Exec, is used to carry out all of these function calls, the corresponding attribute is Cmd and Eof, where Cmd can be SaveGState, RestoreGState, Stroke, Fill, StrokeFill, CloseSubPath, ClosePath, NewPath, and Intersect, and Eof should be 1 if present (meaningful for Fill, StrokeFill, and Intersect).

6.3 Positioning Operations

These functions set or move current drawing position, which could be thought as a virtual paintbrush. To set or get current drawing position, call the following two methods. Method setPos starts a new graphic state and begins a new path at given point. Method getPos returns a pair of float numbers as the coordinates.

```
GraphContent::setPos( $x: size, $y: size )
GraphContent::getPos( ): size[2]
```

However, `moveTo` is more frequently used, which moves the current position and continues to append shapes to current path. You may also need the `moveBy` to move it by offset from current position.

```
GraphContent::moveTo( $x: size, $y: size )
GraphContent::moveBy( $offsetX: size, $offsetY: size )
```

The XML tags for these functions are:

```
<Set X="float" Y="float"/>
<MoveTo X="float" Y="float"/>
```

6.4 Parameter-setting Operations

- **Stroke**

These functions set the stroke (line) styles – width, cap style, join style, miter limit, and dash pattern – for subsequence drawings until a new graphic state is created or another setting is used via the same functions.

```
GraphContent::setLineWidth( $width: size )
GraphContent::setLineCap( $cap: string | integer )
GraphContent::setLineJoin( $join: string | integer )
GraphContent::setMiterLimit( $limit: size )
GraphContent::setDash( $dash: string )
```

The functions `setLineCap` and `setLineJoin` take a parameter that can be a predefined name as a string or corresponding integer value (0, 1, 2), as illustrated in Figure 6-1. Round and square caps extends the covered area. Miter limit cut out the corners that are extended too far (angle too sharp).

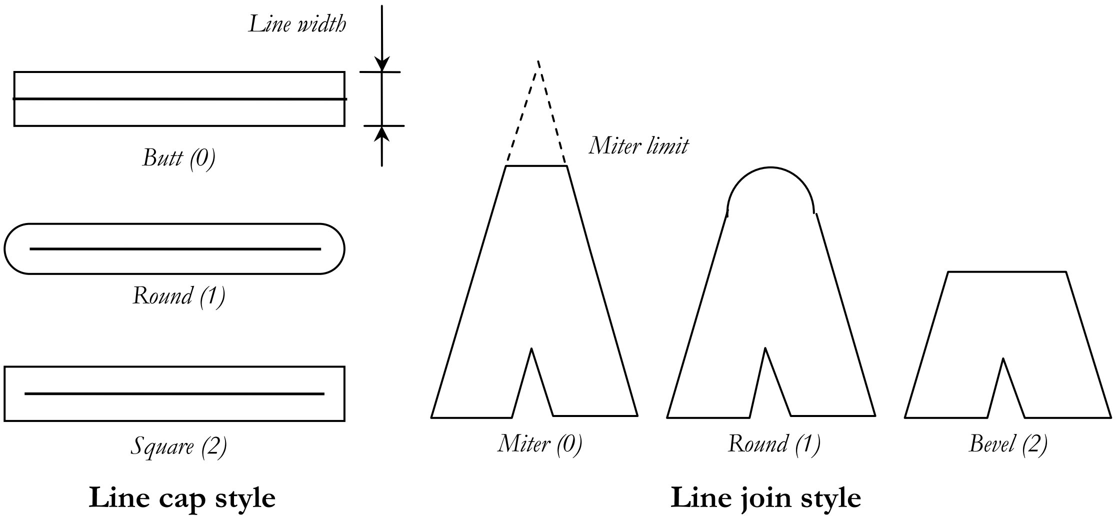
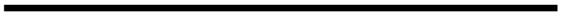
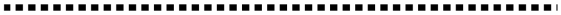
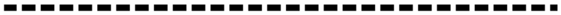
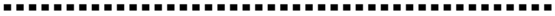
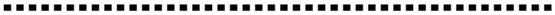
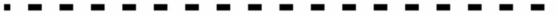
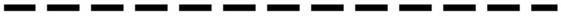
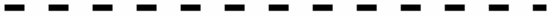
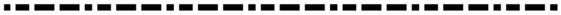
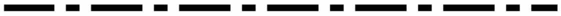


Figure 6-1. Line cap and join styles.

A line can be drawn in a customizable dash pattern. The parameter passed to `setDash` should be in one of the following three acceptable forms:

1. Standard PDF format "[*a b*] *c*", such as "[2 3] 4", where *a* and *b* specify the length of solid and open portions (units is dependent on device), and *c* specifies the phase (starting offset).
2. Comprised of 0's and 1's such as '011000' (visually depicted) -- 1 means a pixel would be drawn and 0 means gap. This string should contain one non-repeating unit (here '0'), followed by a repeating unit (here '11000'), in similar sense that you express 3/7 as $0.\dot{4}2857\dot{1}$.
3. One of the predefined dash pattern names: `Solid`, `Dotted`, `Dashed`, `DashDot`. Illustrations of dash patterns are listed in Table 6-3.

Table 6-3. Illustration of line dash patterns.

Pattern	PDF Standard	PDF Everywhere
	`[] 0`	`1 (Solid)`
	`[1] 0`	`10 (Dotted)`
	`[2 1] 0`	`110`
	`[1] 1`	`010`
	`[1] 1`	`101 (same as 010)`
	`[2 3] 1`	`100011`
	`[5 2] 0`	`1111100 (Dashed)`
	`[3 4] 5`	`001110000`
	`[1 1 2 1 3 1 3 1] 0`	`1011011101110`
	`[8 2 2 2] 0`	`11111111001100 (DashDot)`

The XML tag for these functions is <set>, which can take one or many attributes in the following list:

```
<Set
    Dash="Solid|Dashed|Dotted|DashDot"
    LineCap="0|1|2|Butt|Round|Square"
    LineJoin="0|1|2|Miter|Round|Bevel"
    LineWidth="size"
    MiterLimit="size"
/>
```

Note that the **Set** tag is also used to set current position, colors and textures.

Color

Both foreground (stroke) and background (fill) color can be set, using RGB (default), CMYK, or Gray color space and any color name or value (RGB, CMYK, Gray).

```
GraphContent::setColor( $color: color, [ $ground: Bool, $space
    string ] )
```

The third argument must have a string value of either `RGB`, `CMYL`, or `Gray` (letter case sensitive); if it is omitted, current color space for the document is used. The second argument determines whether the color is set for fill or stroke, with a default value of false (0).

To change color space for subsequent `setColor` calls, use this function:

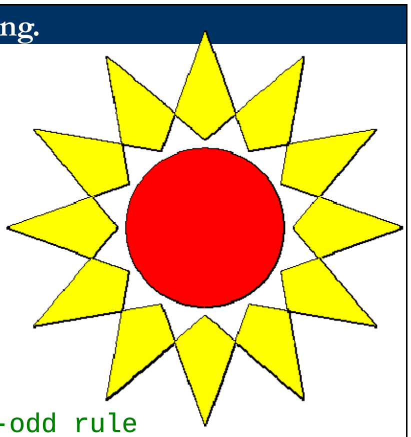
```
GraphContent::setColorSpace( $space: string )
```

The XML tag for these functions is also `<Set>`:

```
<Set
    Color="color"
    Ground="0|1"
    ColorSpace="RGB|CMYK|Gray"
/>
```

In the following example, we draw a sunshine design using two intersecting hexagons and a circle. What's tricky here is that we need two clipping paths: one for the flares and one for the pie. However, there can be only one path for each page at any time, so we call `newPath` between the two sets of drawing.

Example 6-1. Using colors in a vector graphics drawing.



```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( );
$oGraph = $doc->newGraph( );
$oGraph->setLineWidth( 0.5 );
$oGraph->setColor( 'yellow', 1 );
$oGraph->intersect( 1 );
$oGraph->moveTo( 300, 300 );
$oGraph->drawPolygon( 100, 6, 0, 70, 3 );
$oGraph->moveTo( 300, 300 );
$oGraph->drawPolygon( 100, 6, 30, 70, 3 );
$oGraph->strokeFill( 1 );  # Filling by even-odd rule
$oGraph->intersect( );     # Reset intersecting mode
$oGraph->newPath( );       # Begin a new path; discard the old one
$oGraph->setLineWidth( 0.5 );
$oGraph->setColor( 'red', 1 );
$oGraph->moveTo( 300, 300 );
$oGraph->drawCircle( 40, 2 );
$doc->writePDF( 'sample6-1.pdf' );
```

The equivalent XML code for this program is:

```
<?xml version="1.0" standalone="yes"?>
<PDF>
<Page Name="N000" Width="612" Height="792">
   <Graph Rect="0, 0, 612, 792" Name="N001">
      <Set LineWidth="0.5" />
      <Set Color="yellow" Ground="1" ColorSpace="" />
      <Exec Cmd="Intersect" Eof="1" />
      <MoveTo X="300" Y="300" />
      <DrawPolygon Radius="100" Sides="6" Rotation="0" Tilt="70"
Fill="3" />
      <MoveTo X="300" Y="300" />
      <DrawPolygon Radius="100" Sides="6" Rotation="30" Tilt="70"
Fill="3" />
      <Exec Cmd="StrokeFill" Eof="1" />
      <Exec Cmd="Intersect" Eof="0" />
      <Exec Cmd="NewPath" />
      <Set LineWidth="0.5" />
      <Set Color="red" Ground="1" ColorSpace="" />
      <MoveTo X="300" Y="300" />
      <DrawCircle Radius="40" Fill="2" />
   </Graph>
</Page>
</PDF>
```

- **Texture pattern**

A texture is like a special color and it can be used in stroke or fill operations, but with a special color space. A texture can be colored or uncolored. In the latter case, a normal color is needed to render the texture in desired color. The function and XML tag for this operation are:

```
GraphContent::setTexture( $texture: string, [ $ground: Bool, $color:
    color ] )
```

```
<Set
    Texture="name"
    Ground="0|1"
    Color="color"
/>
```

The value for the attribute `Texture` must be the name of an already-defined `PDFTexture` object. The parameter `Ground` determines whether the texture is to be applied to ground (fill color) or stroke. `Color` is needed for uncolored textures only.

For more information on texture, please later sections in this chapter.

6.5 Drawing Functions

- **Lines**

Line segments, poly-lines, and Bézier curves can be drawn. An arrow head can be added to the end of a line segment, whose shape is relatively adjustable. These shapes and their attributes are illustrated in Figure 6-2.

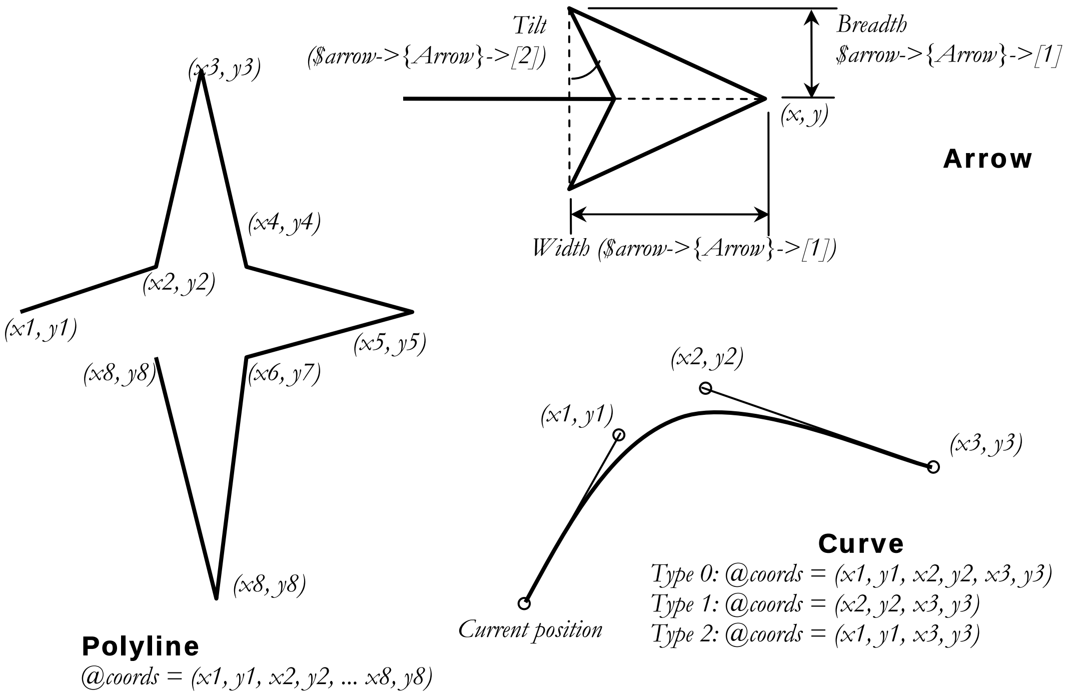


Figure 6-2. Parameters for line-drawing functions.

The function `lineTo`, which draws a line from current position to designated point, can take various arguments (only the coordinates of the new position are required). When an arrow head is needed, use a hash reference. This hash table must contain a key called `Arrow`, which points to a 3-element array, as described in Table 6-4.

```
GraphContent::lineTo( $x: size, $y: size, [ $attr: HASH ] )
GraphContent::lineTo( $x: size, $y: size, [ $width: size, $color:
    color, $pathonly: Bool ] )
```

```
<LineTo
    X="size" Y="size"
    Width="size"
    Color="color"
    PathOnly="0 | 1"
    Arrow="width, breath, tilt"  <!-- Three positive numbers -->
/>
```

Table 6-4. Attributes of an arrow head at the end of a line segment.

Key	Meaning
`Arrow`	Reference to a three-element array, defining the **width**, **breadth**, and **tilt angle** in that order. Length values are in points only. Angle is expressed as a degree, measured inwards from vertical direction.
`Width`	Line width
`Color`	Line color (not fill color)

The last (but optional) argument to the function call determines whether the line is to be drawn or just added to current clipping path.

If an arrow is defined, then the line cannot be used as part of a clipping path. In addition, the program calculates the slope of the line to before determining the profile of the arrowhead, then retracts the destination point to the root of the arrowhead to avoid showing a butt for wide lines. As a result, the current position is moved to an undefined point.

A similar function, `connectTo`, is used to build clipping path without drawing the lines (there is no corresponding XML tag):

```
GraphContent::connectTo( $x: size, $y: size )
```

Lastly, a function `drawPolyLine` is used to draw consecutive line segments over designated points:

```
GraphContent::drawPolyLine( $x: size, $y: size, [ $fill: int ] )
```

```
<PolyLine Coords="x1, x2, y1, y2, ..." Fill="0|1|2|3"/>
```

The argument `Fill` affects the behavior of the vector graphics rendering, as shown in Table 6-5. Many functions defined in this class will also use this parameter.

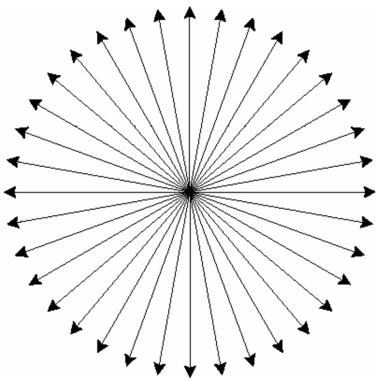
Table 6-5. The meaning of different values for "Fill".

Value	Meaning
0	Draw strokes only
1	Fill in shape only
2	Render both stroke and fill
3	Add to clipping path only (for being rendered later)

Now we draw a starburst lines with arrowheads, which are filled with black color.

Example 6-2. Drawing lines with arrowheads.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( );
my $oGraph = $doc->newGraph( );
$oGraph->setLineWidth( 0.5 );
my $arrow = { 'Arrow' => [ 6, 3, 20 ] };
for my $i ( 1..36 ){
   my $angle = ( $i * 10 ) * 3.142 / 180;
   $oGraph->moveTo( 300, 300 );
   $oGraph->lineTo( 300 + 100 * cos( $angle ),
      300 + 100 * sin( $angle ), $arrow );
}
$doc->writePDF( 'sample6-2.pdf' );
```



- **Bézier curve**

A type 0 Bézier curve is defined with two control points and a destination point. For a type 1 curve, the first control point coincide with the current position, so only the coordinates of the last control point and the destination point are needed. For a type 2 curve, the second control point coincides with the destination point (cf. illustration in Figure 6-2).

```
GraphContent::curveTo( $type: int, @coords: size[] )
```

```
<CurveTo
    Type="0|1|2"
    Coords="x1, y1, x2, y2, ..."
/>
```

The arguments to the function call are a type value (0, 1, 2) followed by a list of coordinates.

- **Rectangle**

The following functions draw a rectangle from *current position* to a designated point with specified width and height (`drawRect` and `drawRoundedRect`) or coordinates (`drawRectTo` and `drawRoundedRectTo`) or using the data of an existing `Rect` object (`drawRectAt` and `drawRoundedRectAt`). The rectangle can have rounded corners, and can be centered at current position, controlled by the optional parameters `Centered`. Note that for the `Rect` object, if used, is specified in absolute coordinate system, i.e. the entire `page`).

```
GraphContent::drawRect( $width: size, $height: size, [ $fill: int,
    $centered: Bool ] )
GraphContent::drawRectAt( $rect: Rect, [ $fill: int ] )
GraphContent::drawRectTo( $x: size, $y: size, [ $fill: int ] )
GraphContent::drawRoundedRect( $width: size, $height: size, $round:
    size, [ $fill: int, $centered: Bool, $reverse: Bool] )
GraphContent::drawRoundedRectAt( $rect: Rect, $round: size, [ $fill:
    int, $reverse: Bool ] )
GraphContent::drawRoundedRectTo( $x: size, $y: size, $round: size,
    [ $fill: int, $reverse: Bool ] )
```

```
<DrawRect
    Width="size"
    Height="size"
    Round="size"      <!-- Radius of the rounded corner -->
    Centered="0|1"
    Fill="0|1|2|3"
    Reverse="0|1"     <!-- Clockwise or counterclockwise? -->
/>
```

Regular rectangle (drawn with PDF operator `re`) has no direction. By contrast, a rounded rectangle is the concatenation of four line segments and four quarter circles. Figure 6-3 shows the illustration of rectangles and polygons (next segment).

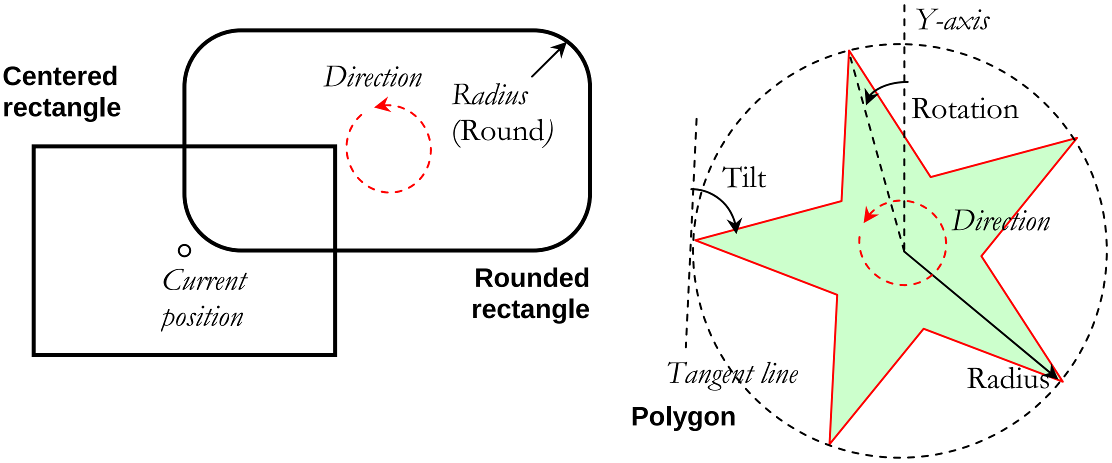
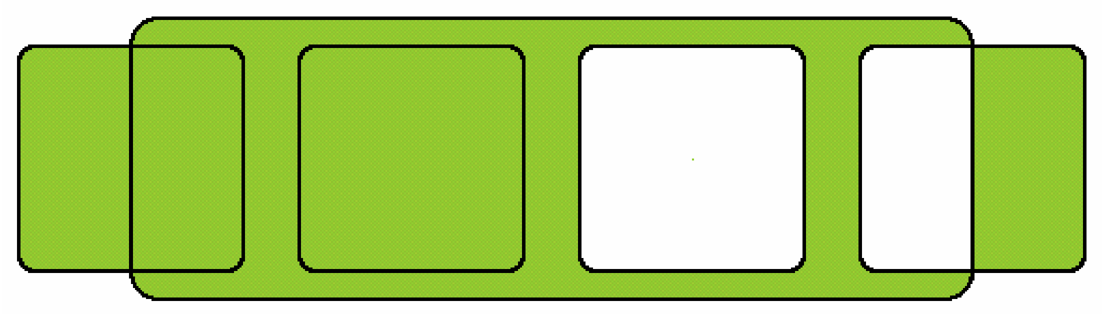


Figure 6-3. Parameters for rectangles and polygons.

PDF Everywhere always draws the rectangle (and other shapes) along a fixed direction (counter clockwise in this case), unless you instruct it to draw in the reversed direction. This direction affects the area to be filled in subsequent `fill` or `strokeFill` operations. To illustrate this, we juxtapose four rounded rectangles over a larger one, with the right-hand side two drawn in reversed direction. Here is what we would get:



Example 6-3. Effect of drawing direction of rectangles.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( );
my $g = $doc->newGraph( );
$g->moveTo( 350, 400 );
$g->drawRoundedRect( 300, 100, 10, 3, 1 );   # Add to clipping path
$g->moveTo( 200, 400 );
$g->drawRoundedRect( 80, 80, 6, 3, 1 );
$g->moveTo( 300, 400 );
$g->drawRoundedRect( 80, 80, 6, 3, 1 );
$g->moveTo( 400, 400 );
$g->drawRoundedRect( 80, 80, 6, 3, 1, 1 );   # Reversed direction
$g->moveTo( 500, 400 );
$g->drawRoundedRect( 80, 80, 6, 3, 1, 1 );
$g->strokeFill( );
$doc->writePDF( 'sample6-3.pdf' );
```

- **Polygon**

Function `drawPolygon` draws a symmetrical polygon with arbitrary number of sides. Each side is broken into two parts, which can be rotated around the bounding vertex, resulting in a star form (*cf.* Figure 6-3).

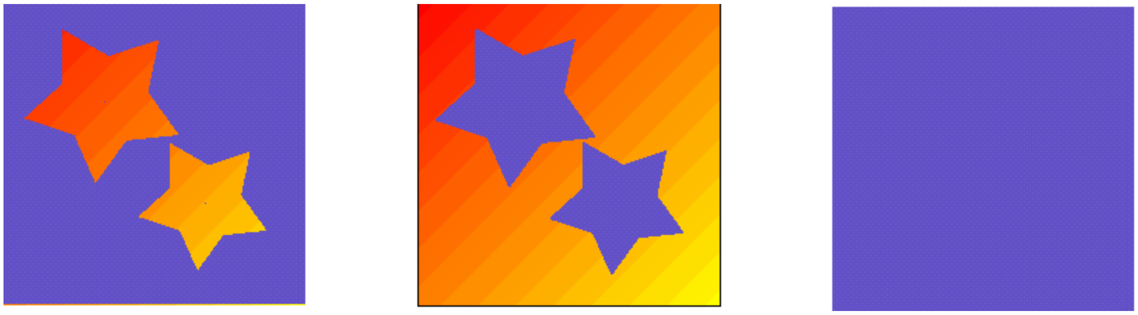
```
GraphContent::drawPolygon( $radius: size, $sides: int, $rotation:
   float, $tilt: float, [ $fill: int, $reverse: Bool ] )
```

```
<DrawPolygon
    Radius="size"
    Sides="int"       <!-- Must be greater than 3 -->
    Rotation="float"  <!-- Must be in the range of 0 through 180 -->
    Reverse="0|1"
    Tilt="float"      <!-- Must be in the range of 0 through 90 -->
    Fill="0|1|2|3"
/>
```

In the following example, a rectangular region is first filled with a gradient pattern (see later chapter in this document for detailed information about `PDFShading` object), then a rectangle (which covers the entire pattern) is used to intersect with two reversely drawn stars, and the clipping path is filled with a solid color. The resultant shape is like the stars are cut off paper and the underlying pattern is revealed.

Example 6-4. Effect of drawing direction on clipping path and fill.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( );
my $g = $doc->newGraph( );
$g->gradFill( new PDFShading( 'Linear', {'Dir'=>'TL->BR',
    'FromColor'=>'Red', 'ToColor'=>'Yellow'} ), new Rect( '0.5in',
    '0.5in', '3.5in', '3.5in' ) );
$g->newPath( );      # Begin a new path
$g->moveTo( '0.5in', '0.5in' );
$g->setColor( 'slateblue', 1 );
$g->drawRect( '3in', '3in', 3 ); # Add to clipping path
$g->moveTo( '1.5in', '2.5in' );
$g->drawPolygon( 60, 5, 30, 60, 3, 1 ); # Draw in reversed direction
$g->moveTo( '2.5in', '1.5in' );
$g->drawPolygon( 50, 5, 30, 60, 3, 1 );
$g->fill( );   # Fill the resultant clipping path
$doc->writePDF( 'sample6-4.pdf' );
```



If we remove the third argument when calling `drawRect`, then we are actually defining the stars to be the new clipping path, and the command `fill` will cause the starts to be filled in stead (the illustration at the center).

If we don't draw the stars in reversed direction, the stars would have no effect as long as they are entirely contained within the underlying rectangle (far right side of the illustration).

- **Circular and oval arcs**

The following functions draw an arc or a pie (when the arc is filled). You need to specify the radius and the starting and ending angles. The arc may be oval. (*cf.* Figure 6-4).

The arc is broken at the points where they intersect with the x- or y-axis. The algorithm of breaking a Bézier curve at a given point is the de Casteljau algorithm posted at http://www.faqs.org/faqs/graphics/algorithms-faq/. To find such a point, this program uses Newton's interpolation method.

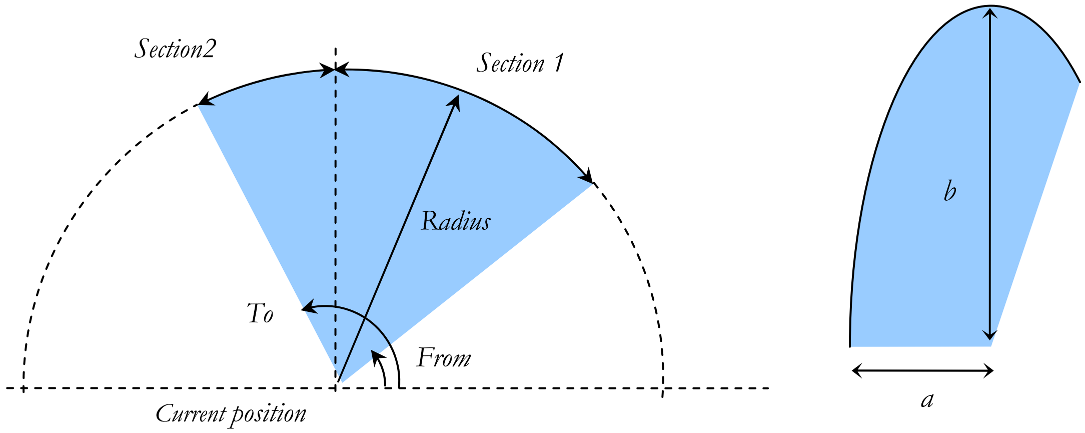


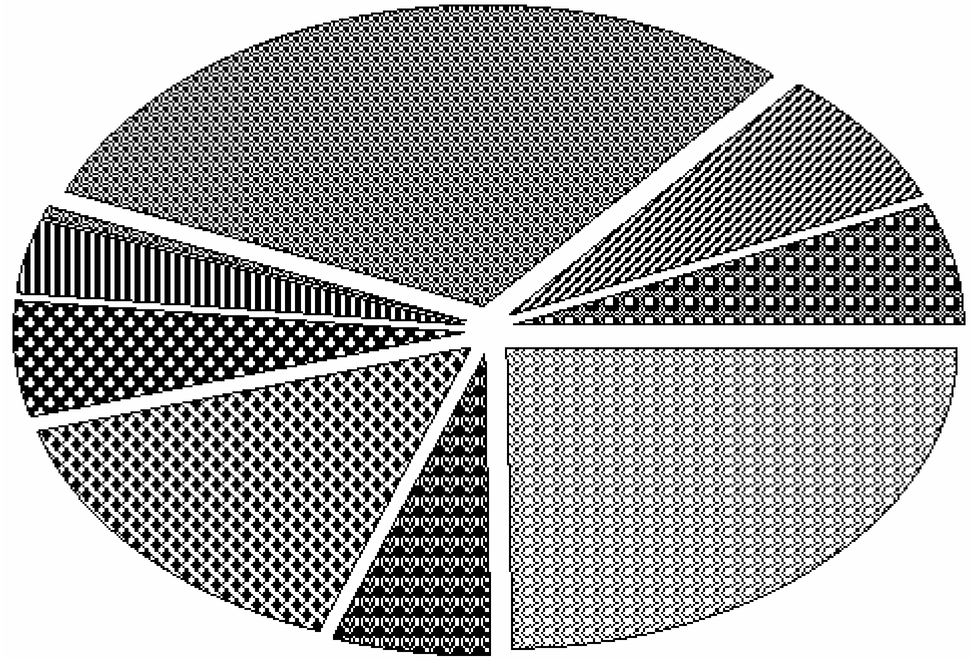
Figure 6-4. Illustration of circular and oval arcs.

```
GraphContent::drawArc( $radius: size, $from: float, $to: float,
   [ $fill: int ] )
GraphContent::drawOvalArc( $a: size, $b: size, $from: float, $to:
   float, [ $fill: int ] )

<DrawArc
    Radius="size"
    From="float"      <!-- Must be within 0 through 360 -->
    To="float"        <!-- Must be greater than From -->
    Fill="0|1|2|3"
/>

<DrawArc
    A="size"          <!—Used to draw oval arcs -->
    B="size"
    From="float"
    To="float"
    Fill="0|1|2|3"
/>
```

The example in this segment draws pie chart with random-generated value of angles and random-picked texture, using oval arcs.

Example 6-5. Pie chart implemented with random-generated oval arcs.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( );
my $g = $doc->newGraph( new Rect( 100, 100, 500, 400 ) );
my @nums = ( 0, 360 );     # Degree of angles
for( 1..8 ){ push( @nums, rand( 300 ) ); }
# We use the values in @num to divide 0~360 into subdomains.
@nums = sort { $a <=> $b } @nums;
my @txnames = keys %PDFTexture::BitMaps; # All texture type names
for( 0..$#nums-1 ){
   $g->setTexture( new PDFTexture( $txnames[ int( rand( 1 ) *
36 ) ] ), 1 );
   my $ang = ( $nums[$_] + $nums[$_+1] ) * 3.1416 / 360;
   $g->moveTo( 200 + 10 * cos( $ang ), 150 + 10 * sin( $ang ) );
   $g->drawOvalArc( 180, 120, $nums[$_], $nums[$_+1], 2 );
}
$doc->writePDF( 'sample6-5.pdf' );
```



▪ Circle

A circle is drawn in counterclockwise direction centering at current drawing position.

```
GraphContent::drawCircle( $r: size, [ $fill: int, $reverse: Bool ] )
```

```
<DrawCircle Radius="size" Reverse="0|1" Fill="0|1|2|3"/>
```

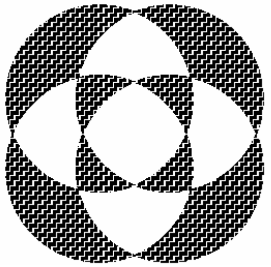
PDF Everywhere simulates a circle with four Bézier curves. Drawing direction affects how intersecting or overlapping shapes are filled.

Flood-fill and texture fill should be set prior to drawing. To fill the circle with a gradient pattern, set the parameter `Fill` to 3, then close path and call `fill`, `stokeFill`, or `gradFill` with the return value (see later content in this chapter). This behavior applies to most of the similar "*drawXXX*" functions.

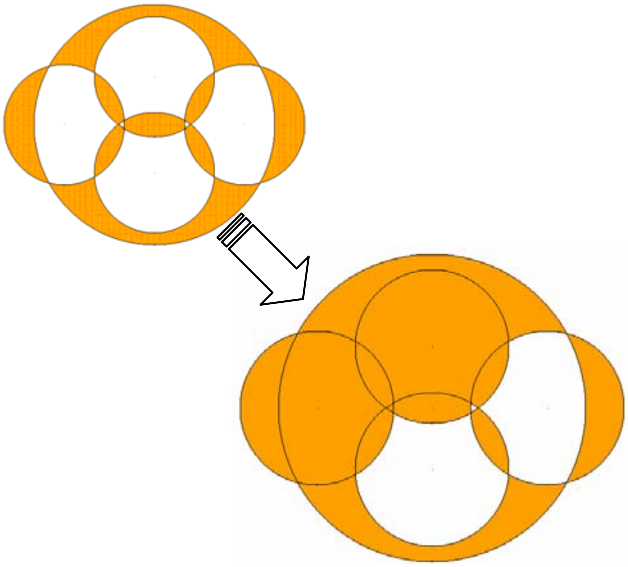
Now we create a two-page PDF document to show the effect of drawing direction. The first page shows a clipping path composed with four intersecting circles. The second page shows a similar one but two of the circles are drawn in reversed direction.

Example 6-6. Effect of circle's drawing direction and odd-even rule.

```
use PDF;
my $doc = new PDFDoc( );
# Page 1
$doc->newPage( );
my $g = $doc->newGraph( );
# Now we build a texture and use it for flood fill.
$g->setTexture( new PDFTexture( 'WireTilt' ), 1 );
map {
   $x = $_;
   map {
         $g->moveTo( $x, $_ );
         $g->drawCircle( 80, 3 );      # Add to clip path only
   } ( 300, 360 );
} ( 300, 360 );
$g->intersect( 1 );  # Intersect all circles with even-odd rule
$g->fill( 1 ); # Fill the path with even-odd rule
```



```
# Page 2
$doc->newPage( 'LetterR' );
$g = $doc->newGraph( );
$g->setColor( 'orange', 1 );
$g->moveTo( 500, 400 );
$g->drawCircle( 200, 3 );
$g->moveTo( 350, 400 );
$g->drawCircle( 100, 3 );
$g->moveTo( 500, 480 );
$g->drawCircle( 100, 3 );
$g->moveTo( 500, 320 );
$g->drawCircle( 100, 3, 1 );
$g->moveTo( 650, 400 );
$g->drawCircle( 100, 3, 1 );
# Case 1, odd-even rule
$g->intersect( 1 );
$g->strokeFill( 1 );
# Case 2 (replace these two lines with the following)
# $g->strokeFill( ); # Non-zero winding rule
$doc->writePDF( 'sample6-6.pdf' );
```



- **Oval (Ellipse)**

An oval is simulated by 8 segments of Bézier curves. The oval can be rotated by an angle in the range of 0 through 90 degrees.

```
GraphContent::drawEllipse( $a: size, $b: size, [ $rotation: float,
   $fill: int ] )

<DrawEllipse A="size" B="size" Rotation="float" Fill="0|1|2|3"/>
```

- **Text label**

The function `showText` shows a short text label at given position with fixed font face and line height, but the text can be stretched to given width and height, achieving certain text art effect. Each line break in the text causes a real line break. The text label can take many attributes, as illustrated in Figure 6-5 and explained in Table 6-6. These attributes are also used in the XML tag for this function.

```
GraphContent::showText( $x: size, $y: size, $text: string, $attr:
    HASH );

<ShowText X="size" Y="size" Text="string"
    <!-- More attributes here -->
/>
```

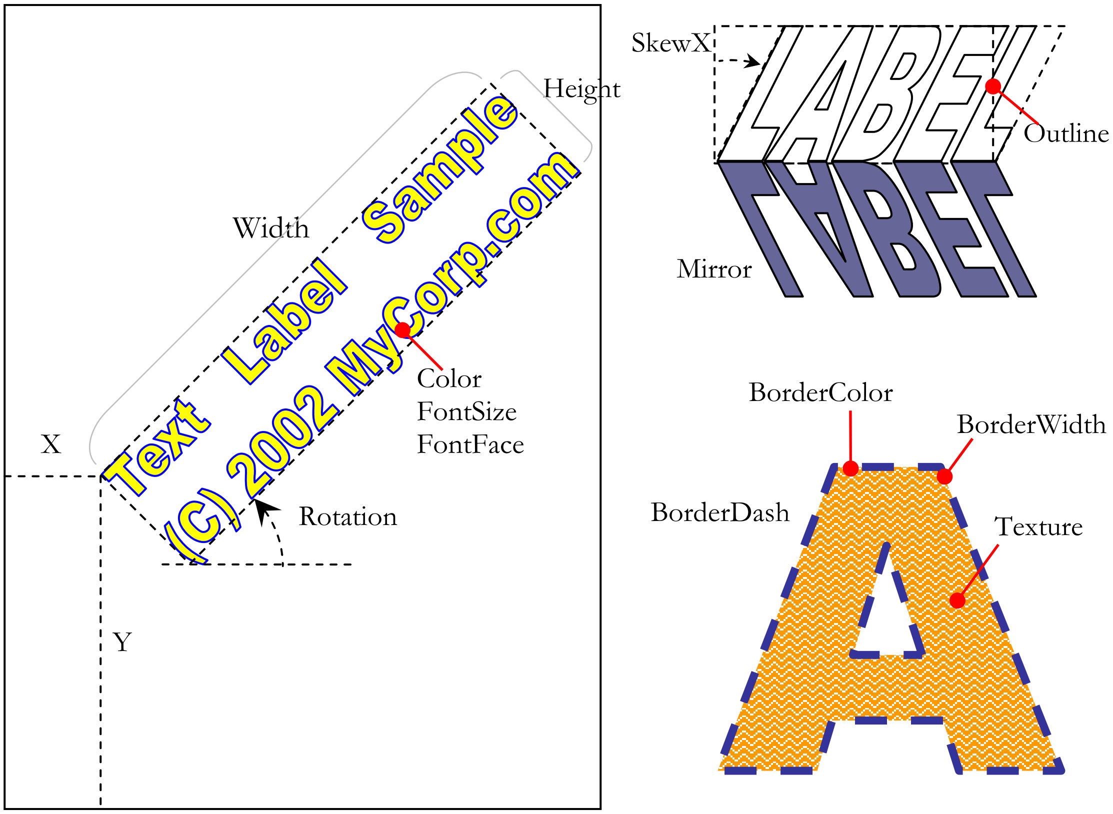


Figure 6-5. Illustration of the attributes of a text label.

Table 6-6. Available attributes of a text label.

	Key	Type	Explanation	Default
Font settings	FontFace	String	Name of the font to be used for text label.	Arial
	FontSize	Size	Font size.	12
	Leading	Size	Leading (line height).	12
	Encoding	String	Character encoding.	
Box & alignment	Width	Size	Length value specifying the width to fit.	
	FitWidth	String	String specifying the method to fit the width, must be WordSpacing, CharSpacing, or CharScaling	
	Height	Size	Length value specifying the height to fit.	
	FitHeight	String	String specifying the method to fit the height, must be LineHeight of Stretch.	
	Align	String	Left, Center or Right.	Left
Color & border	Color	Color	Text color.	Black
	Outline	Bool	If true, the text label is shown as outline only.	0
	BorderColor	Color	Border color.	Undef
	BorderWidth	Float	Border line width in points.	0
	BorderDash	String	Border dash pattern: can be Dashed, Dotted, Solid, or DashDot.	Solid
	Texture	Object	A PDFTexture object defining text fill pattern.	
	Shading	Object	A PDFShading object defining background shading (gradient) pattern.	
Transformation	SkewX	Float	Skew angle along horizontal direction (0-45).	
	SkewY	Float	Skew angle along vertical direction (0-45).	
	Rotation	Float	Rotation angle. 0 through 360.	
	Mirror	String	X or Y. Show the label as mirror image.	

The text can be filled with solid color, or a texture or shading. In outline mode, only outline is drawn and the text is transparent.

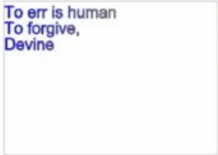
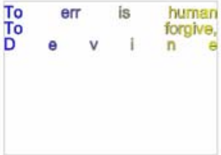
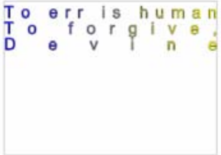
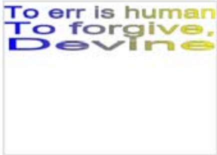
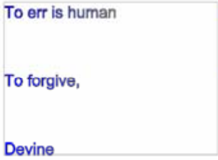
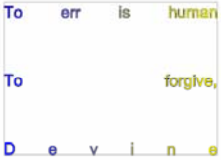
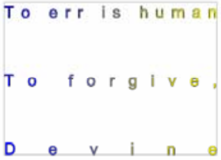
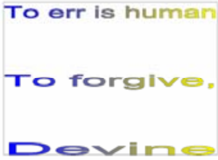
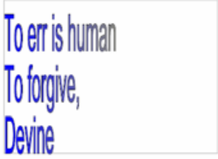
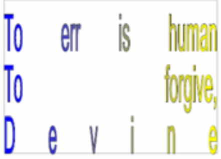
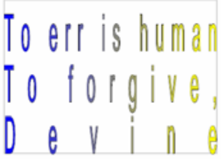
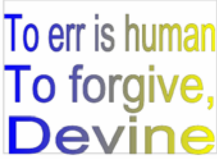
Should the text be rotated, it is anchored at top-left corner and rotates in counterclockwise direction. The label can also be skewed or shown as mirror image along x- or y-axis.

To fit the text into a given box, different approaches can be used. For horizontal fit, the program can adjust word spacing, character spacing, or scale the glyphs in proportion. For vertical fit, the program can either enlarge the line height or stretch the text. The illustration is in Table 6-7.

Each line of the text can be aligned to the anchor point by its left edge, center position, or right edge. The attribute `Align` is effective only when `FitWidth` is not present.

If you need to show a text with shading pattern, be sure to call `newPath` before calling `showText`, unless this call is the first call. This is to define a new path, which is the text outline. After calling `showText`, another call to `newPath` is needed if other graphics commands would follow.

Table 6-7. Various alignment schemes of a text label.

FitWidth FitHeight	(None)	WordSpacing	CharSpacing	CharScaling
(None)				
LineHeight				
Stretching				

Here we demonstrate how to import a page from a PDF file and add a text label onto it.

Example 6-7. Stamping a page with a text label.

```
use PDF;
my $doc = new PDFDoc( );
my $pdf = new PDFFile( 'foobar.pdf' );
$doc->newPage( { ImportSource => $pdf, ImportPage => 0 ) }; # Page 1
my $g = $doc->newGraph( { Opacity => 80 } );
$g->showText( 200, 600, "SAMPLE", {
    Width => '5cm',
    Height => 20,
    BorderWidth => 2,
    BorderColor => 'red',
    BorderDash => 'Dashed',
    Color => 'Yellow',
    FitWidth => 'CharScaling',
    FitHeight => 'Stretch',
    Rotation => 30,
} );
$doc->writePDF( 'sample6-7.pdf' );
```

- **Circular text label**

The function **`showBanner`** is similar to **`showText`** but is used to display the single- or multiple-line text in a circular form. All of the attributes listed in Table 6-6 for regular text labels can be used here, with the exception for "box and alignment" attributes. Additional attributes are listed in Table 6-8.

To distribute the characters evenly along the circular path, the program first finds out the width of each character and the total width, then computes the total arc span to determine the gap angle between the characters. Ideally, the center line of a character should pass the circle origin, so PDF Everywhere positions the character origin by observing the character width and current angle of that character. A character is then further rotated, scaled, and/or skewed with respect to this own origin (Figure 6-6).

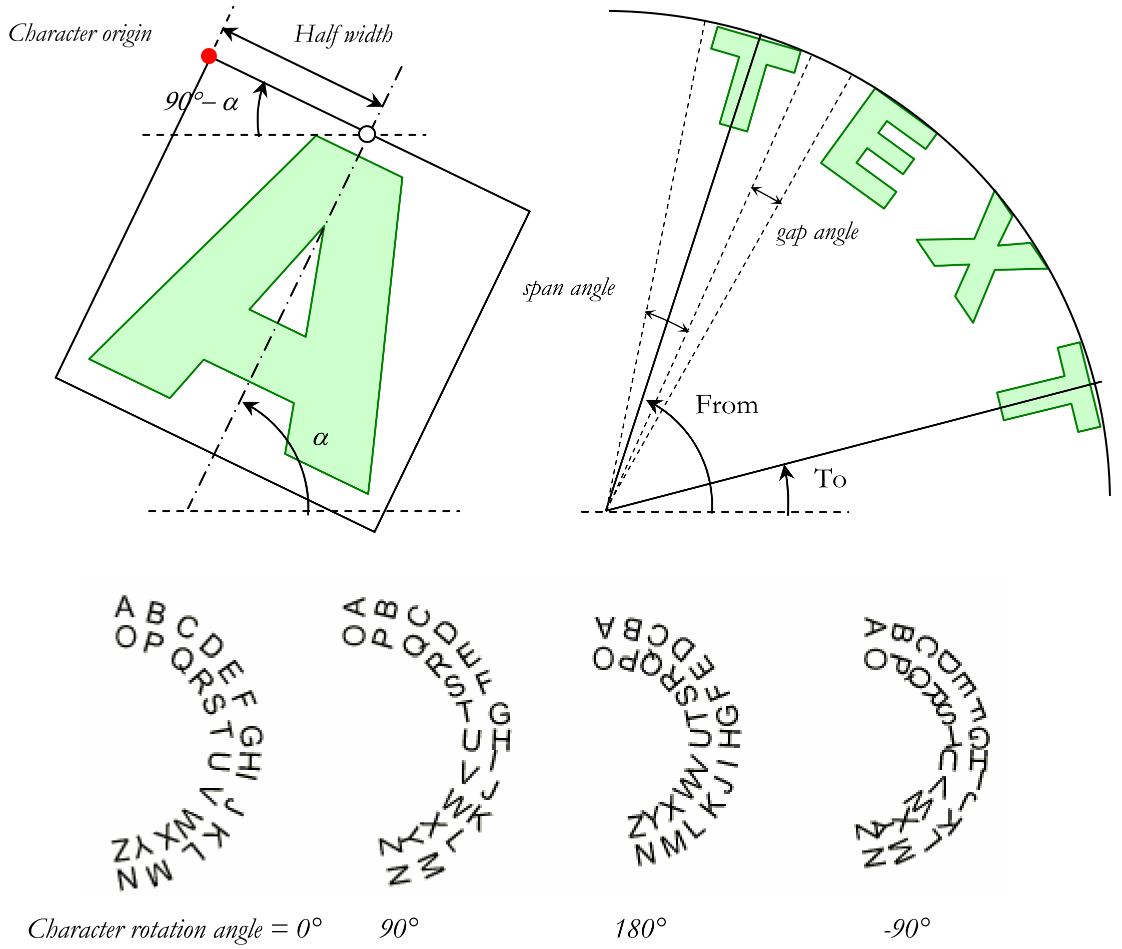


Figure 6-6. Illustration of the attributes of a circular text label.

```
GraphContent::showBanner( $x: size, $y: size, $text: string, $radius:
    size, $attr: HASH );

<ShowBanner FromAngle="float" ToAngle="float" Text="string"
    Radius="size" <!-- More attributes here -->
/>
```

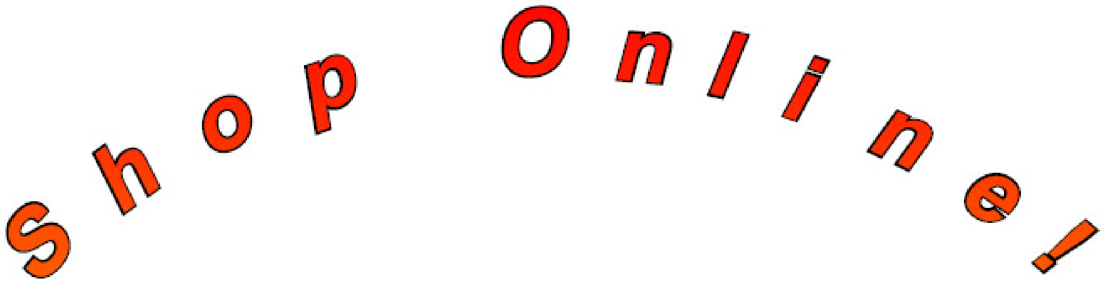
Table 6-8. Additional attributes of a circular text label.

Key	Type	Explanation	Default
`FromAngle`	Float	Greater then `ToAngle`, 0 through 360.	`180`
`ToAngle`	Float	0 through 360.	`0`
`Radius`	Size	Radius of the bound circle.	

In this example, we use an XML tag to define the text label, filled with smooth shading. The shading pattern is named "`Stamp`", and the XML is converted by the function `execXML` to be described later in this chapter.

Example 6-8. Circular text label implemented via XML code.

```
use PDF;
my $doc = new PDFDoc( );
my $tag = q{<ShowBanner X="2.5in" Y="2.5in" Radius="2in"
FromAngle="135" ToAngle="45" Text="Shop Online!" FontFace="Arial"
SkewX="10" FontSize="18" Shading="Stamp"/>};
# We use Page::getGraphics to get the default graphics layer
my $g = $doc->newPage( )->getGraphics( );
# Now create a shading pattern
$doc->newShading( 'Linear', {
    Name=>'Stamp', FromColor=>'Yellow', ToColor=>'Red', Centered=>1
} );
$g->execXML( $tag );
$doc->writePDF( 'sample6-8.pdf' );
```



- **Table grid**

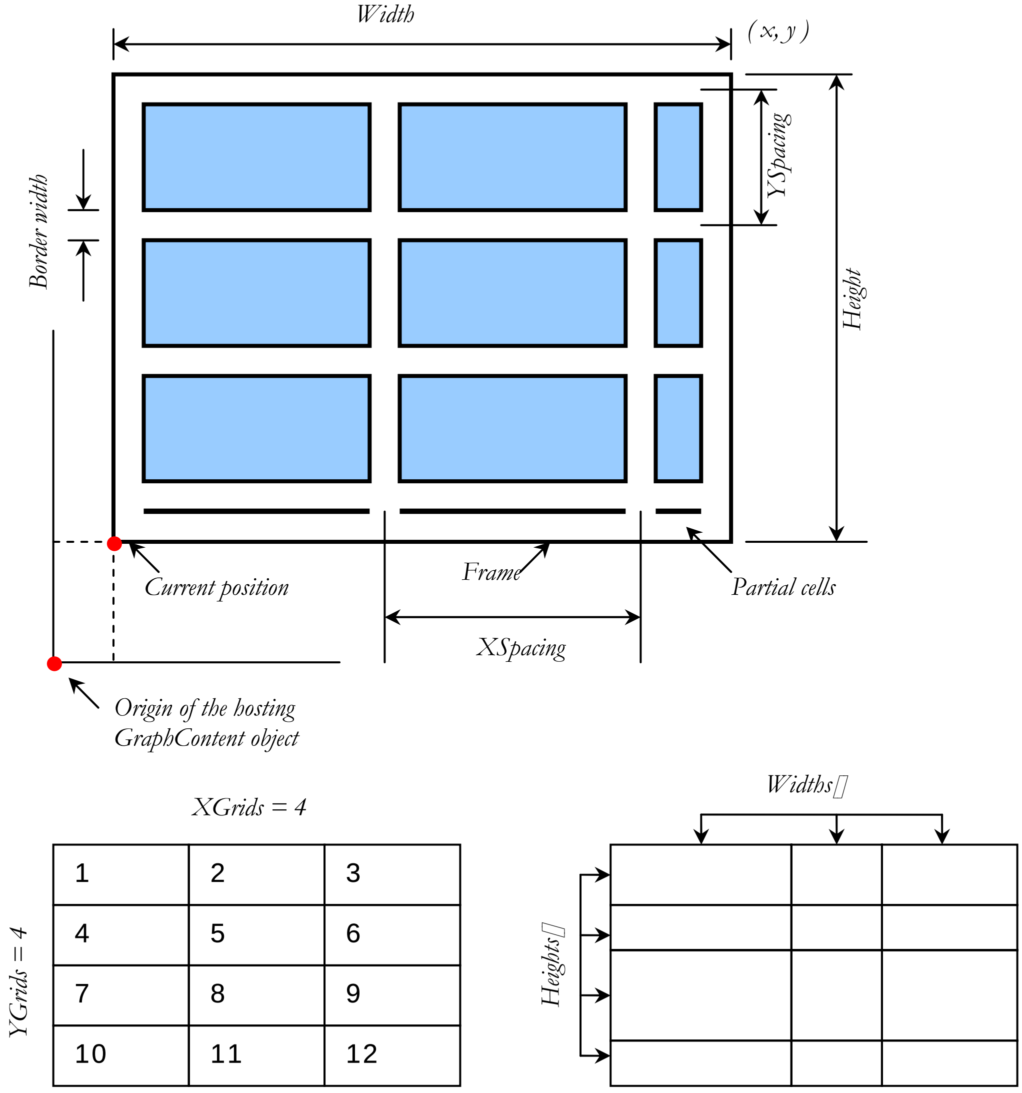
The following functions draw a grid in designated rectangular area from current position to a specified point or with given width and height, and with various attributes.

```
GraphContent::drawGrid( $width: size, $height: size, $fill: int,
    $attr: HASH ) : Rect[]
GraphContent::drawGridTo( $x: size, $y: size, $fill: int,
    $attr: HASH ) : Rect[]

<DrawGrid Width="size" Height="size" Fill="0|1|2|3" <!-- More -->
/>
```

The function returns a list of `Rect` objects for the cells created, measured in page coordinate system. The order is from left to right, top to bottom, as illustrated in Figure 6-7. This function can be useful when preparing a series of regularly distributed `Rect` objects to create new text areas or similar purposes. It is the basis for the table-based layout scheme.

The grids along any axis must be defined on one of the three available methods, although two different methods can be used for X- and Y-axis:

1. *By number of grids (`XGrids`, `YGrids`)*: All cells will have same width or height.
2. *By cell width/height (`XSpacing`, `YSpacing`)*: As many cells will be created to fill the available area. As a result, partial cells are likely to show up. To avoid getting partial cells, set `NoPartial` to true.
3. *By array of length values (`Widths`, `Heights`)*: The number of cells is determined by the number of values in each array. In addition, the values can also be in the form "*`d`*`#`" to specify ratios (where *`d`* is a number), "`*`" to mean freely-adjusted cells, or "`%`" to mean percentage.



1	2	3
4	5	6
7	8	9
10	11	12

Figure 6-7. Drawing a table grid.

The notations #, *, and % can not be mixed. The program does the following to calculate actual cell widths or heights:

1. Observe cells with specific widths/heights, including "%" cells.
2. Sum up the totals of the proportions of "#" cells, and the count of "*" cells.
3. Calculate unused width/height, excluding boundaries.
4. Allocate width/height to the "#" cells according to their ratio specifications, or evenly allocate it to all the "*" cells.

There must be one and only one of the attributes `XGrids`/`XSpacing`/`Widths` present for columns; similar for rows (Table 6-9).

Border width actually means the gap width between cells. This width is subtracted from actual cell width and height (Figure 6-6).

In addition, the X and Y values passed to `drawGridTo` can be negative, which means the coordinates are measured from the top-right corner of the page's art box.

Table 6-9. Attributes of a table grid.

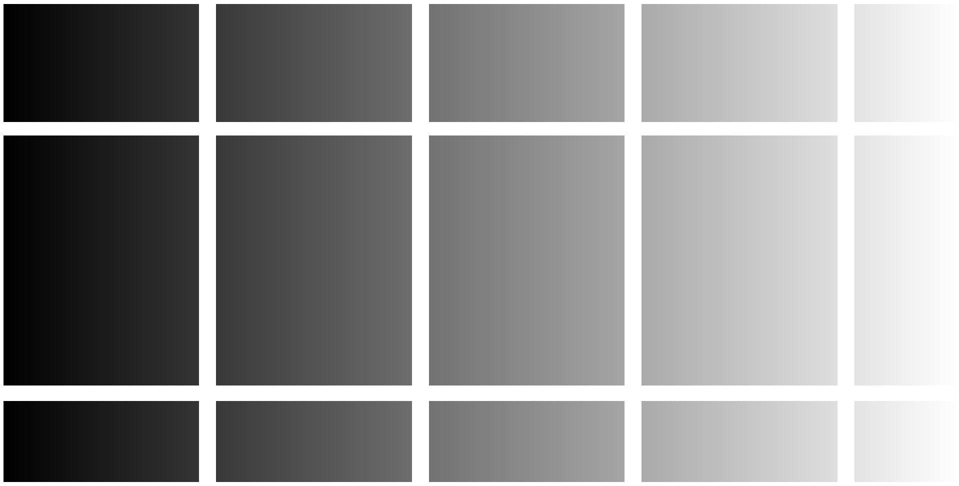
Key	Type	Meaning
`XGrids`	Integer	Number of grids along X-axis (columns)
`XSpacing`	Size	Cell width
`Widths`	ARRAY	Reference to an array of cell width values (left-right).
`YGrids`	Integer	Number of grids along Y-axis (rows)
`YSpacing`	Size	Cell height
`Heights`	ARRAY	Reference to an array of cell height values (top-down).
`BorderWidth`	Integer	(Optional) Border width actually means the gap width between cells.
`NoFrame`	Bool	(Optional) Omit the outer frame.
`NoPartial`	Bool	(Optional) If true, partial cells are not created. Meaningful only when cells are drawn with fixed width and/or height by using `XSpacing` and/or `YSpacing`.
`NoDraw`	Bool	(Optional) Do not draw the grid, but create the cells only.

To flood fill the cells, set a fill color *before* calling this function and set `Fill` to 1, or 2 if cell boundaries are also needed. This also applies to texture fill. The boundary color and width are also determined by previous operations. To fill the cells with a single gradient shading, i.e. filling through all the cells at one time rather than repeating shadings in each and every cell, set the parameter `Fill` to 3, then close path and call `gradFill` while building a new `Rect` with the first and the last return values (see Example 6-9 below).

To fill each cell independently, just use the return values of the function call.

Example 6-9. Fill a table grid with a smooth shading.

```
use PDF;
my $doc = new PDFDoc( );
# We use Page::getGraphics to get the default graphics layer
my $g = $doc->newPage( )->getGraphics( );
$g->moveTo( 100, 400 );
my @cells = $g->drawGridTo(
    '-2cm', '-4cm',  # Measured from right and top side
    3,   # Build clipping path only
    {
        XSpacing => 30,
        Heights => [ '1#', '2#', 50 ],
        BorderWidth => 5,
        NoFrame => 1
    } );
$g->closePath( );    # Important: must close the path before filling
$g->gradFill(
    new PDFShading( 'Linear',    # Linear gradient, left to right
        { FromColor=>'Black', ToColor=>'White', Dir=>'L->R' }
    ),
    new Rect(  # Build a new Rect that covesr all cells
        $cells[0]->left( ),
        $cells[-1]->bottom( ),
        $cells[-1]->right( ),
        $cells[0]->top( )
    )
);
$doc->writePDF( 'sample6-9.pdf' );
```



In XML, the tag `<drawGrid>` can also be used outside the `<Graph>` tag but within a `<Page>` tag to draw table cells for the table-based layout scheme. See Figure 3-3. In Perl, every time the function `drawGrid` is called, it puts the new table into the page's tables collection. To use previous table, the current table must be disabled using `PDFDoc::undefCurrTable`.

6.6 Utility Functions

- **Execute an XML tag**

The following function parses a XML tag, as shown in an XML document, to carry out corresponding operations without translating the code into Perl.

```
GraphContent::execXML( $xml: string )
```

The XML tag must be one of the tags described in this chapter, which in turn call the corresponding functions. A full list of tag names: `CurveTo`, `DrawArc`, `DrawCircle`, `DrawCustom`, `DrawEllipse`, `DrawGrid`, `DrawRect`, `Exec`, `Fill`, `LineTo`, `MoveTo`, `PolyLine`, `Set`, `ShowBanner`, `ShowImage`, `ShowText`, `ShowWMF`.

Please see Example 6-8 for the usage of this function.

- **Show a Windows Metafile**

Vector graphics stored in Windows® Metafile format (WMF files only) could be parsed by a `GraphContent` object to turn it into PDF code. The picture is aligned to a given point by its top-left corner. Please note that not all information contained in a WMF file is preserved, and the appearance may not be precisely correct.

```
GraphContent::showWMF( $filename: string, $left: size, $top: size,
    $attr: HASH )

<ShowWMF Left="size" Top="size" File="filename"
    <!-- Attributes goes here -->
/>
```

Table 6-10. Attributes for showing a Windows Metafile.

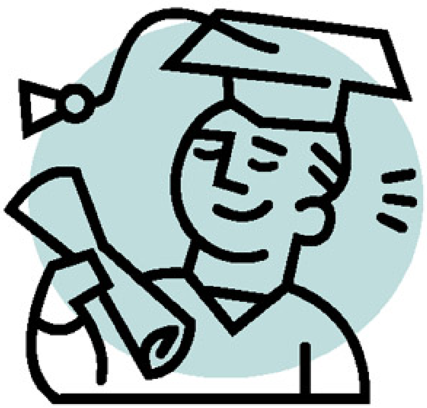
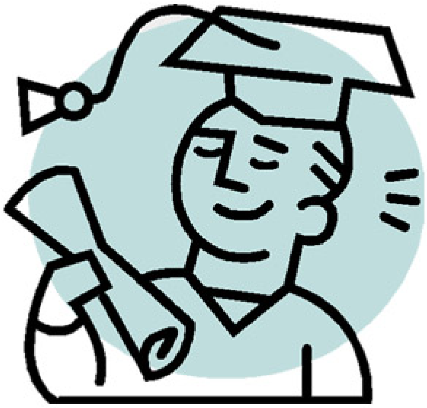
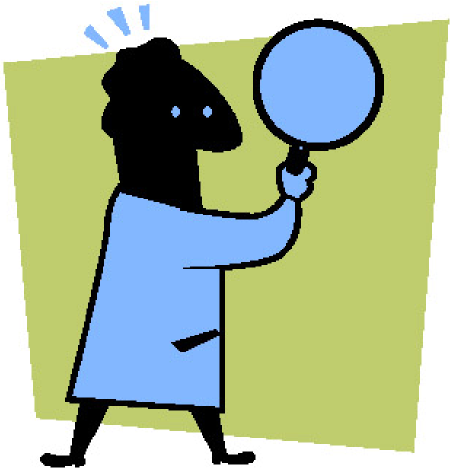
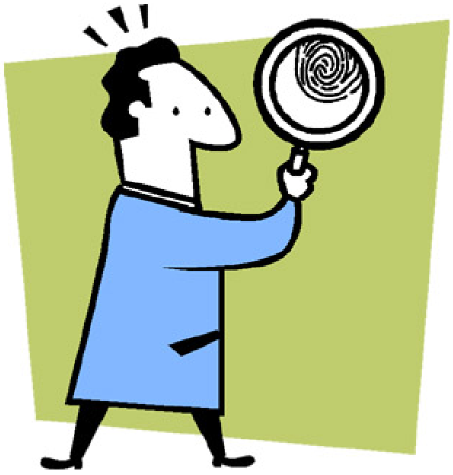
Key	Type	Meaning
`Scale`	Float	Scale factor (base 1), proportional
`ScaleX`	Float	Scale factor along X direction
`ScaleY`	Float	Scale factor along Y direction

A WMF file normally uses twip (1/20 point) as the unit for all measurements, and its X and Y axes are different than that used by PDF. This function always rotates the picture by 180 degree and always assumes that twip is used. To scale the picture, specify `Scale` or both `ScaleX` and `ScaleY` (need not to be the same).

⚠ WMF graphics are not treated as an image. Due to the different coordination system used in Windows, a mirror-Y transformation is applied. This is why the position to be specified is the top-left corner (an image's anchoring point is the bottom-left corner). If you find the image inversed, please specify a negative `ScaleY` value to correct it.

The following Windows GDI function calls most-frequently used in a metafile are converted into PDF Everywhere graphics function calls (and XML tags if XML output is needed in stead of PDF): `CreateBrushIndirect`, `CreateFontIndirect`, `CreatePenIndirect`, `Ellipse`, `ExtTextOut`, `LineTo`, `MoveTo`, `Pie`, `Polygon`, `Polyline`, `PolyPolygon`, `Rectangle`, `RoundRect`, `SetTextColor`. These functions are usually sufficient for most files, including those containing text. Unrecognized functions are simply overlooked. Table 6-10 shows two instances of converting metafiles. One of the file was converted without much loss, but the other is changed, as some GDI functions are skipped, causing the clipping paths to change.

Table 6-11. Misrepresentation of a Windows Metafile.

Key	Converted	Original
Good		
Bad		

- **Draw a POSTNET / FIM barcode**

POSTNET® and FIM barcode specified by the US Postal Service that can be created and used on letters and envelopes, as shown in Figure . The functions are:

```
GraphContent::drawPostnet( $x: size, $y: size, $zip: string, $dest:
    string )
GraphContent::drawFIM( $x: size, $y: size, $FIMtype: string )
```

The value for `Zip` must be a 5-digit zip code or 9-digit zip+4 code. The value for `Dest` must be a 2-digit delivery point code. The value of `FIMType` must be `A`, `B`, or `C`.

```
<DrawCustom Type="Postnet" X="size" Y="size" Zip="zip" Dest="dd"/>
<DrawCustom Type="FIM" X="size" Y="size" FIMType="A|B|C"/>
```

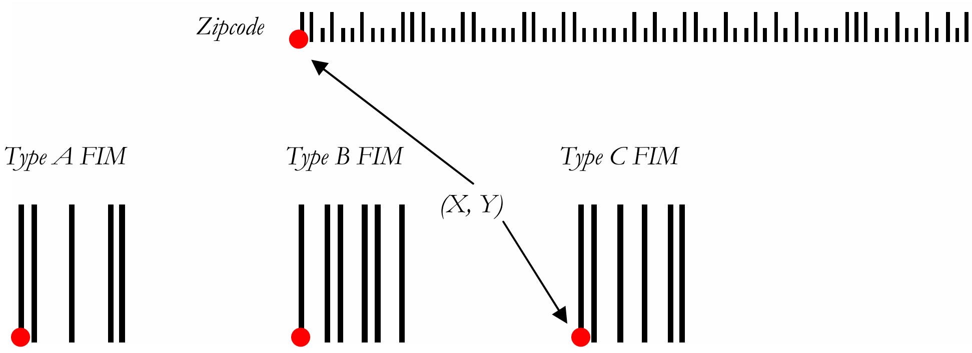


Figure 6-8. Postnet/FIM code.

Both functions draw strictly defined bars (width, length, spacing, color), which are not affected by current line style and color settings. Any non-digit characters in `Zip` or `Dest` values are skipped. A checksum digit will be appended to the generated barcode.

- **Texture fill and gradient fill**

For convenience, method `textureFill` is provided for rendering a *standalone* rectangle. The parameters are a `PDFTexture` object (in XML use the name of this object) and a Rect object (in XML use a list of coordinates). However, this function cannot be called via XML. In stead, you should set texture first (using `Set` tag) and then add a rectangle to clipping path and call the `Fill` command:

```
GraphContent::textureFill( $pattern: PDFTexture, $rect: Rect )
```

```
<Set Texture="name" Ground="1"/>
<MoveTo X="size" Y="size"/>
<DrawRect Width="size" Height="size" Fill="3"/>
<Exec Cmd="Fill"/>
```

Function `gradFill` is used to fill in a standalone rectangle with a previously defined gradient shading (`PDFShading` object, in XML use its name in stead). The third is useful when filling a custom path. Since the function is often used internally, an invisible rectangle shape is drawn and intersected with current clipping path. If `ApplyOnly` is true, then this treatment is skipped.

```
GraphContent::gradFill( $shading: PDFShading, $rect: Rect,
    $applyonly: Bool )
```

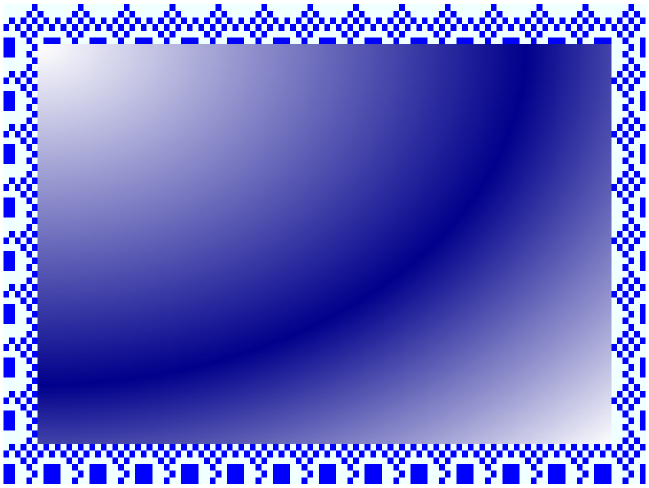
```
<Fill Shading="name" Rect="x1, x2, y1, y2" ApplyOnly="0|1"/>
```

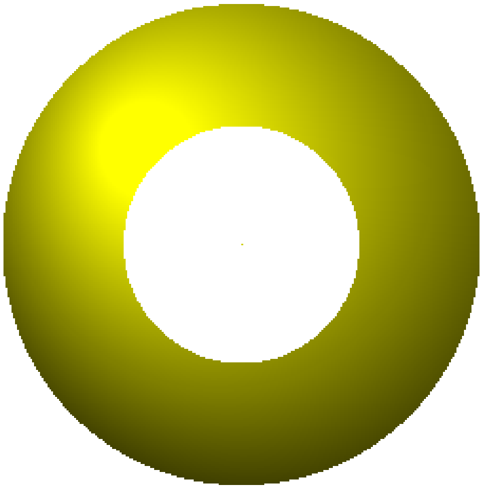
In Example 6-10, we draw a rectangle with a textured border, filled with a radial shading.

In Example 6-11, a `PDFShading` object is created via XML, and the page template is also defined in XML. This shading object is referred by name in XML.

Example 6-10. The use of texture and shading.

```
use PDF;
my $doc = new PDFDoc( );
# We use Page::getGraphics to get the default graphics layer
my $g = $doc->newPage( )->getGraphics( );
$g->setLineWidth( 12 );
$g->moveTo( 300, 400 );
$g->setTexture( new PDFTexture( 'Buttons',
    {'FgColor'=>'Azure', 'BgColor'=>'Blue'} ) );
# Intersect with current clipping path
$g->intersect( 1 );
$g->gradFill(
    new PDFShading( 'Radial', { FromColor=>'darkblue',
    ToColor=>'White', Centered=>1, Bound=>0.6, Dir=>'TL->BR' }
),
$g->drawRect( 100, 60, 0, 1 ) );
$doc->writePDF( 'sample6-10.pdf' );
```





Example 6-11. Use a gradient shading to fill in a custom clipping path.

```
use PDF;
my $tpl = q{
<Template>
   <Shading Type="Radial" Name="sphere" FromColor="Yellow"
         ToColor="Black" Coords="0.3, 0.7, 0.1, 0.6, 0.6, 0.8"/>
   <Graph>
   <MoveTo X="300" Y="150"/>
   <DrawCircle Radius="100" Fill="3"/>
   <DrawCircle Radius="50" Reverse="1" Fill="3"/>
   <Exec Cmd="Intersect" Eof="1"/>
   <Exec Cmd="ClosePath"/>
   <Fill Shading="sphere" Rect="200, 50, 400, 250" ApplyOnly="1"/>
   </Graph>
</Template>
};
my $doc = new PDFDoc( );
new Page( {'Template'=>$tpl, 'Size'=>'Postcard'} );
$doc->writePDF( 'sample6-11.pdf' );
```

- **Show an image or an XObject**

An image is represented by an `ImageContent` object. It can be shown in a graphics layer with various transformations. As a *resource*, any image can be used anywhere in the same document for multiple times once it is defined, and each time the appearance can be different. The image definition can be inserted into the `GraphContent` object's stream data, know as an inline image. In this case, the image is not a resource and cannot be used elsewhere. The function `showInlineImage` is intended only for *internal* use for form buttons and annotations, and no XML tag is prepared.

The image's anchoring point is its lower left corner, as illustrated in Figure 6-9. Transformations (not available for inline images) are passed like attributes via a hash. In XML, the image is referred to by name, just like other resources (shading, texture, etc.)

An XObject (see §3.5) is just like an image, and be displayed in a graphics layer with same transformations and attributes.

```
GraphContent::showImage( $img: ImageContent, $left: size,
    $bottom: size, $attr: HASH )
GraphContent::showInlineImage( $img: ImageContent, $left: size,
    $bottom: size )
GraphContent::showXObject( $img: ImageContent, $left: size,
    $bottom: size, $attr: HASH )

<ShowImage
    Image="name" Left="size" Bottom="size"
    <!-- More attributes here -->
/>

<ShowXObject
    XObject="name" Left="size" Bottom="size"
    <!-- More attributes here -->
/>
```

For table-based layout scheme, the image can be aligned to a one of the nine possible positions in current active cell (cf. Figure 3-3), provided that it is neither skewed nor rotated: the center of cell, the center of four borders, and the four corners. The default cell is the entire box of the graphics layer. The cells are changed after each `drawGrid` call. In XML, both functions can take the parameters defined in Table 3-2.

An image (not skewed, not rotated) can also be linked to a hyperlink address. The image can be scaled or flipped, and still be clickable. The attributes are illustrated in Figure 6-9, and listed in Table 6-12.

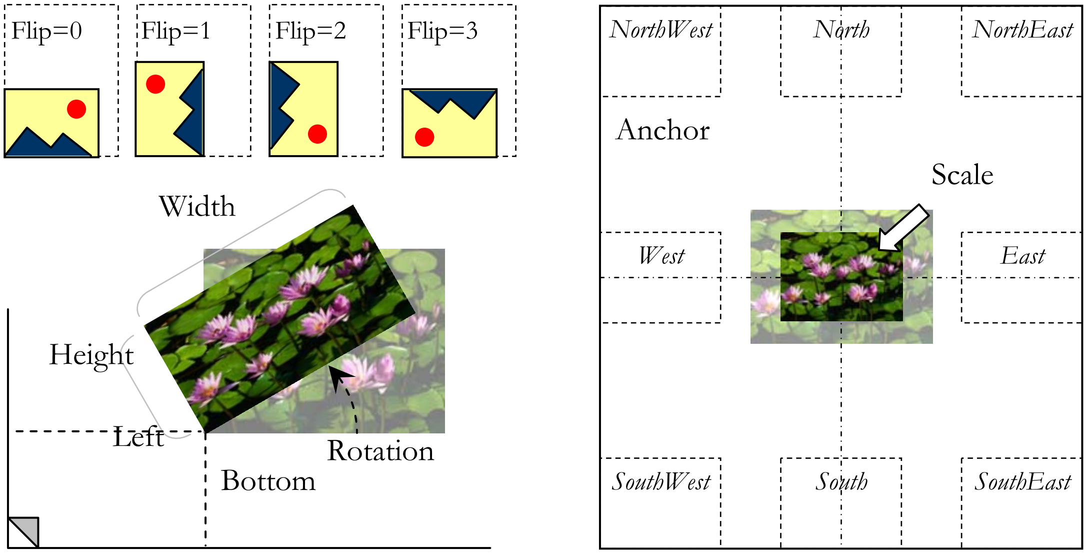


Figure 6-9. Illustration of attributes for showing an image.

Table 6-12. Attributes for showing an image.

Key	Type	Meaning
`Scale`	Float	Scale factor for proportional resizing, default 1.
`Scale,ScaleY`	Float	Scale factor for X and Y direction, need not to be same.
`Width`	Size	Desired display width.
`Height`	Size	Desired display height.
`Anchor`	String	Anchored position, must be one of the following: `NorthEast`, `North`, `NorthWest`, `East`, `Center`, `West`, `SouthEast`, `South`, `SouthWest`.
`Rotation`	Float	Rotate angle around the origin of the image (0-360).
`Flip`	Integer	Tip the image while aligning to the given (x, y) point. Must be 0, 1, 2, or 3 (*cf.* Figure 6-8).
`Mirror`	String	Mirror transformation, must be `X` or `Y`.
`SkewX,SkewY`	Float	Skew angle along X or Y direction (0-45).
`Href`	String	URL to link the image to.

Acrobat Reader program renders an image in a 1 x 1 cell then stretches and rotates it by applying mathematical transformation. Therefore, the actual resolution and shape of an image is freely changeable. When showing an image, either `Scale` or both `Width` and `Height` should be specified, or otherwise the image would be shown at a resolution of 72dpi (one pixel to one point). If you specify either `Width` or `Height` but not both, the images will be scaled *proportionally* to specified width or height.

Rotation, scaling, and skewing are performed with respect to the bottom-left corner. However, flipping is done by swapping and offsetting so that the left and bottom sides of a flipped image aligns to the given position. Mirroring is done by transforming the image after all other operations are complete, with respect to bottom (Y) or left (X) side.

Example 6-12. A minimal image-to-PDF converter.

```
use PDF;
use File::Basename;
my $filename = $ARGV[0]; # Supply a JPG file name at command line
my $doc = new PDFDoc( { PageMode => 'FullScreen' } );
my $img = new ImageContent( $filename ); # Import the image
$doc->newPage( {
    Width => $img->getWidth( ), Height => $img->getHeight( )
} ); # Make page same dimension as the image
$doc->newGraph( )->showImage( $img, 0, 0 ); # Show image as is
$doc->writePDF( basename( $filename, ".jpg" ) . ".pdf" );
```

6.8 Texture

A `GraphContent` object can set a texture as current *color* using the `setTexture` method. Textures, represented by `PDFTexture` objects, are of a special “pattern” color space, which could be defined with a series of graphic operations. In PDF Everywhere, the common texture type is defined with an image, which can be chosen from one of the predefined 36 patterns (Figure 6-10) or a user-created `ImageContent` object. A pattern is repeated along both X and Y directions, with the cell spacing changeable. The predefined patterns are 8 × 8 pixels, with changeable foreground and background colors. A user-supplied image can have arbitrary dimensions. The image used as texture can be scaled in advance. Alternately, you can specify desired width and height by inserting more keys to the attribute hash.

A texture can also be created from a `GraphContent` object to build custom tiling pattern. Two types of patterns can be created this way: colored and uncolored. If uncolored, a color must be specified each time the pattern is used. The `GraphContent` should be assigned to a small working space (rectangle), because this region is going to be used as the tiling cell. Complex patterns can be made by working on those designs in a `GraphContent` then using it as a pattern. In an XML document, all the XML tags denoting vector graphics functions (used within `Graph` tags) can also be used within `Texture` tags, which are used to create a `PDFTexture` object.

A texture can be created via one of the following three forms of the function call (`newTexture` defined in `PDFDoc`, which calls `PDFTexture::new`):

1. Predefined texture types (the type is one of the names listed in Figure 6-10). The attributes are checked in Table 6-13.

```
PDFDoc::newTexture( $type: string, $attr: HASH )

<Texture Type="typename" FgColor="color" BgColor="color"
    <!-- More attributes -->
/>
```

2. Create a texture from an image. The parameter for the function call is the `ImageContent` object; but in XML, please use its name.

```
PDFDoc::newTexture( $img: ImageContent, $attr: HASH )

<Texture Image="name"
    <!-- More attributes -->
/>
```

3. Create a texture from a graphics layer. In the function call, use the `GraphContent` object. In XML, put the tags for the graphics layer inside the `Texture` tag.

```
PDFDoc::newTexture( $graph: GraphContent, $attr: HASH )

<Texture Name="name" Width="size" Height="size"
    <!-- More attributes -->
>
    <!-- Tags for graphics drawings -->
</Texture>
```

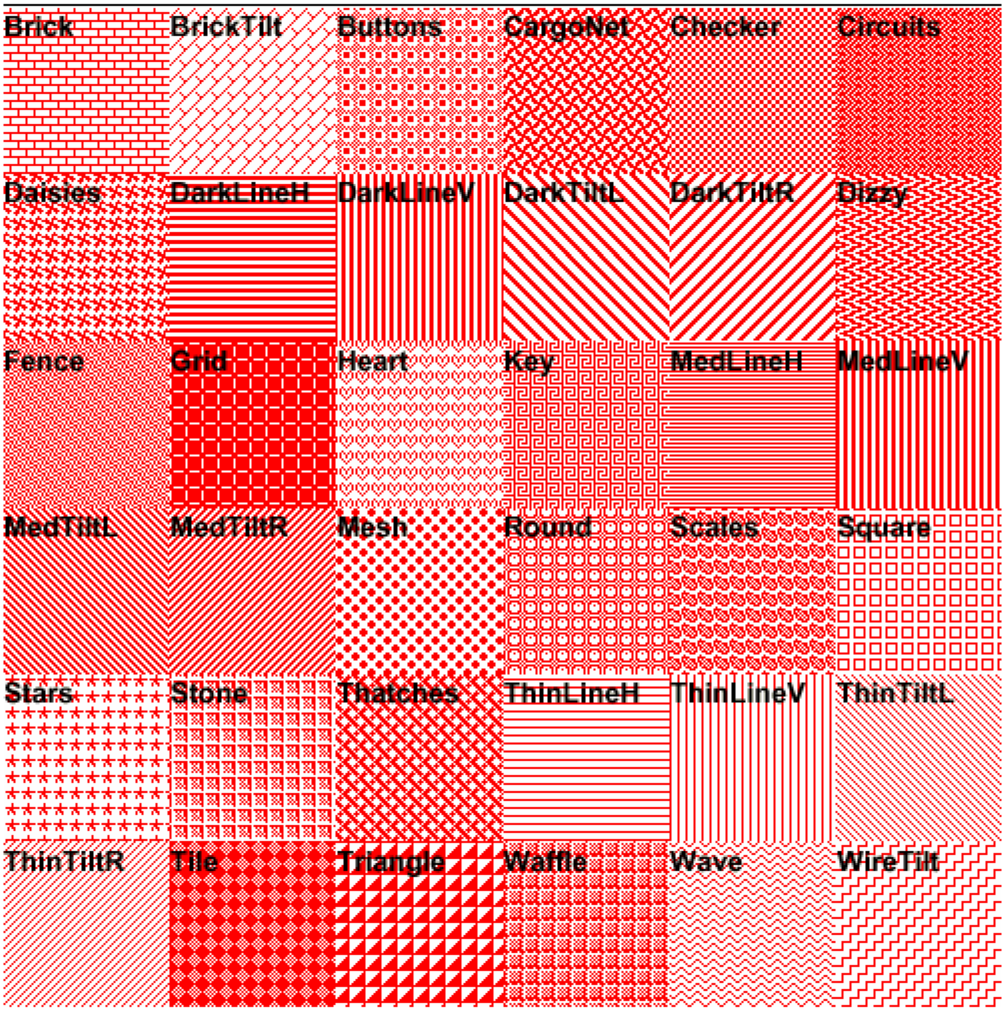


Figure 6-10. 36 predefined texture types.

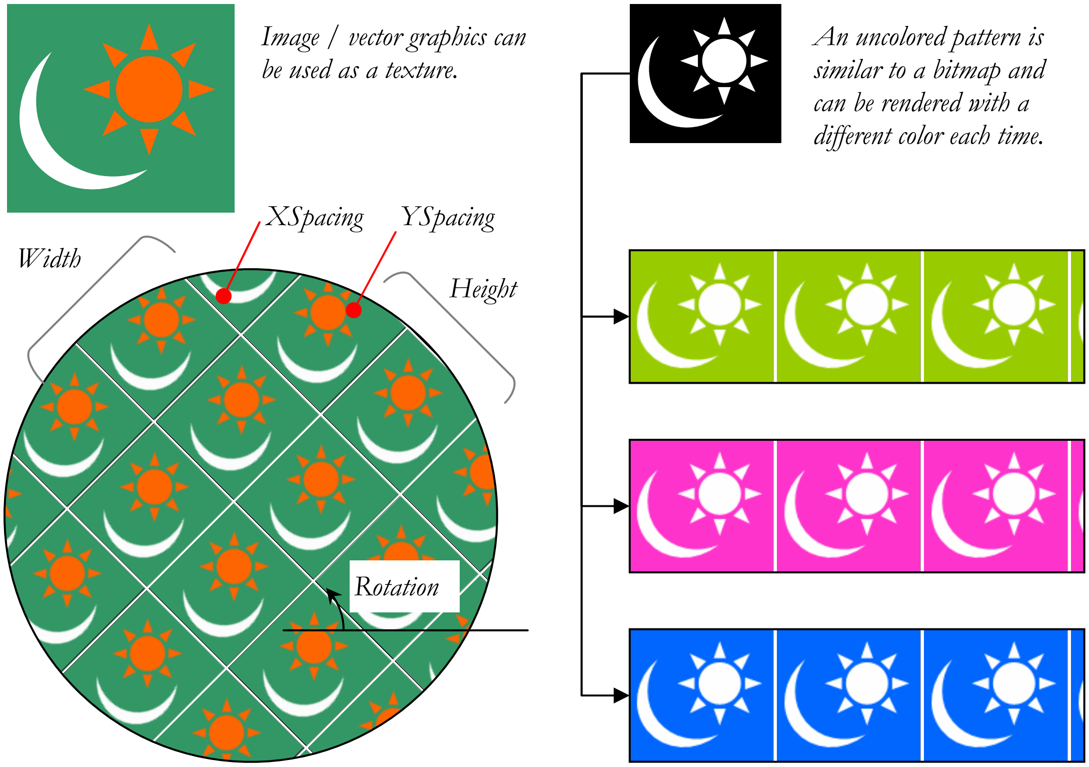


Image / vector graphics can be used as a texture.
An uncolored pattern is similar to a bitmap and can be rendered with a different color each time.
XSpacing
YSpacing
Width
Height
Rotation

Figure 6-11. Illustration of a texture.

Table 6-13. Attributes of a texture.

Key	Type	Meaning	▨	🌷	☆
Name	String	A name for this texture resource.	√	√	√
Image	Image	The image that is used as tiling pattern.		√	
XSpacing	Integer	Column gap width in pixel.	√	√	√
YSpacing	Integer	Row gap width in pixel.	√	√	√
Width,Height	Size	Desired display width and height for image texture.	√	√	√
Scale	Float	Scaling factor for image texture only.	√	√	√
Rotation	Float	Rotation angle.	√	√	√
SkewY,SkewY	Float	Skew angle along X and Y axes, for image texture only.	√	√	√
FgColor	Color	Foreground color, for predefined texture only.	√		
BgColor	Color	Background color, for predefined texture only.	√		
Uncolored	Bool	For vector graphics texture only.			√
ColorSpace	String	For uncolored texture: RGB, CMYK, Gray.			√

▨ *Predefined texture;* 🌷 *Image-based texture;* ☆ *Vector-graphics-based texture.*

The following example creates the output shown in Figure 6-10. It accesses the internal variable to get the names of these textures.

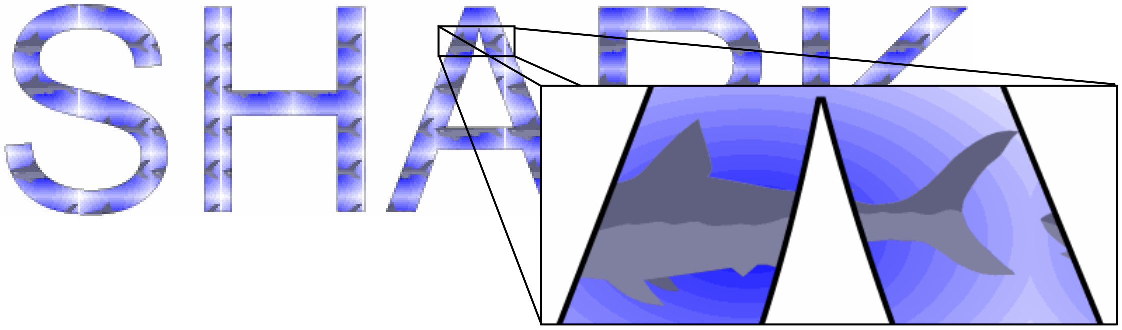
Example 6-13. Printing all predefined texture types.

```
use PDF;
my $doc = new PDFDoc( );
my $oGraph = $doc->newPage( )->getGraphics( );
$oGraph->moveTo( 100, 100 );
@cells = $oGraph->drawGrid( 500, 500, 0, {XGrids=>6, YGrids=>6} );
for my $pat ( sort keys %PDFTexture::BitMaps ){
   my $r = shift( @cells );
   $oGraph->textureFill(
     new PDFTexture( $pat, {FgColor=>'white',BgColor=>'Red'} ),
     $r,
   );
   $oGraph->setColor( 'Black', 1 );
   $oGraph->showText( $r->left( ), $r->top( ), $pat );
}
$doc->writePDF( "sample6-13.pdf" );
```

The following example creates a texture from vector graphics. A standalone graphics layer is first created (marked for internal use so that is won't mess up the PDF output), then graphics drawing operations are called, and finally a texture is created from this layer and used on a text label, which is done by another graphics layer.

Example 6-14. Creating and using a vector graphics texture.

```
use PDF;
my $doc = new PDFDoc( );
my $oGraph = $doc->newPage( 'LetterR' )->getGraphics( );
# Mark the layer internal. It won’t be linked to the page.
$g = new GraphContent( new Rect(0, 0, 50, 30), { InternalUse=>1 } );
# Now we build the vector graphics
$g->gradFill(
   new PDFShading( 'Radial', {FromColor=>'Blue',ToColor=>'white'} ),
   new Rect( 0, 0, 50, 30 ) );
$g->showWMF( "shark.wmf", 25, 20, {'Scale'=>0.3} );
$oGraph->showText( 40, 400, "SHARK", {Texture=>new PDFTexture( $g ),
   FontSize=>200, BorderColor=>'Black'} );
$doc->writePDF( 'sample6-14.pdf' );
# $doc->exportXML( );
```



The XML output of this program is as follows. We can see that the graphics-drawing tags are inserted into a pair of `Texture` tags. A name is automatically assigned.

```
<?xml version="1.0" standalone="yes"?>
<PDF>
   <Shading Type="Radial" Name="N002" Centered="0" FromColor="Blue"
ToColor="white" ColorSpace="RGB"/>
   <Texture Name="N003" Width="50" Height="30">
         <Fill Shading="N002" Rect="-1, -1, 51, 31" ApplyOnly="0"/>
         <ShowWMF File="shark.wmf" Left="25" Top="20" Scale="0.3"/>
   </Texture>
   <Page Name="N000" Width="792" Height="612">
         <Graph Rect="0, 0, 792, 612" Name="N001">
               <ShowText X="40" Y="400" Text="SHARK" FontSize="200"
BorderColor="Black" Texture="N003" />
         </Graph>
   </Page>
</PDF>
```

An uncolored texture has to be created from a `GraphContent` object. When applied, a color has to be designated. In the following example, we create an uncolored pattern and use it to paint a circle and to draw a line, using different colors. Remember a texture is just like a special color. Also, note the pattern is always continuous, despite the difference colors used.

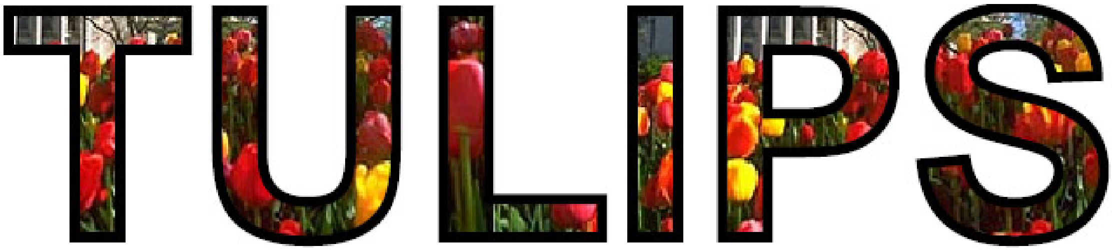
Example 6-15. Creating and using an uncolored vector graphics texture.

```
use PDF;
my $doc = new PDFDoc( );
my $G = $doc->newPage( 'LetterR' )->getGraphics( );
my $g = $doc->newGraph( new Rect(0, 0, 40, 40), {InternalUse=>1} );
$g->moveTo( 5, 5 );
$g->drawRoundedRect( 30, 30, 8 );
$g->showText( 5, 32, "LOVE YOU!",
   {'Width'=>30, 'FontSize'=>20, 'FitWidth'=>'CharScaling'} );
# Now create the texture
my $tx = new PDFTexture( $g, {'Uncolored'=>1, 'Rotation'=>30} );
# Now apply the texture
$G->setTexture( $tx, 1, 'lightblue' );
$G->moveTo( 300, 200 );
$G->drawCircle( 100, 1 );

$G->setLineWidth( 20 );
$G->setLineJoin( 1 );
$G->setTexture( $tx, 0, 'Tomato' );
$G->drawPolygon( 60, 3 );
$doc->writePDF( 'sample6-15.pdf' );
```

Finally, here's an example to fill in a text label with image via texture. Note that the texture is always continuous and starts from page origin.

Example 6-16. Using a texture made from an image.

```
use PDF;
my $doc = new PDFDoc( );
my $g = $doc->newPage( 'LetterR' )->getGraphics( );
$g->showText( 100, 300, "TULIPS", {
   FontSize=>120, BorderWidth=>4, BorderColor=>'black',
   FontFace=>'Arial,Bold', Texture=>
   $doc->newTexture( new ImageContent("tulip.bmp"), {Scale=>0.8} )
   } );
$doc->writePDF( 'sample6-16.pdf' );
```



6.9 Gradient Shading

A gradient shading, represented by a `PDFShading` object, is a smooth transition from one color to another. PDF standard defines a variety of shadings, but PDF Everywhere chose to implement linear (axial) and radial shadings only. It doesn't support gradients with three or more colors.

A shading needs two colors and one direction. When the direction is not specified, linear shading will go from left to right, while radial shading goes from center outwards.

Alternately, an array can be supplied to specify the coordinates of the shadings. A linear shading needs two points, while the radial shading needs two additional values to define the radii of circles to begin and end with. The coordinates should be in the range of 0 ~ 1, assuming the cell (no matter how large the final pattern is stretched) has a 1 × 1 unit size. The radius of the circle, however, can be larger. By default, the colors extend beyond both ends (if not, the area beyond the ends will be blank).

A shading can also be symmetrical. That is, the shading reaches the ending color on halfway then transits back to the starting color till the ending point. A parameter can be defined to specify the boundary of the two semi-domains.

⚠ Each shading must be applied to a clipping path. It can not be used as a stroke or fill color as a texture can. Therefore, care must be taken to ensure proper sequence of graphic-state operators are executed, otherwise the shading might cover the entire page.

To create a shading, call function `newShading` on a `PDFDoc` object, which calls the `new` operation defined in `PDFShading` class. Pass type name (`Axial` has the same meaning as `Linear`) and a hash defining the attributes as the arguments (Table 6-14).

```
PDFDoc::newShading( $type: string, $attr: HASH )

<Shading Name="string" Type="Axial|Linear|Radial"
    <!-- Attributes -->
/>
```

Table 6-14. Attributes of a gradient shading.

Key	Type	Meaning
`Name`	String	Name for this shading resource.
`FromColor`	Color	Starting color
`ToColor`	Color	Ending color
`ColorSpace`	String	Color space: RGB, CMYK, Gray.
`Dir`	String	Direction of the color transition, see Figure 6-12.
`Centered`	Bool	If true, the shading is symmetrical.
`Bound`	Float	In the range of 0 through 1. Only used for Centered = 1.
`Coords`	ARRAY	Optional. Four of six elements, see Figure 6-12.

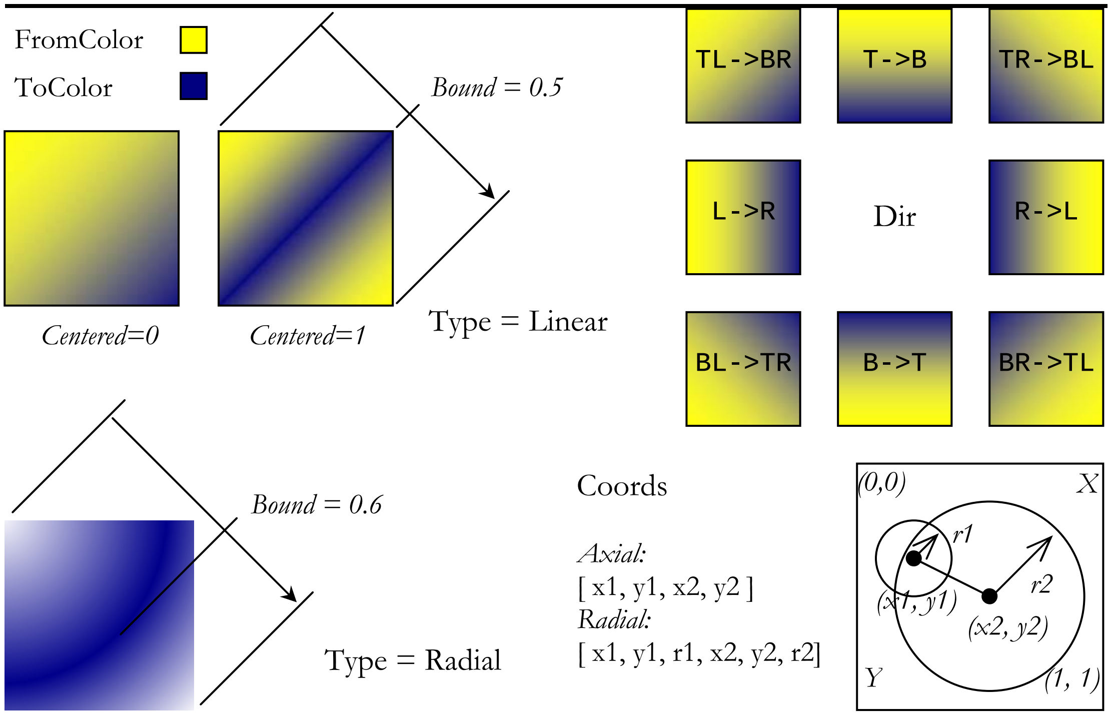


Figure 6-12. Illustration of a gradient shading.

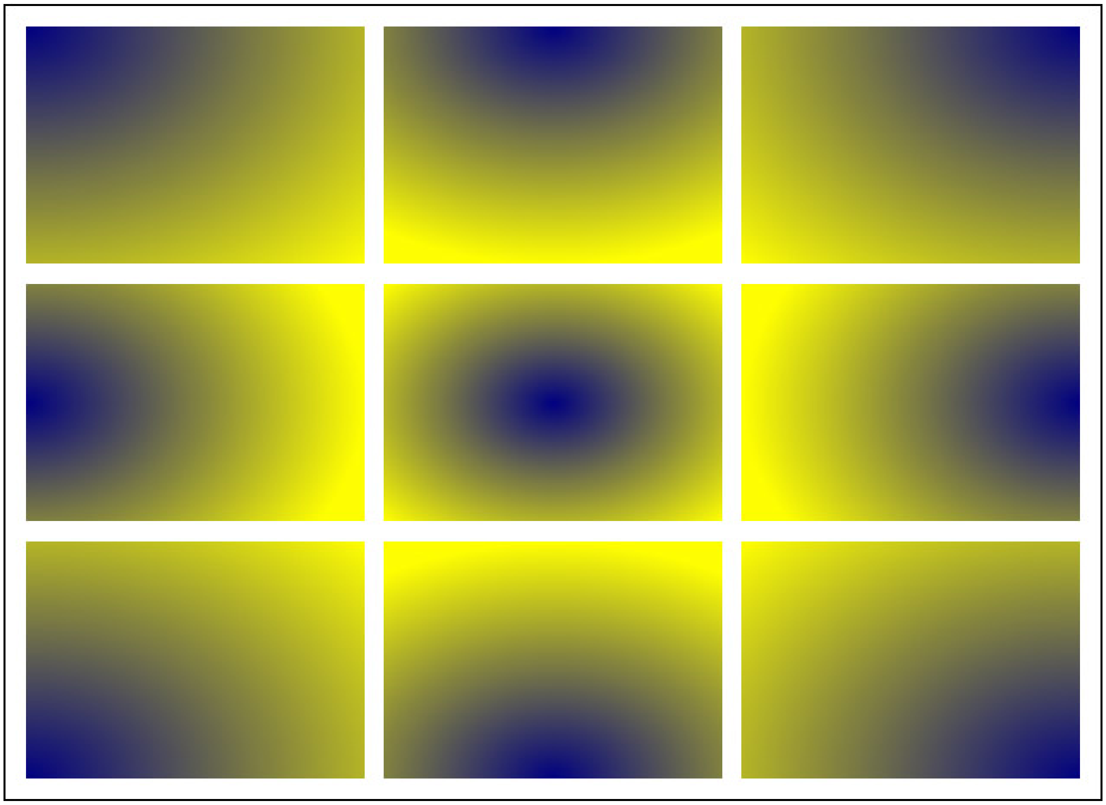
The attribute `Coords` defines a custom shading. For a linear shading, the value is a list of the coordinates of the starting and ending point, passed in an anonymous array. For a radial shading, the list includes the radii of the circles (Figure 6-12). In XML, pass it as a string of comma-separated values.

The following sequence of commands is suggested to apply a shading: 1. Build a using fill mode to 3 (path only); 2. Call `intersect` with no parameter to set clip mode; 3. Call `closePath` to begin a new path.

In the following example, we create a table to show the 9 directions of a radial shading. The center cell is filled with a shading without any direction (coaxial).

Example 6-17. Nine different directions of a radial shading.

```
use PDF;
my $doc = new PDFDoc( );
my $g = $doc->newPage( 'LetterR' )->getGraphics( );
$g->moveTo( 100, 100 );
my @cells = $g->drawGrid( 500, 360, 0,
   { 'XGrids' => 3, 'YGrids' => 3, 'BorderWidth'=> 10 } );
my @dirs = ( 'TL->BR', 'T->B', 'TR->BL', 'L->R', '', 'R->L',
   'BL->TR', 'B->T', 'BR->TL' );
for( 0..$#cells ){
   $g->setPos( $cells[$_]->left( ), $cells[$_]->bottom( ) );
   # Path only
   $g->drawRect( $cells[$_]->width( ), $cells[$_]->height( ), 3 );
   $g->intersect( ); # Confine the shading pattern in this cell
   $g->closePath( ); # Finish up current clipping path
   $g->gradFill(
     new PDFShading(
       'Radial',
       {FromColor=>'Navy', ToColor=>'Yellow', Dir=>$dirs[$_] }
     ), $cells[$_] );
}
$doc->writePDF( 'sample6-17.pdf' );
```



Chapter 7: Images

An image used in a PDF file is represented by an `ImageContent` object, built by parsing the image file data directly or from in-memory data. PDF Everywhere uses the public-domain module `Compress::Zlib` to decompress as well as to compress image data, an operation that is mandatory for image formats such as PNG and BMP.

7.1 Image file formats

JPG (Joint Picture Expert Group), GIF (Graphics Interchange Format), BMP (Windows Bitmap), PNG (Portable Network Graphics), TGA (Truevision Targa), TIF (Tagged Image Format), and PCX (ZSoft PC Paintbrush) files can be parsed, 24-bit true-color, palette-color, grayscale, and bitmap images (Table 7-1), with certain restrictions.

Table 7-1. Image formats that can be parsed.

	JPG	GIF	PNG	TIF	BMP	TGA	PCX
24-bit	√		√	√	√	√	√
256-color		√	√	√	√	√	√
16-color			√			√	
Grayscale	√	√	√	√	√	√	√
Bitmap				√	√		

Due to the wide variety of image formats, PDF Everywhere is unable to process all possible variants in each format. Images of the following formats cannot be parsed:

- Interlaced GIF or PNG files (Interlacing causes an image to be shown like a stack of scrunched copies);
- PNG, or TGA files that contain transparency and alpha-channel;
- Non-256-color GIF files;
- Multiple-image GIF, PNG, or TIF files;
- RLE (run-length) compressed BMP files;
- Compressed TIF files.

GIF images with transparency used to be processed with transparency pixels masked out. However, although it worked for Acrobat 4, it generates errors in Acrobat 5. Therefore, the treatment is now discarded and no color-masking is implemented for GIF files (and PNG files as well). The parsing of PNG files is not stable and has to be improved in the future.

To create an `ImageContent` object, use this function:

```
PDFDoc::newImage( $filename: string, $width: size, $height: size,
    $attr: HASH )

<Image Name="name" File="filename"
    Width="size" Height="size”  <!-- Optional -->
/>
```

The function forwards the call to the operation `new` defined in class `ImageContent` with same arguments. The `Width` and `Height` values, if not zero, designate the desired display dimension of the image, disregarding the original dimension of the image and need not to be proportional. However, you don't need to specify these values, which are automatically read from image file.

For a normal page, the attribute hash contains only two keys, see Table 7-2. Other keys are defined for in-memory images.

Table 7-2. Attributes of an image object.

Key	Type	Meaning
`Name`	String	Name for this image resource.
`Inline`	Bool	To make it an inline image

⚠ An inline image doesn't have XML implementation. Inline images are defined mainly for internal use only.

7.2 Image Manipulation

An image can be manipulated to some extent, and this manipulation affects all instances of the same image shown throughout the PDF document. The function `showImage` applies further transformations but the effects are local to that instance only.

In XML, the tags are to be enclosed within a pair of `<Image>` tags.

- **Stretch an image**

An image can be scaled proportionally or stretched to an arbitrary dimension:

```
ImageContent::scaleTo( $width: size, $height: size )
ImageContent::scaleBy( $ratio: float )

<Scale Width="size" Height="size"/>
<Scale Ratio="float"/>
```

- **Mask an image**

An image can be masked by another image, which is called "Stencil Masking" in PDF terms. The imaging acting as the stencil must be black/white bitmap. This type of masking can be applied on any image.

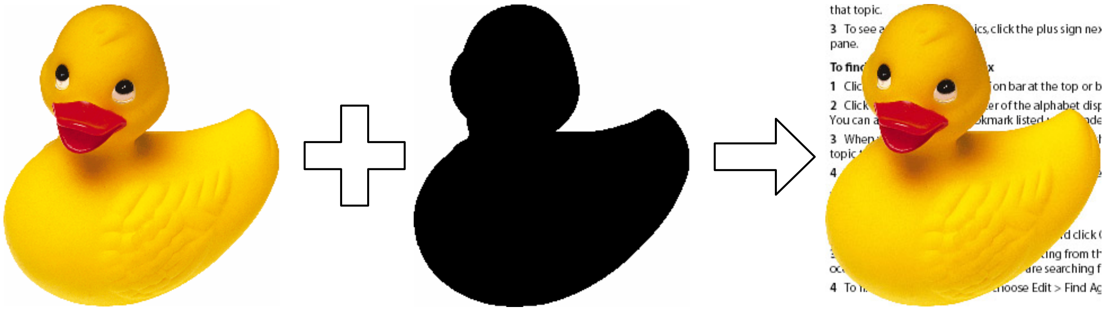
```
ImageContent::setMask( $image: ImageContent )
```

<code><Mask Image="*name*"/></code>

Alternately, an image can be color-key masked: colors fell into a given range will not be shown. You need to specify a color and a range (0 through 255). Color-key masking is known to cause problems with GIF and PNG files, although in Acrobat 4 there wasn't any problem.

```
ImageContent::setMask( $color: Color, $range: integer )
```

<code><Mask Color="*color*" Range="*integer*"/></code>


that topic.
3 To see a … ics, click the plus sign ne…
pane.
To finc … x
1 Clic … on bar at the top or b
2 Click … er of the alphabet disp
You can a … okmark listed … nde
3 When …
topic …
4
… d click
… ing from th
oc … are searching f
4 To … hoose Edit > Find Ag

In the following example, we use either a color-key or a stencil masking scheme to overlay an image onto a page with white areas cut out (see illustration above).

Example 7-1. Color-key and stencil masking of an image.

```
use PDF;
my $doc = new PDFDoc( );
my $pdf = new PDFFile( 'samplepage.pdf' );
$doc->newPage( { ImportSource => $pdf, ImportPage => 1 } );
my $g = $doc->getCurrPage( )->getGraphics( );
# Import the image of a duckling
my $img = new ImageContent( 'duck.bmp' );
# Use the color-key masking or stencil masking
# $img->setMask( 'white', 10 );
# Import the stencil image (black/white bitmap)
$img->setMask( new ImageContent( 'duckmask.bmp' ) );
$g->showImage( $img, 100, 100 );
$doc->writePDF( 'sample7-1.pdf' );
```

- **Adjust the colors**

The colors of a *24-bit true-color* or *grayscale* image can be adjusted by using color mapping. A bitmap image (which uses a palette) can be adjusted by changing the colors in the palette. The function `lighten` brightens an image by a ratio between -1 and 1, with negative values meaning darkening. The function `invert` does what the name implies.

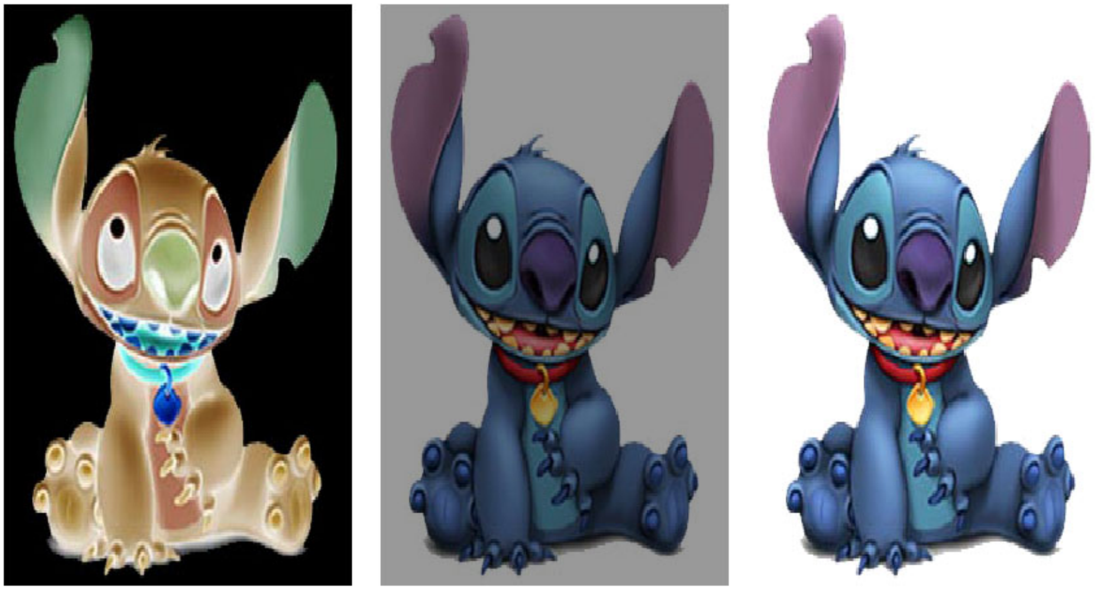
```
ImageContent::invert( )
ImageContent::lighten( $ratio: float )
```

```
<Exec Cmd="Invert "/>
<Exec Cmd="Lighten" Ratio="float"/>
```

In Example 7-2, we use an XML template to show you how to use XML tags for the above function.

Example 7-2. XML tags for manipulating an image.

```
use PDF;
$tpl = q{
<Template>
<Image Name="inverted" File="stitch.jpg">
   <Scale Width="120" Height="200"/>
   <Exec Cmd="Invert"/>
</Image>
<Image Name="darkened" File="stitch.jpg">
   <Scale Width="120" Height="200"/>
   <Exec Cmd="Lighten" Ratio="-0.4"/>
</Image>
<Image Name="masked" File="stitch.jpg">
   <Scale Width="120" Height="200"/>
   <Mask Color="white" Range="10"/>
</Image>
<Graph>
   <ShowImage Name="inverted" Left="40" Bottom="40"/>
   <ShowImage Name="darkened" Left="170" Bottom="40"/>
   <ShowImage Name="masked" Left="300" Bottom="40"/>
</Graph>
</Template>
};
my $doc = new PDFDoc( );
$doc->newPage( {'Size'=>'Postcard', 'Template'=>$tpl} );
$doc->writePDF( 'sample7-2.pdf' );
```



7.3 In-memory Image

To create a virtual image with in-memory data, the data must be passed as a `PDFStream` object that complies with PDF's requirements. In particular, image data should observe byte boundaries. For example, for an indexed image, if the product of pixel width times bits per component is not evenly divided by 8, padding is required (the padded bytes can be arbitrary). For true color data, the component sequence is R, G, B.

In-memory image has no XML implementation.

To create an in-memory image, use the function `newImage` with a different form:

```
PDFDoc::newImage( $data: PDFStream, $width: size, $height: size,
    $attr: HASH )
```

The attributes can take keys listed in Table 7-3, in addition to those listed in Table 7-2.

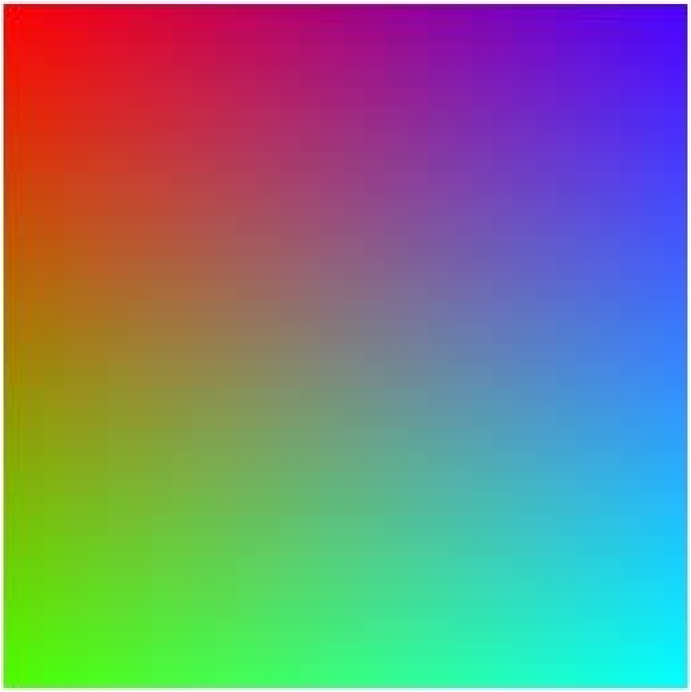
Table 7-3. Additional attributes of an in-memory image.

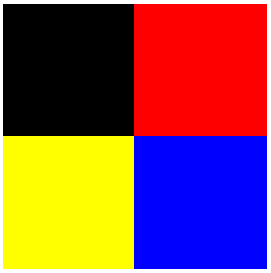
Key	Type	Meaning
`Type`	String	Image type, must be `RGB`, `CMYK`, `Gray`, or `Indexed`.
`BitsPerComponent`	Integer	Number of bits per component (pixel or index).
`Colors`	ARRAY	A list of colors for the palette of an indexed image.
`ColorTable`	String	Byte values of the palette in the order of $r_1g_1b_1r_2g_2b_2$…

We present two examples for in-memory images. Example 7-3 shows the building of a true-color image with RGB data. First we make a 65535-element array for the pixels of a 256 by 256 image, then for each pixel, we calculate the RGB values. In Example 7-4, a 4-color indexed image is built, where each component needs only two bits (so we will pack for indices into one byte).

Example 7-3. In-memory RGB true-color image.

```
use PDF;
my $doc = new PDFDoc( );
my @colors = ( chr( 0 ) ) x 65536;
for my $i ( 0..255 ){
   for my $j ( 0..255 ){
     $colors[ $i * 256 + $j ] = chr( 255 - int( sqrt($i*$i + $j*$j)
         / 1.414 ) ) . chr( $i ) . chr( $j );
   }
}
my $img = new ImageContent( new PDFStream( join( '', @colors ) ),
   256, 256, { BitsPerComponent => 8, Type => 'RGB' } );
$doc->newPage( )->getGraphics( )->showImage( $img, 50, 50 );
$doc->writePDF( 'sample7-3.pdf' );
```





Example 7-4. In-memory indexed-color image.

```
use PDF;
my $doc = new PDFDoc( );
my $img = new ImageContent( new PDFStream( ( "\x00\x55" x 4 ) .
   ( "\xAA\xFF" x 4 ) ),
   8, 8, {
     BitsPerComponent => 2,
     Type => 'Indexed',
     Colors => [ 'black', 'red', 'yellow', 'blue' ],
   } );
$doc->newPage( )->getGraphics( )->showImage( $img, 50, 500,
   { Scale => 10 } );
$doc->writePDF( 'sample7-4.pdf' );
```

When dealing with an indexed-color image, note that only 1-, 2-, 4-, 8-bit indices can be used. Also, each row of image pixels must starts at a byte boundary, that is, the total bits of the indices for a row must be multiplication of 8. For example, if the value for bits-per-component is 4, yet a row has 35 pixels, then 4 bits of zero must be padded.

7.4 Inline Image

Small images (4KB or less) can be included directly in a content stream of a page or, more frequently, an annotation such as a form button with an image on it.

To create and use an inline image, insert key Inline to the attribute hash for `newImage` and call the function `showInlineImage` on a `GraphContent` object in stead of `showImage`. This approach is very inflexible and should not be used often.

Chapter 8: Annotations and Form Fields

An annotation is an object that can be considered a small component floating above the paper, occupying a rectangular region and optionally linked to a comment. A form field is a special type of annotation that adds more interactive and the data can be submitted.

8.1 Annotations

The following types of annotations are implemented in this version of PDF Everywhere.

- `Link`: Specifies a URI hyperlink (site or email address).
- `Stamp`: Specifies a rubber stamp, which shows a predefined seal pattern.
- Markup annotations (`StrikeOut`, `Highlight`, and `Underline`): Can be understood as specifying a corresponding special text format.
- `Line`: A straight line drawn as a diagonal of a given rectangle.
- `FileAttachment`: Shows a small icon on page, which is linked to a file embedded in the PDF file. You can launch a program to open it or save it to disk.
- `Text`: Shows a small icon. When clicked, a pop-up window is shown to display notes.
- `Popup`: Shows the contents of another annotation. This type of annotation is *internally* created and should not be used directly in your programming.
- `FreeText`: Shows an editable free text note.

A text box can automatically generate `Line`, `Link`, `Highlight`, `StrikeOut`, and `Underline` annotations from inline markers.

Function `newAnnot` defined in class `PDFDoc` is used to create an annotation, which forwards the call to function `new` defined in class `Annot`. You need to pass a `Rect` and a type name (one of the names listed above), and attributes are passed via a hash. The annotation is always added to current page, unless the `OnPage` attribute is defined, whose value is either the name or a reference to the `Page` object.

```
PDFDoc::newAnnot($rect: Rect, $type: String, $attr: HASH );

<Annot
    Name="string"
    Rect="x1, y1, x2, y2"
    Type="string"
    OnPage="name" <!-- Name of the Page object -->
    <!-- More attributes here -->
/>
```

The attributes are listed in Table 8-1.

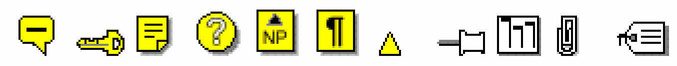
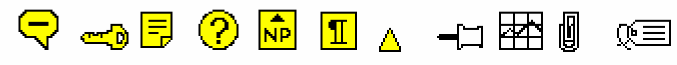
In XML, the tag can be used anywhere after the referred page has been created. Normally they should be place just before the ending tag `</Page>` of the hosting page or even the ending tag `</PDF>` of the document.

Table 8-1. Attributes of an annotation.

				Annotation Types									
	Key	**Type**	**Meaning**	`FileAttachment`	`FreeText`	`Highlight`	`Line`	`Link`	`Popup`	`Stamp`	`StrikeOut`	`Text`	`Underline`
Flags	`Hidden`	Bool	Hide and disable it.	●	●	●	●	●	●	●	●	●	●
	`NoPrint`	Bool	Do not print it.	●	●	●	●	●	●	●	●	●	●
	`NoZoom`	Bool	Do not zoom it.	●	●	●	●	●	●	●	●	●	●
	`NoRotate`	Bool	Do not rotate it.	●	●	●	●	●	●	●	●	●	●
	`NoView`	Bool	Do not display but still enable the interaction.	●	●	●	●	●	●	●	●	●	●
	`ReadOnly`	Bool	Disable interaction.	●	●	●	●	●	●	●	●	●	●
Identity	`Name`	String	The name for the object.	●	●	●	●	●	●	●	●	●	●
	`OnPage`	Name	Name or reference to the parent `Page` object.	●	●	●	●	●	●	●	●	●	●
	`Title`	String	Window title.	●	●				●	●		●	
Appearance	`Color`	Color	Icon/line/font color.	●	●	●	●	●	●	●	●	●	●
	`Width`	Integer	Line width.			●	●	●			●		●
	`Dash`	String	Dash pattern.			●		●			●		●
	`BgColor`	Color	Background color.		●								
	`BorderWidth`	Integer	Border width.		●								
	`FontFace`	String	Font name.		●					●			
	`Align`	String	`Left`, `Center`, `Right`		●								
	`SkewAngle`	Float	Skew angle of text.							●			
	`SkewDir`	String	Skew direction (`X` or `Y`).							●			
	`Highlight`	String	Highlight behavior.					●					
	`IconName`	String	Icon name (Figures 8-2,3).	●						●		●	
Miscellaneous	`UseAdobeIcon`	Bool	Do not generate icon.	●						●		●	
	`Contents`	String	Text to be shown in pop-up or free text window.	●	●					●		●	
	`PopupRect`	Rect	Area for the pop-up.	●						●		●	
	`Open`	Bool	Whether the pop-up is open by default.	●						●		●	
	`File`	String	Name of the attached file.	●									
	`MimeType`	String	Content MIME type.	●									
	`CheckSum`	Bool	Compute the checksum.	●									
	`URI`	String	Email adderss or URI					●					

`Text`, `Stamp`, and `FileAttachment` annotations are displayed as icons; when clicked, a pop-up window (the Popup annotation) appears to show comments, with a specific background color, a title and date information. PDF Everywhere is able to draw these icons (Figure 8-2) using in-memory images; this behavior is disabled when `UseAdobeIcon` is set.

Table 8-2. Icons for Text and FileAttachment annotations.

Text annotation							FileAttachment				
`Comment`	`Key`	`Note`	`Help`	`NewParagraph`	`Paragraph`	`Insert`	`PushPin`	`Graph`	`PaperClip`	`Tag`	`IconName`
											**Acrobat default**
											**PDF Everywhere**

Stamps created by Acrobat has predefined appearances and colors. The attribute IconName is used to select one of these stamps, with the following values and the samples are shown in Figure 8-1:

> `Approved`, `AsIs`, `Confidential`, `Departmental`, `Draft`, `Expired`, `Final`, `ForComment`, `ForPublicRelease`, `NotApproved`, `NotForPublicRelease`, `Sold`, `TopSecret`.

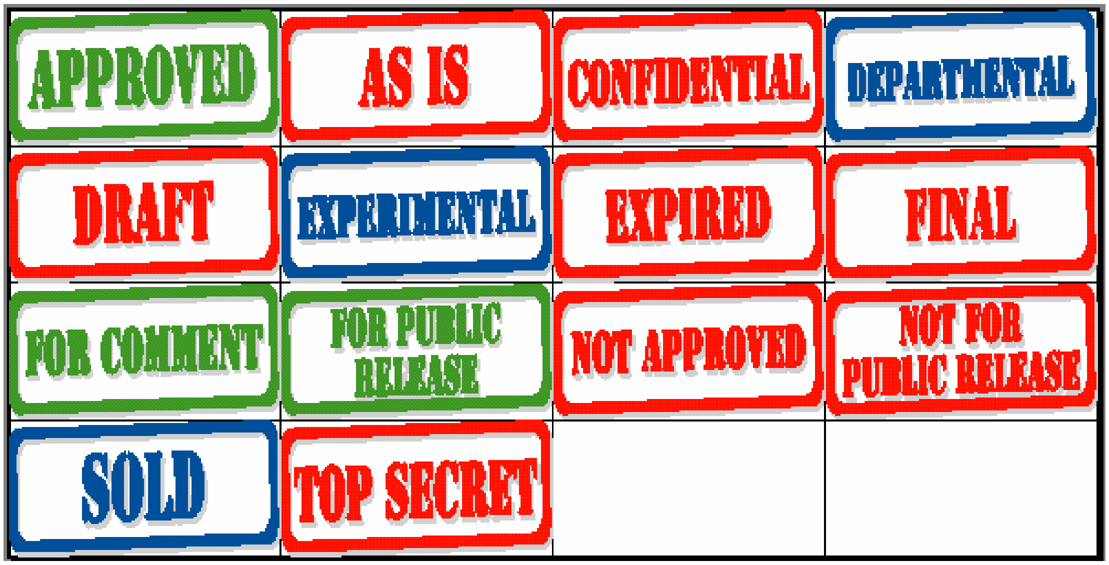


Figure 8-1. Illustration of Acrobat built-in stamp annotations.

- **Popup**

  A `Popup` annotation is associated with a `Text`, `Stamp`, or `FileAttachment` annotation to show the optional comments. Its location and default open state are determined by the parent annotations' `PopupRect` and `Open` attributes. Plus, it is set to be non-zoomable, non-rotatable, and non-printable.

- **Link**

  A clickable area on page, linked to a destination in the same document (not implemented in this version) or a hyperlink address (email or URL, passed via attribute `URI`). It is not related to any underlying text or graphics although it might seem to be. The annotation is shown as a rectangle, with a customizable border. When clicked, the pixels in the region will be inverted by default. This behavior can be changed, by set the Highlight attribute to `Invert` (default), `Push`, `Outline`, or `None`.

  The function `GraphContent::showImage` can automatically generate a borderless link annotation if the image's `Href` attribute is set.

- **Line**

  A `Line` is just a straight line, which is drawn as a diagonal of the given `Rect`, with specified color and width. This type of annotation may not be as useful as others.

- **Stamp**

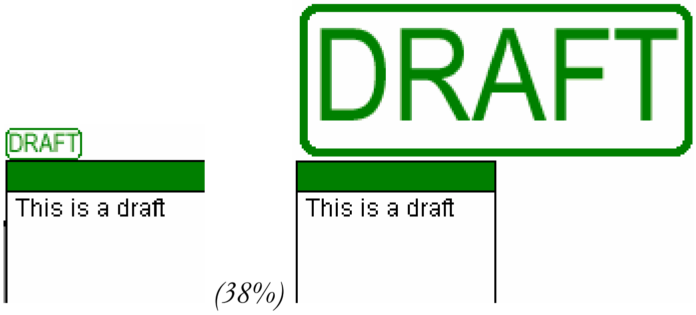
  Acrobat defined 14 `Stamp` names (Figure 8-1). PDF Everywhere is able to draw them with custom font, and the text can be changed (see the attribute `IconName` to an arbitrary string). The text can also be skewed by an angle along X or Y direction.

  The stamps are shown as stretched text within a rounded rectangle (fixed width). Font size is undefined; the text automatically fits in the given box, and is surrounded by a rounded rectangle with a width of 2 points and a corner radius of 4 points. The text is not reflowed.

  The following sample code shows a stamp with an open note (not zoomable).

Example 8-1. Stamp annotation with an open popup note.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( )->getGraphics( );
new Annot( new Rect( 100, 80, 200, 120 ), 'Stamp', {
   'Color'=>'Green', 'IconName' => 'Draft',
   'Contents'=>'This is a draft',
   'Open'=>1 } );
$doc->writePDF( 'sample8-1.pdf' );
```



This is a draft

This is a draft

(38%) (200%)

- **Markup**

  Markup annotations, including `StrikeOut`, `Highlight`, and `Underline`, are normally used as text markup tools, generated automatically by text boxes. The advantage is that the position and size of these annotations are automatically adjusted to match the actually location of text no matter the how the alignment, font style, and text direction change.

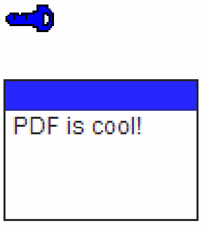
- **Text**

  This type of annotation is used to add a text note (read-only). An icon is shown at specified position, and you can choose to use Adobe's built-in icon, although Acrobat (full version) would be needed to display the icon correctly.

  Example 8-2 shows how to create a text annotation. The `Rect` passed to the function `newAnnot` has zero width and height, as only the top-left corner of the `Rect` is used (and the icon is aligned to this point by its *bottom-left* corner); the icon is always displayed in fixed dimension. The values for `IconName` are listed in Table 8-2.

Example 8-2. Text annotation shown as a key icon.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newPage( new Rect( 0, 0, 400, 150 ) )->getGraphics( );
$doc->newAnnot( new Rect( 100, 100 ), 'Text', {
   'Color'=>'Blue', 'IconName' => 'Key',
   'Contents'=>'PDF is cool!',
   'PopupRect'=> new Rect( 100, 40, 200, 100 ),
   'Title'=>'My comment',
   'Open'=>1,
} );
$doc->writePDF( 'sample8-2.pdf' );
```



PDF is cool!

- **File attachment**

  A file can be embedded into the PDF file with this type of annotation; when the icon is clicked, the file is launched by an external program. To save it to disk, right-click it then chose "save" from the popup menu. This annotation is active only with Acrobat but *NOT* the Reader.

A file has a MIME type. The program is able to determine most MIME types by looking at file name extension, such as '`application/pdf`' for PDF files. If your file type is listed in Table 8-3, then attribute `MimeType` is not needed. A checksum of the file data can be computed to verify the integrity, if the attribute `CheckSum` is set to 1.

Table 8-3. MIME types for common file name extensions.

Extension	MIME type	Extension	MIME type
`aiff`	*audio/x-aiff*	`pdf`	*application/pdf*
`au, snd`	*audio/basic*	`pgm`	*image/portable-graymap*
`avi`	*video/avi*	`pic,pict`	*image/pict*
`bmp`	*image/bmp*	`png`	*image/png*
`css`	*text/css*	`pnm`	*image/portable-anymap*
`doc`	*application/msword*	`ppm`	*image/portable-pixmap*
`dwf`	*drawing/x-dwf*	`ppt`	*application/vnd.ms-powerpoint*
`dwg`	*application/acad*	`qt`	*video/quicktime*
`dxf`	*application/dxf*	`ra,rm,ram`	*audio/x-pn-realaudio*
`eps,ps`	*application/postscript*	`rgb`	*image/rgb*
`fdf`	*application/vnd.fdf*	`rtf`	*application/rtf*
`fh`	*image/x-freehand*	`shtml`	*text/x-server-parsed-html*
`gif`	*image/gif*	`swf`	*application/x-shockwave-flash*
`gz`	*application/x-gzip*	`tar`	*application/x-tar*
`hqx`	*application/mac-binhex40*	`tex`	*application/x-tex*
`hta`	*application/hta*	`text,txt`	*text/plain*
`htm,html`	*text/html*	`tgz`	*application/x-compressed*
`ico`	*image/x-icon*	`tif,tiff`	*image/tiff*
`jpeg,jpg`	*image/jpeg*	`uin`	*application/x-icq*
`js`	*application/x-javascript*	`vrml`	*x-world/x-vrml*
`m3u`	*audio/x-mpegurl*	`wav`	*audio/wav*
`mdb`	*application/msaccess*	`xls`	*application/vnd.ms-excel*
`mid`	*audio/midi*	`xml`	*text/xml*
`mp3`	*audio/x-mpeg*	`xsl`	*text/xsl*
`mpeg,mpg`	*video/mpeg*	`z`	*application/x-compress*
`pbm`	*image/portable-bitmap*	`zip`	*application/x-zip-compressed*

⚠ This type of annotation can not be created with XML, for security reasons.

- **Free text**

A `FreeText` annotation is like a `Text` annotation but it doesn't have open or close states. The text is always visible, in stead of being display in an additional popup window. The font face, size, color, and background color can be set. The font color is always the same as the border color.

8.2 Acrobat Form

A form, represented by an `AcroForm` object, is used to manage *all* form fields in a PDF document. The form has no visible appearance; it is used solely as a structural container for all the form fields, which are specialized annotations called "widgets".

Form data can be submitted using POST, GET, FDF, PDF or XML method.

In Perl, the form must be created before any form fields are created, using the function `newForm` defined in `PDFDoc`, which forwards the call to `AcroForm::new`. In XML, the `<Form>` tag and all its enclosed tags for form fields must be place at the end of document, just before the ending `</PDF>` tag.

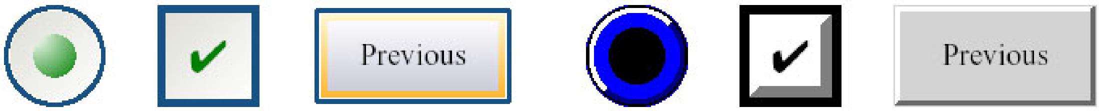
```
PDFDoc::newForm( $url: string, $attr: HASH );
```

```
<Form
    Url="string"
    FontFace="fontname"
    Theme="WinXP|Default"
    >
<!-- Tags for form fields hosted here -->
</Form>
```

The `Url` is the address of a web service that handles data submitted by this form. The attributes are listed in Table 8-4. Checkbox, radio button, and push button can show a different appearance in the `Theme` is set to `WinXP`, as shown in Figure 8-2.

Table 8-4. Attributes for an Acrobat form.

Key	Type	Meaning
`FontFace`	String	Default font face for form fields (default is `Helvetica`). Must be one of the 12 Adobe Acrobat built-in font names (Table 5-1, excluding `ZapfDingbats` and `Symbol`)
`Theme`	String	Appearance theme for form fields. `WinXP` or `Default`.



Theme = WinXP *Theme = Default*

Figure 8-2. Appearances of form fields affected by theme.

8.3 Form Fields

Form fields, defined in class `Field`, are special annotations. Eight types of fields are defined: buttons (`Button`, `Radio`, and `CheckBox`), text inputs (`Text`, `TextArea`, and `Password`), and drop-down lists (`List` and `Combo`), similar to HTML `<Input>` and `<Select>` components. The creation is similar to creating an annotation:

```
PDFDoc::newField($rect: Rect, $type: String, $attr: HASH );

<Field
    Name="string" Rect="x1, y1, x2, y2" Type="string"
    OnPage="name" Value="string"
    <!-- More attributes here -->
/>
```

Table 8-5. Common attributes of form fields.

				Form field types							
	Key	**Type**	**Meaning**	`Button`	`Text`	`TextArea`	`Password`	`Radio`	`CheckBox`	`List/Select`	`Comno`
Flags	`Hidden`	Bool	Hide and disable it.	•	•	•	•	•	•	•	•
	`NoPrint`	Bool	Do not print it.	•	•	•	•	•	•	•	•
	`NoZoom`	Bool	Do not zoom it.	•	•	•	•	•	•	•	•
	`NoRotate`	Bool	Do not rotate it.	•	•	•	•	•	•	•	•
	`NoView`	Bool	Do not display but still enable the interaction.	•	•	•	•	•	•	•	•
	`ReadOnly`	Bool	Disable interaction.	•	•	•	•	•	•	•	•
Identity	`Name`	String	The name for the object.	•	•	•	•	•	•	•	•
	`OnPage`	Name	Name or reference to the parent `Page` object.	•	•	•	•	•	•	•	•
	`Tips`	String	A message showed when mouse cursor rolls over the field.	•	•	•	•	•	•	•	•
Data	`Value`	String	Export value as a string.	•	•	•	•	•	•	•	•
	`NoExport`	Bool	If set, the value is not submitted.		•	•	•	•	•	•	•
	`Required`	Bool	If set, value can't be empty.		•	•	•	•	•	•	•
Appearance	`FontFace`	String	Must be one of the built-in Type-1 fonts. Default: `Times-Roman`	•	•	•	•			•	•
	`FontSize`	Float	Fone size in points.	•	•	•	•	•	•	•	•
	`FontColor`	Color	Default: black.	•	•	•	•	•	•	•	•
	`BgColor`	Color	Background color.	•	•	•	•	•	•	•	•
	`BorderColor`	Color	Default: black.	•	•	•	•	•	•	•	•
	`BorderDash`	String	`Dashed`, `Dotted`, or `DashDot`.	•	•	•	•	•	•	•	•
	`BorderWidth`	Integer	Border width in points.	•	•	•	•	•	•	•	•
	`BorderStyle`	String	Border style name.	•	•	•	•	•	•	•	•

Attributes for form fields are listed in Table 8-5. Moreover, each specific field type has its own additional attributes. A field must have a name and an associated value, so that the form data can be correctly submitted.

The appearance of a field is highly customizable. A field can have a border style of `None`, `Solid`, `Dashed`, `Beveled`, `Inset`, or `Underline` (Figure 8-3). By default a push button has a beveled border, while other types of fields have a solid borderline. Figure 8-3 also shows the effect of mouse-over events and the definition of border.

When the `WinXP` theme is used, buttons are given special appearances, and the settings for `BorderColor`, `BorderWidth`, and `BgColor` will no take normal effect.

The program draws the fields using vector graphics functions. If the value of a field is changed, the viewer application (such as Acrobat) may redraw the field, causing a slightly variation of the appearance. This is normal.

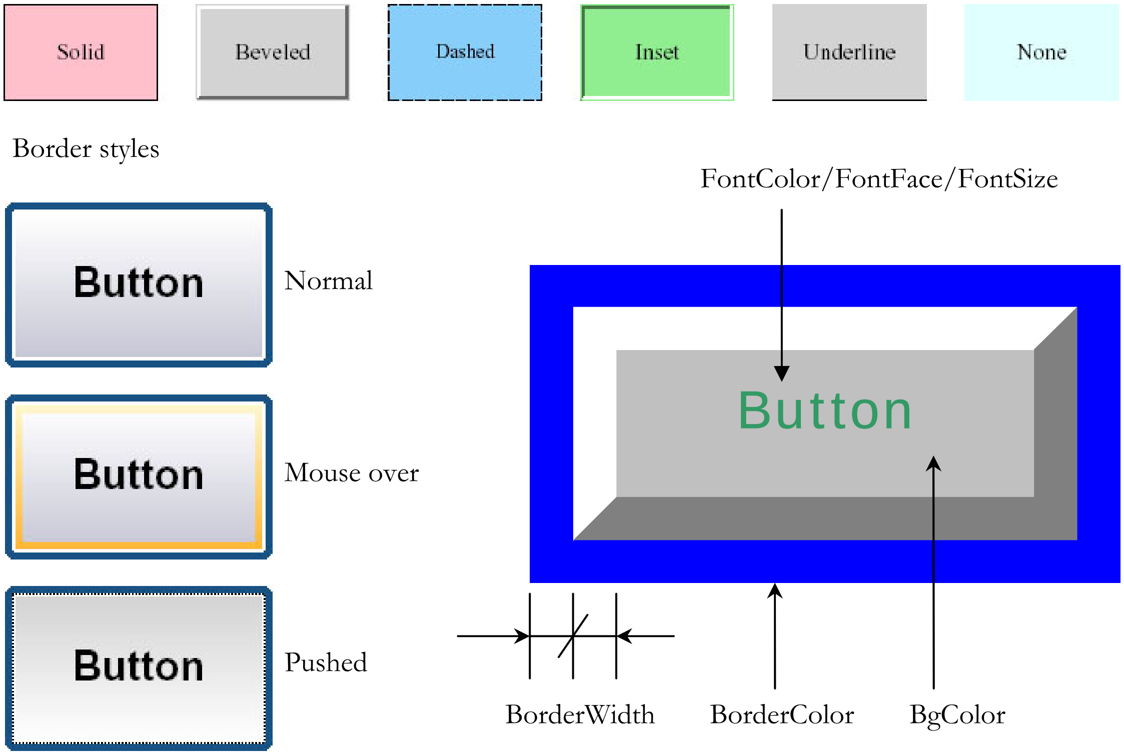


Figure 8-3. Appearances and attributes of form fields.

- **Push button**

A push button, similar to an HTML component `<Input type="Button">`, is used to submit a form or execute an action. It has three states as shown in Figure 8-3. When pushed (clicked), it is said to be "highlighted", and the highlight mode can be chosen from `Outline` (outline inverted), `Invert` (entire button inverted), and `None`. When the border style is `Beveled`, the highlighted mode gives most realistic effect because the border is changed to `Inset` when pressed. If the border is already `Inset` in normal state, the background color is darkened, as illustrated in Table 8-6.

A push button can have many attributes, in addition to those listed in Table 8-5. These attributes are listed in Table 8-7 and 8-8. Many attributes are used only when certain type of action is associated with the button.

Table 8-6. Highlight modes for a push button.

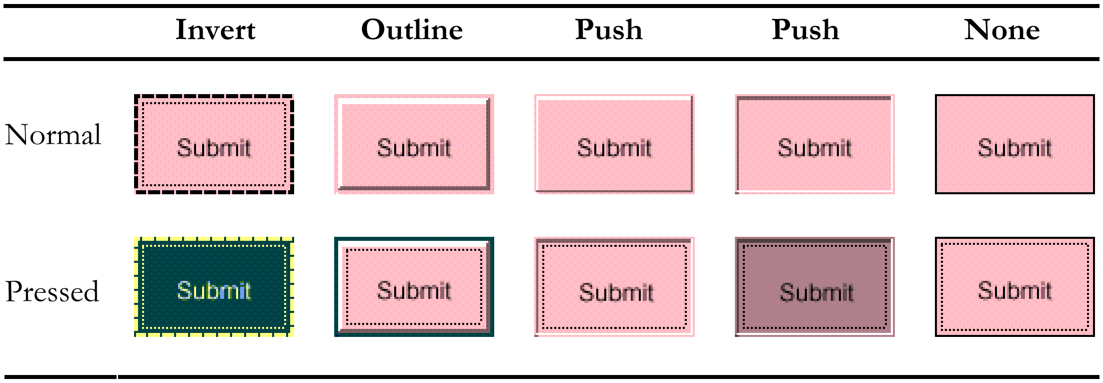


Table 8-7. Additional attributes for a push button.

Key	Type	Meaning
`Caption`	String	Button caption. Always shown on one line. If not defined, its `Name` is used in stead.
`Highlight`	String	Must be `Invert`, `Outline`, `Push` (default), or `None`.
`Icon`	String	Graphics file name.
`IconPos`	String	Icon position, see Figure 8-4.
`Action`	String	Action of the button (*cf.*.Table 8-8).

At present, a button can be associated with one of the following actions:

- `Submit` – Submit data of all form fields excluding those whose `NoExport` attribute is set. You can choose to submit the data in one of the various forms.
- `Reset` – Reset form fields to initial states.
- `Goto` – Jump to another page in the same document or to another file. The page number is needed, with page number starts from 0. If the same document, the destination `Page` can be the `Page` object (in Perl) or its name (in XML).
- `SendMail` – Send the file via email.
- `Print` – Print the file.
- `FlipPage` – Flip to first, previous, next, or last page.
- `JS` – Execute a piece of JavaScript code.

More actions will be added in the future. The attributes needed for different types of actions are listed in Table 8-8.

Table 8-8. More attributes for a push button.

			Action types						
Key	**Type**	**Meaning**	`Submit`	`Rest`	`SendMail`	`FlipPage`	`Print`	`GoTo`	`JS`
`FDF`	Bool	Submit form data in FDF format.	•						
`Get`	Bool	Use Get method in stead of Post.	•						
`Coord`	Bool	Submit coordinates as well.	•						
`XFDF`	Bool	Submit form data in XML.	•						
`AsPDF`	Bool	Submit form as a PDF file.	•						
`Annots`	Bool	Submit annotation data as well.	•						
`File`	String	Destination file name.						•	
`Page`	Integer	Destination page number.						•	
`Dest`	String	Destination page. Must be <<, <, >, or >> to denote first, previous, next, and last pages.				•			
`JS`	String	JavaScript code to be executed.							•

A button can have an optional icon shown on it, which can be placed at center, top, bottom, left or right positions, or resize to fit the width or height, or even stretch to cover the entire region. In this case, the button is similar to an image map used in an HTML file (and the coordinates of the clicked point can be submitted as well). Shown in Figure 8-4 are examples of these positioning methods. The values for `IconPos` are: `Left`, `Top`, `Right`, `Bottom`, `Stretch`, `Center`, and `Fit`. For the first four types, the region covering both the image and the text is aligned to the center of the button.

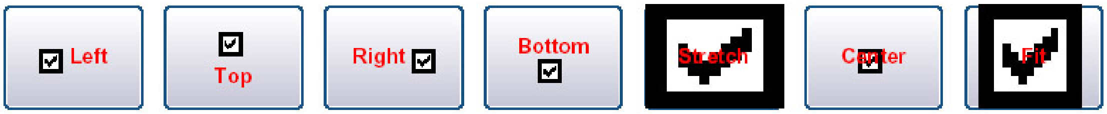


Figure 8-4. Placement of an image in a push button.

In Example 8-3, we create the buttons shown in Figure 8-4. We use the table-based layout scheme of the page: first create a 7 × 1 grid, and then fill in a button for each cell using a different position for the icon.

Example 8-3. Program used to create image buttons in Figure 8-4.

```
use PDF;
my $doc = new PDFDoc( );
$doc->newForm('http://foo.com/cgi-bin/bar.cgi', {Theme=>'WinXP'});
my $page = $doc->newPage( { Size => 'LetterR',
    Paddings => [ '1in', '1in' ] } ); # Change the art box
$page->drawTable( '9in', '1in', 0, { XGrids => 7, YGrids => 1,
    BorderWidth => 10, NoDraw => 1 } );
my @positions = qw(Left Top Right Bottom Stretch Center Fit);
for (1..7){
    my $pos = shift( @positions );
    $doc->newField( $doc->getCell( 1, $_ ), 'Button', {
        Name => 'btn' . $_,
        Caption => $pos,
        Icon => 'check.bmp',
        IconPos => $pos,
        FontColor => 'red',
        FontFace => 'Helvetica-Bold'
    } );
}
$doc->writePDF( 'sample8-3.pdf' );
#$doc->exportXML( ); # Uncomment to get XML output.
```

In the XML output of the same program, the form fields are directly placed on page with correct positions and sizes. The theme uses certain shadings, which are automatically created and assigned predefined names (to avoid repeated creation of same shadings). The form definition is place at the end of document, with nested field definitions.

```
<PDF>
   <Shading Type="Linear" Name="WinXP_Normal" Centered="0"
FromColor="C6C6D6" ToColor="White" ColorSpace="RGB" Coords="0, 0, 0, 1" />
   <Shading Type="Linear" Name="WinXP_Down" Centered="0"
FromColor="LightGrey" ToColor="White" ColorSpace="RGB" Coords="0, 1, 0, 0"
/>
   <Shading Type="Linear" Name="WinXP_Halo" Centered="0" FromColor="FFF7CE"
ToColor="FFB531" ColorSpace="RGB" Coords="0, 1, 0, 0" />
   <Page Name="N000" Width="792" Height="612" Size="LetterR">
     <Graph Rect="72, 72, 792, 612" Name="N001"></Graph>
   </Page>
   <Form Url="http://foo.com/cgi-bin/bar.cgi" FontFace="Times-Roman"
Theme="WinXP" >
     <Field Type="Button" Name="btn1" OnPage="N000" Rect="77, 77, 159.5714,
139" FontFace="Helvetica-Bold" FontColor="red" Caption="Left"
Icon="check.bmp" IconPos="Left" />
     <Field Type="Button" Name="btn2" OnPage="N000" Rect="169.5714, 77,
252.1429, 139" FontFace="Helvetica-Bold" FontColor="red" Caption="Top"
Icon="check.bmp" IconPos="Top" />
     <Field Type="Button" Name="btn3" OnPage="N000" Rect="262.1429, 77,
344.7143, 139" FontFace="Helvetica-Bold" FontColor="red" Caption="Right"
Icon="check.bmp" IconPos="Right" />
     <Field Type="Button" Name="btn4" OnPage="N000" Rect="354.7143, 77,
437.2857, 139" FontFace="Helvetica-Bold" FontColor="red" Caption="Bottom"
Icon="check.bmp" IconPos="Bottom" />
     <Field Type="Button" Name="btn5" OnPage="N000" Rect="447.2857, 77,
529.8571, 139" FontFace="Helvetica-Bold" FontColor="red" Caption="Stretch"
Icon="check.bmp" IconPos="Stretch" />
     <Field Type="Button" Name="btn6" OnPage="N000" Rect="539.8571, 77,
622.4286, 139" FontFace="Helvetica-Bold" FontColor="red" Caption="Center"
Icon="check.bmp" IconPos="Center" />
     <Field Type="Button" Name="btn7" OnPage="N000" Rect="632.4286, 77,
715, 139" FontFace="Helvetica-Bold" FontColor="red" Caption="Fit"
Icon="check.bmp" IconPos="Fit" />
   </Form>
</PDF>
```

- **Text, Password, and TextArea**

These fields are similar to HTML elements `<Input type="Text">`, `<Input type="Password">`, and `<TextArea>`. The characters will be shown as "*" in a `Password` field. The `Text` field always shows its value on a single line, while `TextArea` uses multiple lines. These fields can use any of the border and background styles.

These fields display the value as the text content. `Text` and `Password` has an additional attribute, `MaxLen`, whose value is an integer, specifying the max number of characters the value should be.

If `FontSize` is set to 0, the text in a Text field is automatically resized to fit the available height of the field. If `FontSize` is not present, then 12 point font is used.

- **Radio and CheckBox**

A radio button is always drawn in a round shape, and a checkbox is always a square. Both can have custom border and background styles.

A symbol can be used on a radio button or a check box, using `ZapfDingbats` font. The symbols, along with other attributes, are listed in Table 8-9. If a symbol is not defined, the radio button will show a solid circle in checked state and the check box will show two diagonal straight lines.

Table 8-9. Additional attributes for a radio button or a check box.

Key	Type	Meaning			
`NoToggleOff`	Bool	In set, a `Radio` field doesn't turn off once checked.			
`Checked`	Bool	If set, the field is checked by default.			
`Symbol`	String or character	Name (listed below) of the symbol to be displayed, or any other characters.			
		`Diamond`	♦	`Square`	■
		`Star`	★	`Arrow`	➔
		`Check`	✓	`Hand`	☞
		`Cross`	✖	`Heart`	♥

Radio buttons of the *same name* form a group. Multiple groups can be present in a form. At any time, only one of the buttons in a group may be checked.

Similar to a `Text` field, if the `FontSize` is set to 0, the symbol will be resized to fit.

Vivid appearances can be achieved by selecting border, background, and font styles and symbols. Figure 8-5 shows a gallery of such appearances (Top-down: `Radio`, `CheckBox`, `Text`, and `List`).

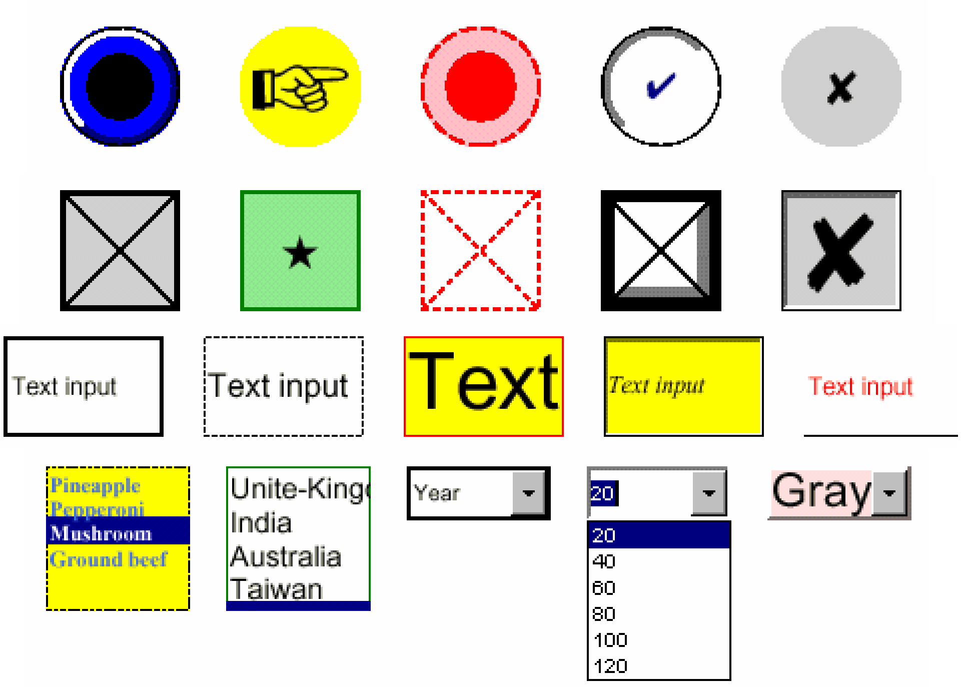


Figure 8-5. A gallery showing the various appearances fields can have.

- **List (Select) and Combo**

The PDF equivalent for HTML element `<Select>`, a `List` (also called `Select`) or a `Combo` shows a list but in `Combo` the selection box is not shown (must be drop-down). The have the additional attributes listed in Table 8-10.

Like with HTML, entry/value pairs are expected. While in HTML you can write “`<option value='Value'>Entry</option>`”, PDF Everywhere requires the `Choices` data be passed in one of the following ways:

1. ***Array of array***: [[‘Entry 1’, ‘Value 1’], [‘Entry 2’, ‘Value 2’], [‘Entry 3’, ‘Value 3’]];
2. ***Array ref***: [‘Entry 1’, ‘Entry 2’, ‘Entry 3’];
3. ***Hash ref***: { ‘Entry 1’ => ‘Value 1’, ‘Entry 2’ => ‘Value 2’, ‘Entry 3’ => ‘Value 3’ }.

In XML they are turned into `<Option>` tags.

Table 8-10. Additional attributes for a selection box.

Key	Type	Explanation
`Choices`		One of the three forms shown above.
`Edit`	Bool	If set, the box is editable. Used only for `Combo`.
`Sorted`	Bool	If set, the entries are sorted.
`SortBy`	String	How the entries should be sorted. Must be ‘`A->Z`’ (default), ‘`Z->A`’, ‘`0->9`’, or ‘`9->0`’ for sorting in ascending or descending order by ASCII or value.
`Selected`	Integer	The index of the selected entry. If negative, count backwards.

The following program was used to create the last row shown in Figure 8-5. If a value for an entry is not set, the entry is also used as the value.

Example 8-4. Passing entries for a list/combo field.

```
use PDF;
my $doc = new PDFDoc( );
my $page = $doc->newPage( );
$page->getGraphics( );
# First one uses array of array
new Field( new Rect( 50, 400, 130, 480 ), List, { Name=>'se1',
   Choices=>[ ['Pineapple', 'P'], ['Pepperoni'], ['Mushroom, M'],
     [], ['Ground beef', 'B'] ], # Empty entry is removed
   FontFace=>'Times-Bold', BorderStyle=>'Dashed',
   BorderDash=>'Dotted', BgColor=>'Yellow', FontColor=>'RoyalBlue',
   Selected=>2 } );
# Second one uses single array
new Field( new Rect( 150, 400, 230, 480 ), List, { Name=>'se2',
   Choices=>[ qw(Unite-Kingdom India Australia Taiwan Argentina) ],
   BorderColor=>'Green', Selected=>4, FontSize=>16 } );
# Third one uses hash
new Field( new Rect( 250, 450, 330, 480 ), Combo, { Name=>'se3',
   Choices=>{ Day=>1, Month=>30, Season=>90, Year=>365 },
   BorderWidth=>2, SortBy=>'Z->A', Edit=>1 } );
# Fourth one uses sorting
new Field( new Rect( 350, 450, 430, 480 ), Combo, { Name=>'se4',
   Choices=>[ qw(100 120 20 40 60 80) ], BorderStyle=>'Inset',
   SortBy=>'0->9', FontColor=>'Blue' } );
# Last one uses editing, sorting, and free font size
new Field( new Rect( 450, 450, 530, 480 ), Combo, { Name=>'se5',
   Choices=>{ qw(Red Blue Yellow Green Purple Navy White Black) },
   BorderStyle=>'Beveled', BgColor=>'MistyRose', FontSize=>0,
   Sorted=>1, Edit=>1 } );
$doc->writePDF( 's8-5.pdf' );
#$doc->exportXML( );
```

The XML output reveals the `<Option>` tags.

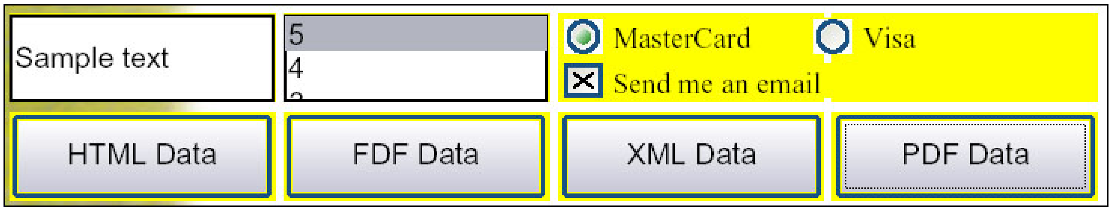
```
<Field Type="Combo" Name="se3" OnPage="N000" Rect="250, 450, 330,
480" BorderWidth="2" Sorted="1" SortBy="Z->A">
   <Option Value="365">Year</Option>
   <Option Value="90">Season</Option>
   <Option Value="30">Month</Option>
   <Option Value="1">Day</Option>
</Field>
```

- **Use in table-based layout**

In XML, you can use more attributes to place the field, without using the `Rect`: `Row`, `Col`, `RowSpan`, `ColSpan`, `Width`, `Height`, and `Anchor`. See Table 3-2.

In Example 8-5, we import a page from a PDF file, then add a form onto the bottom of the page showing a text input field, a list, a group of two radio buttons, a check box, and four submit buttons each of a different encoding scheme.

We first draw a 2 × 4 table, then draw a smaller 2 × 4 table over two cells in the first row. Three text boxes are created to show the captions of the radio buttons and check boxes – the caption is not a part of the field and must be displayed separately.



Sample text

5
4

● MasterCard ○ Visa
☒ Send me an email

Example 8-5. Lay out form fields using tables.

```
use PDF;
my $doc = new PDFDoc( );
my $pdf = new PDFFile( 'samplepage.pdf' );
$doc->newForm( "http://127.0.0.1/test/default.asp", { FontFace =>
'Helvetica', Theme => 'WinXP' } );
my $p = $doc->newPage( { ImportSource => $pdf, ImportPage => 0 } );
# Now set the fill color of the page’s default graphics layer
$p->getGraphics( )->setColor( 'yellow', 1 );
$p->drawTable( 450, 80, 1, {XGrids=>4, YGrids=>2, BorderWidth=>4} );
$doc->newField( $doc->getCell( 1, 1, 1, 1 ), 'Text',
   { Name => 'text1', Value => 'Sample text' } );
$doc->newField( $doc->getCell( 1, 2, 1, 1 ), 'List',
   { Name => 'list1', Choices => [ qw(5 4 3 2 1) ] } );
# Now draw the second table within the table
$p->drawTableAt( $doc->getCell( 1, 3, 1, 2 ), 1, {
   Widths => [ 20, '1#', 20, '1#' ], YGrids => 2,
   BorderWidth => 4, NoFrame => 1 } );
# Radio button (and checkbox ) is separated from its caption
$doc->newField( $doc->getCell( 1, 1, 1, 1 ), 'Radio',
   { Name => 'radio1', Value => 'MasterCard' } );
$doc->newTextBox( $doc->getCell( 1, 2, 1, 1 ), 'MasterCard' );
$doc->newField( $doc->getCell( 1, 3, 1, 1 ), 'Radio',
   { Name => 'radio1', Value => 'Visa' } );
$doc->newTextBox( $doc->getCell( 1, 4, 1, 1 ), 'Visa' );
$doc->newField( $doc->getCell( 2, 1, 1, 1 ), 'CheckBox',
   { Name => 'check1', Value => 'Yes' } );
$doc->newTextBox( $doc->getCell( 2, 2, 1, 3 ), 'Send me an email' );
# Undef current table and use the larger table
$doc->undefCurrTable( );
$doc->newField( $doc->getCell( 2, 1, 1, 1 ), 'Button', {
   Caption => 'HTML Data', Name => 'btn1', Action => 'Submit' } );
$doc->newField( $doc->getCell( 2, 2, 1, 1 ), 'Button', {
   Caption => 'FDF Data', Name => 'btn2', Action => 'Submit',
   FDF => 1 } );
$doc->newField( $doc->getCell( 2, 3, 1, 1 ), 'Button', {
   Caption => 'XML Data', Name => 'btn3', Action => 'Submit',
   XFDF => 1 } );
$doc->newField( $doc->getCell( 2, 4, 1, 1 ), 'Button', {
   Caption => 'PDF Data', Name => 'btn4', Action => 'Submit',
   AsPDF => 1 } );
$doc->writePDF( sample8-4.pdf' );
```

The various forms of data being submitted to the server side are listed in Table 8-11, along with the value for HTTP header `Content-Type`.

Table 8-11. Various formats of the data submitted by a same form.

Form	Content type	Data
HTML	`application/x-www-form-urlencoded`	`check1=Yes&list1=4&radio1=Visa&text1=Sample+text&btn1=`
FDF	`application/vnd.fdf`	`%FDF-1.2 %âãÏÓ 1 0 obj << /FDF << /Fields [ << /T (btn2)>> << /V /Yes /T (check1)>> << /V (4)/T (list1)>>` `......` *(See below for full listing)*

XML	`application/vnd.adobe.xfdf`	`<xfdf xmlns="http://ns.adobe.com/xfdf/" xml:space="preserve">`
		`<fields>`
		`<field name="btn3"></field>`
		`......` *(See below for full listing)*

PDF	`application/pdf`	`%PDF-1.4 %âãÏÓ 1 0 obj << /Creator (PDFeverywhere 3.0) /ModDate (D:20021201151524+08'00') /CreationDate`
		`......`

Both FDF and XFDF (XML) format are easily modifiable by the server side program, which can return the modified values in same format to the requesting browser, where Acrobat can load the file again with new set of data.

As shown in Figure 8-6, a form-containing PDF file can make use of this approach in a multi-pass interaction between a client and the server. After multiple revisions of a data set over a same form, the user can print the finished form, and the server can save the data set in XML or FDF form into database.

```
%FDF-1.2 %âãÏÓ
1 0 obj <<
   /FDF <<
     /Fields [
       << /T (btn2)>>
       << /V /Yes /T (check1)>>
       << /V (4)/T (list1)>>
       << /V /Visa /T (radio1)>>
       << /V (Sample text)/T (text1)>>
     ]
     /F (/C/pdf/abc.pdf)
     /ID [
       <5fdde3022b891fa92c553fddf4bd16e7>
       <5fdde3022b891fa92c553fddf4bd16e7>
     ]
   >>
>> endobj
trailer
<< /Root 1 0 R >>
%%EOF
```

```
<xfdf xmlns="http://ns.adobe.com/xfdf/" xml:space="preserve">
<fields>
   <field name="btn3"></field>
   <field name="check1">
     <value>Yes</value>
   </field>
   <field name="list1">
     <value>4</value>
   </field>
   <field name="radio1">
     <value>MasterCard</value>
   </field>
   <field name="text1">
     <value>Sample text</value>
   </field>
</fields>
<f href="file:///C|/pdf/abc.pdf"/>
</xfdf>
```

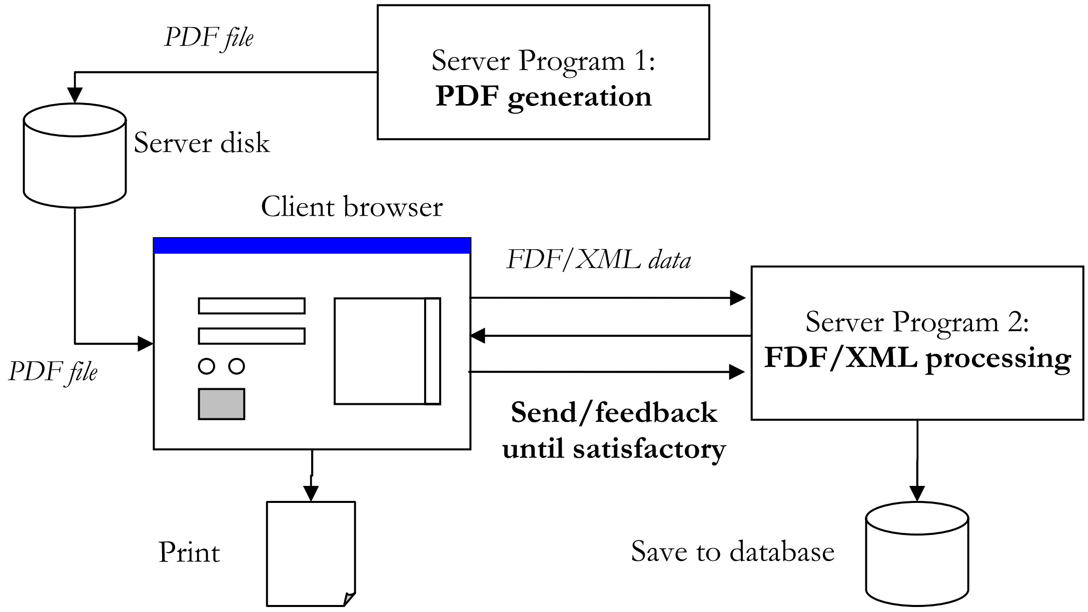


Figure 8-6. Multi-pass FDF/XML data exchange between client and server.

Chapter 9: Parsing PDF Files

A PDF file can be parsed and represented by a `PDFFile` object. The program can deal with normal, linearized, and incrementally updated PDF files. A normal PDF file contains a file header, a collection of object definitions, an Xref table, and a trailer. A linearized file has an additional trailer in the beginning portion of the file, and the trailer at file end is linked to this one. A linearized file has multiple trailers and Xref tables, and a same object may have multiple definitions that one the latest one is used.

The `PDFFile` object is able to write the entire PDF file back to the disk. However, there will always be only one Xref table and one trailer. That is, linearization is not implemented, even if the original file is linearized. For an incrementally updated file, the program discards older versions of the same object, using only the latest one, and resets the object generation number to 0.

If a file is encrypted, the program is still able to parse it. If given a correct password, the program can decode the file. Encryption at 40- and 128-bit level is also supported. The file ID is required in decryption/encryption. Without a file ID, decryption is simply impossible. The program always generates a unique ID for a file without such an ID.

9.1 Classes and Instantiation

Only the class `PDFFile` is instantiable. Each object data block defined in a PDF file is represented by an internally used class `PObject`, which manipulate raw PDF data types with classes shown in Figure 9-1 and listed in Table 9-1.

To create a `PDFFile` object, call the following function. The object is then able to supply pages to any `PDFDoc` object, encrypt/decrypt the file by itself, or make certain modifications to that file.

```
PDFFile::new( $filename: string ): PDFFile
```

The file handle is kept open until the `PDFFile` and all its `PObject` objects are destroyed.

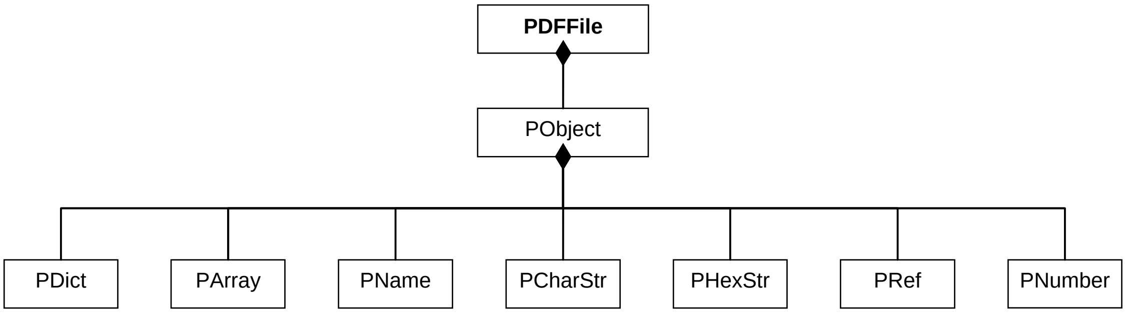


Figure 9-1. Classes used in parsing and manipulating a PDF file.

Table 9-1. Classes representing basic PDF data types.

Class	PDF Data Type	Example
`PNumber`	Numeric data	`34.5`, `-16.2`, `+123`, `0`
`PCharStr`	String literals	`(A string)`, `(Another \(string\))`
`PHexStr`	Hexdecimal strings	`<901FA3>`
`PName`	Name and Boolean	`/`, `/Name`, `/Any&*Chars`, `true`, `false`, `null`
`PRef`	Indirect object	`35 0 R`
`PArray`	Array (list)	`[ 4.3 true /test << >> ]`
`PDict`	Dictionary (hash)	`<< /Type /Page /Contents 5 0 R >>`

The program stops on the following errors:

- File doesn't start with the standard header (such as `%PDF-1.4` or `%!PS.Adobe.3.0 PDF.1.2`);
- Trailer is missing;
- Trailer is malformed (with incorrect offset values);
- File is shorter than reported by linearization data.

Each PDF object defined in the file is wrapped up in a `PObject` object, whose `Data` attribute is a reference to the parsed data structure of the original data. A `PObject` has three states: initiated, read, and parsed. The original data are not read from disk until function `readIt` is called, and not parsed until `parseIt` is called.

9.2 Access / Change File Information

- **File status**

The following functions return 1 or 0 determined by the encryption or linearization state:

```
PDFFile::getEncryptState( ): Bool
PDFFile::getLinearizeState( ): Bool
```

The following function returns the number of revisions the file has undergone:

```
PDFFile::getUpdateState( ): integer
```

The following functions returns the number of pages / PDF objects defined in the file:

```
PDFFile::getPageCount( ): integer
PDFFile::getObjectCount( ): integer
```

- **Document information**

The properties Title, Subject, Keywords, Author, Producer, and Creator of a PDF file can be read or set, using the following functions.

```
PDFFile::getTitle( ): string
PDFFile::getSubject( ): string
PDFFile::getAuthor( ): string
PDFFile::getKeywords( ): string
PDFFile::getCreator( ): string
PDFFile::getProducer( ): string

PDFFile::setTitle( $str: string )
PDFFile::setSubject( $str: string )
PDFFile::setAuthor( $str: string )
PDFFile::setKeywords( $str: string )
PDFFile::setCreator( $str: string )
PDFFile::setProducer( $str: string )
```

- **Viewer preferences**

The default open page can be set using this function (pass a page number, 0 for the first page, etc.):

```
PDFFile::setOpenPage( $page: integer )
```

The viewer preferences can be changed by the following function, where the parameter is a hash with keys `Toolbar`, `Menubar`, `WindowUI`, `FitWindow`, `CenterWindow`, each having a Boolean value (1 or 0). See Table 2-2 for the description of these keys.

```
PDFFile::setViewerPref( $attr: HASH )
```

Page mode and page layout can also be changed. Table 2-2 listed the possible values for the parameters (note that numeric values can ***not*** be used).

```
PDFFile::setPageMode( $mode: string )
PDFFile::setPageLayout( $mode: string )
```

To set transition and duration for a page, use the following function with similar attributes and values listed in corresponding section of Table 3-1 and Table 3-3.

```
PDFFile::setPageDuration( $page: integer, $dur: integer )
PDFFile::setPageTransition( $page: integer, $transtype: integer,
    $attr: HASH )
```

- **Encrypt/decrypt a file**

To protect a PDF file with 40- or 128-bit encryption, call the function `encrypt` and pass attributes in a hash, with same keys and values listed in the “Security” section of Table 2-2. A callback function can be passed as the second argument, which will be given a hash of three keys during processing, listed in Table 9-2.

To decrypt a file, pass the owner password originally used to protect the file. If owner password is not set, use the user password in stead. If the user password is also not set, use an empty string.

```
PDFFile::encrypt( $attr: HASH, $callback: CODE )
PDFFile::decrypt( $pwd: string, $callback: CODE )
```

Table 9-2. Attributes passed to the callback function via hash reference.

Key	Meaning
`ObjId`	ID of the object currently being encrypted/decrypted.
`CurrLen`	Total bytes processed (the sum of lengths of all processed objects)
`TotalLen`	Total bytes of the entire file.

It is an error to encrypt an already-encrypted file, or to decrypt a file that is not encrypted at all.

It is also an error to decrypt a file that uses an encryption scheme the program doesn't understand, for example, the scheme that `WebBuy` service uses.

- **Print file to disk**

The following function saves the PDF files to disk:

```
PDFFile::printFile( $filename: string )
```

The above description doesn't touch the internals of the parsing process and various operations used to integrate with the `PDFDoc` and other related objects such as `Page`. Normally you should just use the `PDFFile` object to supply pages to a `PDFDoc` object you are building, using the function `PDFDoc::importPages` to import frozen pages or instantiate the `Page` or `XObject` with `ImportSource` and `ImportPage` attributes to create an open page.

Index of Class Methods