The Big Faceless Report Generator

User Guide

Version 1.1.63

Introduction

Thank you for your interest in the Big Faceless Report Generator. This userguide will give you an overview of how to use the product, and start you off with some simple examples. For more detailed and up-to-date information, please visit the product homepage at http://bfo.com/products/report.

What is it?

The Report Generator is a java application for converting source documents written in XML to PDF. Build on top of our popular PDF and Graph libraries, the Report Generator combines the features of the two and wraps an XML parser around them. Thanks to this, it is now possible to create a PDF report direct from a database with **little or no Java experience**.

Features

Here's a brief summary of the generator's features

- Create dynamic reports using JSP's, ASP's, XSLT - whatever you would normally use to create dynamic HTML pages
- Simple HTML-style XML syntax makes migration for HTML reports (and HTML programmers) relatively painless
- Use Cascading Style Sheets (level 2) to control the document look-and-feel
- Build Reports on top of existing PDF documents (Extended Edition only)
- Full support for autosized, nested tables, lists, hyperlinks, images and other familar HTML features
- Inline generation of graphs and charts, in full shaded 3D!
- Embed XML metadata directly in the PDF, following Adobes XMP™ specification
- Native Unicode support. No worrying about codepages, encodings and so on, it *just works*
- Embed and subset OpenType and Type 1 fonts, or use one of the 14 latin or 8 east-asian built in fonts
- 40 and 128-bit PDF Encryption, for eyes only documents. Digital signatures too.
- Auto-pagination of content with headers, footers and watermarks
- Use ICC color profiles, spot color and patterns for sophisticated color control
- Draw barcodes and simple vector graphics directly into the document using XML elements

The generator is written in 100% pure Java and requires only JDK 1.4 or better and a SAX XML parser to run. It is supplied with three methods to create the PDF documents - a Servlet Filter, a Servlet or a Standalone application - and installs easily into any Java environment.

Getting Started

Installation

Installing the package is a simple matter of unzipping the distribution file and adding the `bforeport.jar` file to your CLASSPATH. You will also need a SAX parser - Java 1.4 and above are supplied with one, but for those forced to run older JVMs we recommend Xerces.

Several other files are supplied with the package. As well as this userguide and the API documentation under the `docs` directory, two sample applications are included in the `example` directory - a standalone XML to PDF application, and a Java Servlet. Several sample XML documents are in `example/samples`, and several dynamic samples which require a Servlet engine are under `examples/dynamic`.

Be sure to remove any previous versions of the “bforeport.jar” from the CLASSPATH, as well as the “bfopdf.jar” files from our PDF library product, otherwise exceptions during class initialization may result.

For all modern webservers, it is enough to copy the `bforeport.jar` file to the `WEB-INF/lib` directory of your web application and then set up the `WEB-INF/web.xml` file to use either the Filter or the ProxyServlet method of calling the Report Generator, depending on whether your WebServer supports version 2.3 of the Servlet Specification or not. To find out, we’d suggest trying the filter method first. If it doesn’t work, fall back to the Proxy Servlet.

Creating PDFs from Applications

The API for the report generator is extremely simple. Generally you only require three lines to be added to your program to create a PDF Report from XML.

A simple example of this is the `SampleApplication.java` example, supplied with the package in the `example` directory. To use it, first, ensure the `CLASSPATH` is set to include your SAX parser, then run the command:

```
C:\BFOREPORT\EXAMPLE> java SampleApplication samples\HelloWorld.xml
```

This creates the PDF document `samples\HelloWorld.pdf`, which you can check with your PDF viewer.

To add PDF producing code to your own package is simple. Here’s an example method which would take the URL of an XML file and an `OutputStream` to write the PDF to. The PDF specific lines are in **bold**

```
import java.io.*;
import org.faceless.report.ReportParser;
import org.faceless.pdf.PDF;

public void createPDF(String xmlfile, OutputStream out)
{
    ReportParser parser = ReportParser.getInstance();
    PDF pdf = parser.parse(xmlfile);
    pdf.render(out);
    out.close();
}
```

Creating PDFs using the Servlet 2.3 Filter

For servlet environments running the Servlet 2.3 or later environment, such as Tomcat, the recommended way to create dynamic PDF documents is using the *Filter* included in the JAR file supplied with the package. More information on filters is available from http://java.sun.com/products/servlet/Filters.html. To use it, the `WEB-INF/web.xml` file needs to be edited to map the PDF Filter to certain requests.

Here's an example `web.xml` file which maps any requests to `/pdf/*` to be run through the PDF filter. Lines specific to the PDF filter are in **bold**.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd"
>

<web-app>

  <filter>
    <filter-name>pdffilter</filter-name>
    <filter-class>org.faceless.report.PDFFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>pdffilter</filter-name>
    <url-pattern>/pdf/*</url-pattern>
  </filter-mapping>

</web-app>
```

Once this rule is added to `web.xml` and the servlet engine restarted, an XML document will be automatically converted to PDF before it is returned to the browser. For example, to convert the file `/pdf/HelloWorld.xml` to a PDF and view it in the browser, simply load the URL `http://yourdomain.com/pdf/HelloWorld.xml`.

Only files with a mime-type of `text/xml` will be processed, so images and other non-xml files in this path will be returned unaltered. See the API documentation for more detailed information.

If the XML file is being returned directly to the browser rather than being converted to PDF, this is probably caused by the mime-type not being set correctly. For dynamic XML documents like those created from JSP or CGI, the mime-type must be explicitly set by the document author. For static files, the `.xml` extension must be mapped to the `text/xml` mimetype - this is done by adding the following block to your `web.xml` file:

```
  <mime-mapping>
    <extension>xml</extension>
    <mime-type>text/xml</mime-type>
  </mime-mapping>
```

Creating PDFs using the Proxy Servlet

The other option when displaying dynamic PDFs from a Servlet is to use the *Proxy Servlet*. As the name suggests, this is a servlet which relays HTTP requests from a browser, reads the response and converts it to a PDF before sending it back to the browser.

Although the “filter” method described previously is much simpler to install and use, the proxy servlet has a couple of advantages:

- Can be used by Servlet engines supporting only the Servlet 2.2 specification
- Can proxy requests to different webservers, or even different domains - although care must be taken when doing this, as session information may not be passed on.

The disadvantages are mainly that it requires the abstract `PDFProxyServlet` servlet to be extended and the `getProxyURL` method implemented - so you have to write some code before you can use it. Also, the current version doesn’t support the POST method for proxying requests.

An example proxy servlet called `SampleServlet.java` is supplied with the package in the `example` directory. Only the `getProxyURL` method needs to be implemented - the contract for this method is “given the incoming `HttpServletRequest`, return the absolute URL of the XML document to be converted or `null` if an error occurred”.

Here’s the method from the supplied `SampleServlet`, which extracts the XML documents URL from the “PathInfo” of the request - this is anything in the URL path to the right of “/servlet/SampleServlet”.

```
  public String getProxyURL(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
  {
      URL url=null;
      String query = req.getPathInfo();
      try {
   if (query==null) throw new MalformedURLException();
          URL thisurl = new URL(HttpUtils.getRequestURL(req).toString());
          url = new URL(thisurl, res.encodeURL(query));
      } catch (MalformedURLException e) {
          res.sendError(404, "Invalid URL \""+query+"\"");
      }
      return url.toString();
  }
```

With this example, if the servlet was placed in the `WEB-INF/classes` directory as `SampleServlet.class`, then to load and convert an example called `/HelloWorld.xml` just enter the URL `http://yourdomain.com/servlet/SampleServlet/HelloWorld.xml`.

Obviously this is a simple example, and it’s fully expected that smarter proxies will be written with error checking and the like. The main things to remember when implementing this method are:

- The returned URL must be **absolute**. Here we ensure this by making the requested URL relative to `thisurl`, which is the URL of the current request.
- If something goes wrong, this method should return `null` and an error should written to the `HttpServletResponse`.

For those requiring more complete control over the conversion process, source code for the `PDFProxyServlet` is supplied in the `docs` directory.

Creating PDFs using a transformer

When the XML to be converted is a result of one or more transformations, the PDF can be created as the end result of the chain. The transformations can either be a handwritten `XMLFilter`, like the `SampleTransformer.java` example supplied with the package, or the result of an XSL transformation. This saves having to serialize and deserialize the XML, although it does require at least a SAX 2.0 parser. Here's an example, which is also supplied with the download package as `SampleTransformer.java`:

```
import java.io.*;
import org.faceless.report.ReportParser;
import org.faceless.pdf.PDF;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public void createPDF(String xmlfile, OutputStream out)
    throws TransformerException, IOException
{
    // Create your filter, either explicitly or using
    // the SAXTransformerFactory.newXMLFilter() method
    //
    XMLReader reader = XMLReaderFactory.createXMLReader();
    XMLFilter filter = new MyFilter(reader);

    InputSource source = new InputSource(xmlfile);
    ReportParser parser = ReportParser.getInstance();
    PDF pdf = parser.parse(filter, source);
    pdf.render(out);
    out.close();
}
```

Requesting PDF documents via HTTPS

Whether using the Proxy Servlet or the Filter, in principle requesting a PDF document over an SSL encrypted session is identical to requests using normal HTTP. In practice however, many web servers are only set up to handle *incoming* HTTPS requests, not outgoing. This is easy to test - add the line

```
java.net.URL url = new java.net.URL("https://localhost");
```

to any servlet or JSP, and run it. If you get a `MalformedURLException` complaining of `unknown protocol: https`, then your web server isn't set up to allow outgoing HTTPS requests - more specifically, this is caused by the HTTPS protocol handler either not being installed or not being registered with the web-application security handler.

Prior to version 1.1 this was an irritating problem. Any relative links in the document are relative to the base URL of the document, and if it was requested via an HTTPS URL, these links will themselves be HTTPS (in practice, even documents with no relative links were causing problems, as the SAX parsing routines require a base URL regardless). In version 1.1 we added a couple of ways to workaround this issue. The first is all done behind the scenes. If a PDF is requested via HTTPS, but the webserver can't handle outgoing HTTPS requests, the base URL of the document is internally downgraded to HTTP. This isn't a security risk, because any requests to relative URLs for images, stylesheets and so on are all made from the *server to the server* - ie. the requests are made to `localhost`. The completed PDF is still sent back to the browser over a secure link.

If you don't like this, or for some reason it won't work (for example, because your webserver *only* handles HTTPS and not HTTP), there are a couple of other options. First, you can install the JSSE package and register the HTTPS protocol handler (this was the only option for earlier versions of the Report Generator). This can be done either by upgrading to Java 1.4, which includes JSSE1.0.3, or by

installing it separately. The broad details on how to do this are on the JSSE website at http://java.sun.com/products/jsse/install.html - you can probably find specific instructions for your webserver through your normal support channels.

Please remember this problem is *not specific to the report generator*, but applies to any web application that needs to create an HTTPS URL. Although every webserver will have a different way of doing this, we did find some Tomcat 4.0 specific instructions at http://www.planetsaturn.pwp.blueyonder.co.uk/tomcatandhttps) which you may be able to adapt if you can't find anything for your server.

The second option is much simpler. You can use the new `base` meta tag to set the base URL of the document to any value you like. For example, to get all relative links in the document to load from the filesystem, rather than via the webserver, add something like this to your code, immediately after the `<head>` tag:

```
<pdf>
  <head>
    <meta name="base" value="file:/path/to/webapplication"/>
  </head>
```

This will cause relative links in your document like to be resolved as `file:/path/to/webapplication/images/logo.gif`.

Creating the XML

A simple example

```
1.  <?xml version="1.0"?>
2.  <!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
3.
4.  <pdf>
5.    <head>
6.      <meta name="title" value="My First Document"/>
7.    </head>
8.    <body background-color="yellow" font-size="18">
9.      Hello, World!
10.   </body>
11. </pdf>
```

This simple document creates a single page PDF with the text “Hello, World!” in 18pt text at the top of the first page. Barring the first two lines, it should look fairly familiar to anyone that’s ever created an HTML page.

Although it’s simple, there are a couple of key points here. Let’s go through this example a line at a time.

Line 1. the XML declaration `<?xml version="1.0"?>` must always be included as the *very first line* of the file.

Line 2. the DOCTYPE declaration tells the XML parser which DTD to use to validate the XML against. See here for more information on DTDs.

Line 4. the top level element of the XML document must always be `pdf`.

Line 5. like HTML, the document consists of a “head”, containing information *about* the document, and a “body” containing the *contents* of the document.

Line 6. a trap for HTML authors. In XML an element must always be “closed” - this means that `<pdf>` must always be matched by `</pdf>`, `<b>` by `</b>` and so on. When an element has no content, like `<br>`, <img> or `<meta>`, it may close itself by writing it as we’ve done here - `<meta/>`

Line 8. The `<body>` element has some attributes set - `background-color` and `font-size`. In XML, every attribute value must be quoted - this can be frustrating for HTML authors used to typing `<table width=100%>`.

Creating Dynamic Reports

A report generator isn’t much use if it can’t create reports based on dynamic data - creating customer account statements on-the-fly from database queries, for example.

Rather than use custom elements to query the database and include the results, we’ve gone with a much more flexible solution and separated the *generation* from the *PDF conversion*. This means you can use your favorite technology to create the dynamic XML - we prefer JSP, but ASP, XSLT, CGI or any other solution will do - and the Filter or Proxy Servlet will convert that to PDF transparently.

Here's an example showing how to create a PDF with the current date from a JSP. There are some more examples in the `examples/dynamic` directory of the download package.

```
1.  <?xml version="1.0"?>
2.  <%@ page language="java" contentType="text/xml; charset=UTF-8"%>
3.  <!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
4.
5.  <pdf>
6.    <body font-size="18">
7.      Today is <%=new java.util.Date()%>
8.    </body>
9.  </pdf>
```

This is very similar to the previous example. We've marked the two changes in bold.

The first one is the most important. You need to set the page Content-Type to `text/xml`, in order for it to be converted to a PDF. You should also set the `charset` to UTF-8, like we've done here. This is because of an important difference between HTML and XML - the default characterset for HTML (and therefore for JSPs) is ISO-8859-1, but the default for XML is UTF-8. Of course, if you're only using 7-bit ASCII characters characters you can leave this out, but it's a good idea to do it anyway.

You may have noticed that the JSP `page` directive is the second line, rather than the first (as is normally the case with JSP's) - this is because the `<?xml` directive must be on the first line of the XML - most SAX parsers will throw an error if it's not.

The second change is on line 7, where we print the current date using a JSP directive. By now we hope it's fairly clear that creating a dynamic report is basically the same as creating a dynamic HTML document - provided the XML syntax is adhered to.

The DOCTYPE declaration

A quick word about the DOCTYPE declaration (the third line in the example above). The DOCTYPE, or DTD, is used by the XML parser to store information about the structure of the document - which elements can contain which, and so on. The XML document refers to the DTD using two strings - the "public" identifier and the "system" identifier, which are the values "`-//big.faceless.org//report`" and "`report-1.1.dtd`" in the example above.

In practise, XML documents include a DTD for two main reasons:

- To automatically validate the XML document against the DTD
- To convert named entities like ` ` into character values

XML validation isn't used in this package (we do our own validation instead), so the main reason this is required is to convert named entities (see Appendix B for a list of named entities understood by the Report Generator DTD). If you don't use any, you can leave the DOCTYPE line out with no ill effect.

The actual DTD is stored in the JAR file. The Report Generator recognises the public identifier "`-//big.faceless.org//report`" and loads the DTD directly from the JAR, so most of the time you won't need to worry about it. As always, there are a couple of exceptions to this:

- Several XML parsers (including Allaire JRun 3.1 and Caucho Resin up to 2.1.3) are unable to load a DTD from a JAR, and requires the DTD to be loaded from a URL
- When creating a PDF from a `javax.xml.transform.Source` using the `transform` method, the DTD cannot be read from the jar, and must be loaded from a URL.
- If you're trying to examine or edit the XML using a "smart" XML tool, like Internet Explorer 5 (we use the term "smart" loosely), the DTD needs to be accessible.

In all these cases, the DTD will be loaded from the URL specified by the "system" identifier. As the DTD file is supplied in the `docs` directory of the download package, it can be copied into an appropriate directory for your webserver to serve. An alternative is to reference the DTD directly from the Big Faceless Organization web server by changing the DOCTYPE declaration to this:

```
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report"
  "http://bfo.com/products/report/report-1.1.dtd">
```

(this is not recommended for regular use, as loading it from a remote server will slow down the parsing process)

Namespaces: Embedding XML Metadata

One of the new features adding in SAX version 2 was the concept of XML "namespaces". Namespaces don't play a major role in the Report Generator, as the end result is a PDF rather than another XML document. The role they do have relates to XML Metadata, which, with the arrival of Acrobat 5.0, can be embedded directly into a PDF document for later extraction. Adobe call this XMP, and more information on this is available at http://www.adobe.com/products/xmp.

The Report Generator automatically recognises whether a SAX 2.0 parser is being used, and will become "namespace aware" if it is. In this case, any elements with a namespace other than `http://big.faceless.org/products/report` will be considered as XMP metadata, and will be embedded as-is into the PDF document. (Note that this is the default namespace for any element without a namespace explicitly specified). Because of the way this works, XMP metadata *cannot* be embedded with a SAX 1.0 parser - an error will be thrown instead. As it's very difficult to work with XMP *without* using namespaces, this shouldn't be a concern.

Not every structure in a PDF document can contain XML metadata - currently, the only tags that will accept it are <pdf> (to specify metadata about the entire document), <img> (to specify metadata about an image), <body> (to specify metadata about the first page) and <pbr> (to specify metadata about the following page). Metadata that is specified on any other tag will be silently dropped.

Here's a brief example showing how this could be put to use - an image is embedded in a document along with information on from whence it came. Content in **bold** is *not* embedded as metadata but is parsed and processed by the Report Generator

```
<img src="resources/canoe.jpg">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:dc="http://purl.org/dc/elements/1.1"
             xmlns:tc="http://www.w3.org/2000/PhotoRDF/technical-1-0#">
    <rdf:Description about="">
     <dc:type>image</dc:type>
     <dc:title>Fishing Boat</dc:title>
     <dc:description>Photo of a boat on the coast in Ghana</dc:description>
     <dc:creator>mike@bfo.com</dc:creator>
     <dc:date>1999-04-20</dc:date>
     <tc:camera>Canon EOS 100</tc:camera>
     <tc:lens>Sigma 28mm</tc:lens>
    </rdf:Description>
  </rdf:RDF>
</img>
```

For a full example have a look at the `MetaData.xml` example in the download package.

Styles

The look and feel of a document is defined using Cascading Stylesheets (level 2), or *CSS2* - the same system used by HTML. The full CSS2 specification is online at http://www.w3.org/TR/REC-CSS2, and unlike many specifications it's clear enough to be understood by mere mortals - we recommend reading it. We support most, but not all of the specification - see the appendices for details.

The first way to set the style for an element is inline. Unlike HTML, there is no difference between a "style" attribute and a regular attribute - whereas in HTML to specify an inline style you must write `<table style="background-color:red">`, in XML you could simply write `<table background-color="red">`. All the examples up until now have used inline styles.

Although in many cases this method is appropriate, if the same style is used more than once in a document it's generally easier to use a "stylesheet" - a collection of CSS2 rules defined in the HEAD of the document which set attributes for various elements in the BODY.

Stylesheet definitions

Stylesheets can be included directly in the document or linked in from an external file. In both cases the syntax is the same. A *Stylesheet* consists of one or more (selector, attribute) pairs - each *selector* matching certain elements in the document, and the *attributes* defining which attributes to set for those elements. Here's an example:

```
body      { size:Letter; padding:0.5in; }
H1, H2    { font-family:Times; }
.example  { background-color:yellow; }
```

This example sets the "size" attribute for the BODY element to "Letter" and it's "padding" attribute to "0.5in", sets the "font-family" attribute for all H1 and H2 elements to "Times" and sets the background color for any elements with the "class" attribute set to "example", to yellow.

The CSS2 specification gives a great deal of control over the selector. Here's a list of the different options.

Pattern	*Meaning*
`*`	Matches any element
`E`	Matches any E element (i.e., an element of type E)
`E F`	Matches any F element that is a descendant of an E element
`E > F`	Matches any F element that is a child of an element E
`E:first-child`	Matches element E when E is the first child of its parent
`E:last-child`	Matches element E when E is the last child of its parent (custom extension of CSS2)
`E + F`	Matches any F element immediately preceded by an element E
`E - F`	Matches any F element immediately followed by an element E (custom extension of CSS2)
`E.warning`	Matches any E element with the "class" attribute equal to "warning"
`E#myid`	Matches any E element with the "id" attribute equal to "myid"
`E:lang(fr)`	Matches any E element where the "lang" attribute begins with "fr" - including, for example, "fr_CH"
`E[align=right]`	Matches any E element where the "align" attribute is set to the value "right"
`E[align]`	Matches any E element where the "align" attribute is set - the actual value it is set to is irrelevant
`.warning`	Matches any element with the "class" attribute equal to "warning"

Pattern	*Meaning*
`#myid`	Matches any element with the “id” attribute equal to “myid”

Matching certain types of element

To match elements of a specific type in the document is the simplest type of rule. The following example matches every H1 element in the document, and sets the color to red.

```
H1 { color:red; }
```

Classes and ID’s

An HTML-specific extension to CSS2 which we have adopted is the concept of matching “classes” and “ids”. This allows elements in the document to be grouped together, or even to match individual elements. For example, every example in this document is printed in a box on a light blue background. Here’s how we do it:

```
PRE.example { background-color:#D0FFFF; padding:4; border:1; }
```

Then in the document we simply place each example inside a `<PRE class="example">` element.

As of version 1.1.10, each element can belong to multiple classes. For instance, this paragraph would have a red background **and** a black border.

```
.red { background-color:red; }
.outline { border:thin solid black; }

<p class="red outline">
```

Individual elements can be referenced by ID as well. For example, to reference a specific diagram in the document you might set it’s “id” attribute to “diagram1”, and then use the following stylesheet rule:

```
#diagram1 { border:1; }
```

Each page in the document is given a unique ID equal to “page” followed by the current pagenumber. For example, here’s how to set the size and background color of the first page.

```
#page1 { size:A4-landscape; background-color:yellow; }
```

One additional advantage of giving an element an ID is that it can be referenced from outside the document. This can be used to load a PDF at a specific page or section of a page, but only works with documents loaded with the Internet Explorer or Netscape plugin from a webserver. For example, to open the document to the block with an ID of “chapter2”, put the following hyperlink in your HTML document:

```
<a href="http://www.yourcompany.com/YourPDF.pdf#chapter2>See Chapter 2</a>
```

Descendants, Children and Siblings

At times, authors may want to match an element that is a descendant or child of another element in the document tree - for example "match any H1 elements on the first page" (a *descendant* relation), or "match any P elements that are children of BODY" (a *child* relation). These two rules can be described by the following stylesheet entries:

```
#page1 H1 { color:red; }
```

```
BODY > P { color:red; }
```

In the first example, the descendant relation is specified by the whitespace between the `#page1` selector and the `H1` selector. These can be chained together as necessary - for example `DIV * P` matches any P element that is the grandchild or later descendant of a DIV.

In the second example, the child relation is specified by the `>` symbol. Only P elements directly under the BODY element will be matched.

Sometimes it may also be necessary to match elements based on their siblings, rather than their ancestors - for example, to set the vertical space for an H2 element when it's immediately preceded by an H1 element. Another useful option is to match an element that *isn't* preceded by another element - it's the first child of it's parent. This is useful to set a default style for the first column of a table, for example. The following two examples show how to describe these situations.

```
H1 + H2 { margin-top:0pt; }
```

```
td:first-child { font-weight:bold; }
```

Two custom extensions which we support but CSS2 doesn't are the *last-child* psuedo-element and the "previous sibling" relation. These are the opposite of the two rules shown above, and can be matched like this:

```
H2 - H1 { margin-bottom:0pt; }
```

```
td:last-child { font-weight:bold; }
```

Grouping

When several identical attributes are to be set for different elements, they may be grouped into a comma separated list. The following two examples are identical:

```
H1 { font-family:Times; }
H2 { font-family:Times; }
H3 { font-family:Times; }
```

```
H1, H2, H3 { font-family:Times; }
```

Language and Attribute selectors

New in version 1.1 is the ability to select attributes based on the language of an element, as defined by the `lang` attribute, or based on other attributes. The language selector is extremely useful when creating a document that will contain text in more than one language. For example, the following rules set the default font for different languages and the default page size for Americans and Canadians. They are included in the default stylesheet.

```
body:lang(ko)          { font-family: HYMyeongJo; }
body:lang(ja)          { font-family: HeiSeiKakuGo; }
body:lang(zh_CN,zh_SG) { font-family: MSung; }
body:lang(zh_TW,zh_HK) { font-family: STSong; }

body:lang(en_US,en_CA) { size: Letter; }
```

The language of an element can be set using the `lang` attribute in the same way as HTML, by using the XML-specific attribute `xml:lang`, or if neither are set it defaults to the Locale that the Report Generator is running in.

As for the attribute selectors, they're easier to understand with an example. In HTML, an image that is also a hyperlink traditionally has a blue-border around it. This can be done with the following stylesheet entry:

```
img[href] { border: medium solid blue; }
```

Similarly one could create appropriate margins on a floating block by using something like the following, which puts left margins on a right-floated DIV, and right-margins on a left-floated DIV.

```
div[float=right] { margin-left: 10pt }
div[float=left] { margin-right: 10pt }
```

Applying Stylesheets

So how to include this style information in the document? The following three examples show different ways to get the same result.

First, you can include the attributes inline. Quick, but inflexible.

```
<?xml version="1.0"?>
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">

<pdf>
  <body background-color="yellow" font-size="18">
    Hello, World!
  </body>
</pdf>
```

```
<?xml version="1.0"?>
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">

<pdf>
  <head>
    <style>
      body    { background-color:yellow; font-size:18 }
    </style>
  </head>
  <body>
    Hello, World!
  </body>
</pdf>
```

Second, you can embed the stylesheet directly in the document.

Third, for maximum flexiblity, create the stylesheet as a separate file. The first file here is called `style.css`, and we load it using the LINK element.

Relative URLs referenced from the stylesheet will be relative to the sheet, *not* the document that uses it.

```
body    { background-color:yellow; font-size:18 }
```

```
<?xml version="1.0"?>
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">

<pdf>
  <head>
    <link type="stylesheet" src="style.css"/>
  </head>
  <body>
    Hello, World!
  </body>
</pdf>
```

Elements

Most of the XML elements we use are the same as HTML. In this section we'll broadly describe what the various elements are - most of them should be familiar, but there are a few new ones and a couple of important differences to others. See the reference section for a full list and more detail.

Document Structure

Every report is defined by a single PDF element, which may contain an optional HEAD element, and must contain the mandatory BODY element, which contains the actual content of the report. As PDF documents consist of multiple pages, the contents of the BODY may be split into one or more pages - a process known as pagination.

Pagination

Generally speaking the Report Generator uses the following algorithm to place elements on the page.

1. Take the first element in the BODY and try to fit it on the current page.
2. If it can't fit but it can be split in two, split it at the end of the page and carry on.
3. If it can't be split into two halves, place it on the next page and carry on

This process can be altered in one of three ways.

- Using the "page-break-before", "page-break-after" and "page-break-inside" attributes to control breaks between elements.
- Within paragraphs (the P, PRE, BLOCKQUOTE and H1 to H4 elements), set the "orphans" and "widows" attributes to control the minimum number of lines that must remain at the end of a page (the orphans) and the minimum number that may be at the top of a new page (the widows). These both default to 2.
- Using the PBR element to explicitly place page breaks

The first method, which is part of CSS2, allows authors to set various attributes on elements to either prevent or force pagebreaks. For example, the default setting for the H1 to H4 elements is that they are never immediately followed by a page break. The stylesheet entry looks like this:

```
H1, H2, H3, H4 { page-break-after:avoid; }
```

Only some elements may be split and placed on multiple pages if they don't fit - currently the TABLE, UL, OL and all the paragraph elements listed above. To prevent one of these elements being split, set the "page-break-inside" attribute to "avoid".

The third method uses the PBR element to split pages. This is especially useful when you want to change the format of the document, as the page dimensions for the new page (and for all following pages) can be set explicitly.

For example, lets say you want your report to have a cover page on A4 with a yellow background, the bulk of the report on normal A4 but also a special section at the end to be printed on landscape. Here's how to do it:

```
<pdf>
  <body size="A4" background-color="yellow">

     Contents of front page here

    <pbr background-color="white"/>

     Bulk of report here

    <pbr size="A4-landscape"/>

     Landscape section here

  </body>
</pdf>
```

As well as setting page formats and colors, this method can also be used to set page margins and "macros" for setting headers and footers.

Headers, Footers and Macros

To display headers and footers on the page, the Report Generator introduces the concept of a "macro" - a block of XML which can be repeated multiple times throughout the document.

There are three different types of macro attribute, which can be used either on the BODY or PBR elemnts to set a macro for every page, or for a specific page by using a #page*n* entry in a stylesheet.

- `header` - to set the header of the page
- `footer` - to set the footer of the page
- `background-macro` - to set the background of the page

A macro is defined in the HEAD of the document inside a MACROLIST. Each macro must have an ID, which is how it's referenced later in the document. Here's an example which sets a standard footer on each page:

```
<pdf>
  <head>
    <macrolist>
      <macro id="myfooter">
         <p align="center">
           Page <pagenumber/> of <totalpages/>
         </p>
      </macro>
    </macrolist>
  </head>
  <body footer="myfooter" footer-height="20mm">

    Document contents here

  </body>
</pdf>
```

The “footer” attribute is the ID of the macro, and the “footer-height” attribute is the height required for the footer. If the document contents require several pages, the footer will be placed on each one, unless there is a PBR element which changes the footer (or removes it by setting `footer="none"`). The “header” attribute can be used the same way to set a header at the top of each page.

The “background-macro” element allows more control than the “background-image” and “background-color” attributes. A classic example is placing a watermark on each page. Rather than use a bitmap image and set “background-image”, the background-macro allows you to add custom XML to each page. The watermark can cover the whole page - including the header and footer if they’re specified, but excluding any page margin or padding. Here’s an example which places the word “Confidential” on each page in light gray:

```
<pdf>
  <head>
    <style>
      #watermarkbody { font-size:80; font:Helvetica; color:#F0F0F0; }
    </style>
    <macrolist>
      <macro id="watermark">
         <p id="watermarkbody" rotate="-30" valign="middle" align="center">
           Confidential
         </p>
      </macro>
    </macrolist>
  </head>
  <body background-macro="watermark">

    Document contents here

  </body>
</pdf>
```

Displaying the Page number

The current page number and the total number of pages in the document can be displayed in the document by means of two special elements - PAGENUMBER and TOTALPAGES. These can be used inside a text paragraph - the “footers” example above shows how they are used.

The current page number generally starts at one and increases by one for each page, but can be set specifically by using the “pagenumber” attribute. This can be set on a BODY or PBR element to set the page number of the next page.

As well as just printing the current page number, the PAGENUMBER element can be used to print the page number of other elements in the document. This comes into it’s own when creating a table of contents. Every item in the table of contents has an `id` tag - for example, the header at the start of this paragraph has it’s `id` attribute set to “pagenumbers”. Then, in the table of contents, we can print the page number of this section like so:

```
<table>
 <tr>
  <td>Displaying the Page Number</td>
  <td><pagenumber idref="pagenumbers"/></td>
 </tr>
</table>
```

A mildly annoying feature of these two tags is that they cannot be measured accurately during the layout stage of the document. This is obvious when you think about it - there's no way to know how many pages are required until the whole document has been laid out. Because of this the Report Generator takes a guess at the number of digits that might be required. This defaults to three, but since 1.1.12 can be set with the "size" attribute. For instance, if you know your document will have a maximum of 50 pages, you might change your code to read

```
Page <pagenumber size="2"/> of <totalpages size="2"/>
```

Another option added in the same release was the ability to display page numbers in formats other than decimal. The types available are the same as for the "markertype" attribute in the OL tag - so for example, to number your pages with roman digits, try:

```
 <macrolist>
  <macro id="myfooter">
   <p align="center"><pagenumber type="roman-lower"/></p>
  </macro>
 </macrolist>
</head>
<body footer="myfooter" footer-height="0.5in">
```

Page Sizes

As a convenience, the Report Generator defines several standard sizes which can be used to set pages in the document to a standard paper size - so `<body size="A4">` is identical to `<body width="210mm" height="297mm">`. Here's the list of known sizes - every one of these can have the suffix "-landscape" appended to rotate the page size by 90 degrees - e.g. `letter-landscape`.

ISO A series		*ISO B series*		*ISO C series*	
A10	26mm × 37mm	B10	31mm × 44mm	C10	28mm × 40mm
A9	37mm × 52mm	B9	44mm × 62mm	C9	40mm × 57mm
A8	52mm × 74mm	B8	62mm × 88mm	C8	57mm × 81mm
A7	74mm × 105mm	B7	88mm × 125mm	C7	81mm × 114mm
A6	105mm × 148mm	B6	125mm × 176mm	C6	114mm × 162mm
A5	148mm × 210mm	B5	176mm × 250mm	C5	162mm × 229mm
A4	**210mm × 297mm**	B4	250mm × 353mm	C4	229mm × 324mm
A3	297mm × 420mm	B3	353mm × 500mm	C3	324mm × 458mm
A2	420mm × 594mm	B2	500mm × 707mm	C2	458mm × 648mm
A1	594mm × 841mm	B1	707mm × 1000mm	C1	648mm × 917mm
A0	841mm × 1189mm	B0	1000mm × 1414mm	C0	917mm × 1297mm
2A0	1189mm × 1682mm	*American sizes*		*Other sizes*	
4A0	1682mm × 2378mm	**Letter**	**8.5in × 11in**	ID-2	107mm × 74mm
Common envelopes		Legal	8.5in × 14in	ID-3	125mm × 88mm
D1	110mm × 220mm	Executive	7.5in × 10in	OHP-A	250mm × 250mm
E4	280mm × 400mm	Ledger	11in × 17in	OHP-B	285mm × 285mm

The Document Head

The HEAD element of the report contains information about the report. There are five different options that can be specified inside the HEAD.

- Macros (described above) using the MACROLIST and MACRO elements
- Stylesheets, either externally using a LINK or internally using a STYLE element
- Non-standard fonts can be linked in using the LINK element. This is covered in the “Fonts” section later.
- Document meta information, such as report title, password and various PDF specific attributes can be set using the META element.
- Bookmarks can be specified using the BOOKMARKLIST and BOOKMARK elements

Meta information

The META element in the document HEAD requires a “name” and “value” attribute, which specifies which property of the document to set. A number of properties are known to the Report Generator, and those that aren’t can be passed on to the calling process - providing a convenient method of extending the capabilities of the generator. Here’s an example setting the title of the document.

```
<pdf>
  <head>
    <meta name="title" value="My First Report"/>
  </head>
</pdf>
```

Here’s a list of the various “names” that are recognised, ordered roughly from most useful to least useful (as we think anyway)

Name	*Value*	*Description*
`base`	Base URL of the document	Set the base URL of the document. All relative links in the document will be interpreted as relative to this URL. If you’re going to set this, be sure to set it before any stylesheets or fonts are loaded.
`title`	The report title	Set the title of the report
`author`	The authors name	Set the author of the report
`subject`	The report subject	Set the subject of the report
`keywords`	a list of keywords	Set the keywords for the report
`output-profile`	the name of an output profile	This can be set to cause the PDF to be written according to the rules of a specific output profile. For more detail see the `org.faceless.pdf2.OutputProfile` class. Valid values are currently “Default”, “NoCompression”, “Acrobat4”, “Acrobat5”, “PDF/X-1a”, “PDF/X-3 (No ICC)” and “PDF/X-3 (ICC)”
`password`	a password	The password to encrypt the report with
`servlet-filename`	a filename	(For Proxy Servlet and Filter use only) Set the PDF to be saved rather than viewed directly by the browser, and set the name to give the PDF document when it’s saved. This functionality may cause problems with some browsers - see the Filter API documentation for more information
`servlet-cache`	period of time	(For Proxy Servlet only) Set the length of time the generated PDF is to be cached by the Proxy Servlet. See the Proxy Servlet API documentation for more information.

Name	*Value*	*Description*
`access-level`	print-none print-lowres print-highres extract-none extract-accessibility extract-all change-none change-layout change-forms change-annotations change-all plain-metadata	What permissions to give the application viewing the document. One of each of the “print”, “extract” and “change” values should be specified in a string, seperated with spaces. So, for example, `<meta name="access-level" value="print-all change-none extract-none"/>` would create a document that can be printed but is not copyable or alterable. For 40-bit encryption, print-lowres is the same as print-highres, extract can be “none” or “all”, and changes can be “none”, “annotations” or “all”. The “plain-metadata” option will cause XMP metadata in the document to be left unencrypted, although this will result in a PDF that can only be loaded with Acrobat 6.0 or later.
`show-bookmarks`	true / false	Whether to show the bookmarks pane when the document is first opened
`layout`	one-column / two-column-left / two-column-right / single-page	Instruct the PDF viewing application on how to display the document. The default is `single-page`
`encryption-algorithm`	40bit / 128bit / aes	The encryption algorithm to use to secure the document. If a password or access-level is set, defaults to `40bit`. “aes” will result in documents that can only be opened in Acrobat 7.0 or later, but other than that is identical to 128bit.
`creator`	a program name	Set the name of the program that created the original XML
`viewer-fullscreen`	true / false	Whether to open the PDF viewer in fullscreen mode
`viewer-hidetoolbar`	true / false	Whether to hide the toolbar of the PDF viewer when the document is first opened
`viewer-hidemenubar`	true / false	Whether to hide the menubar of the PDF viewer when the document is first opened
`viewer-hide-windowui`	true / false	Whether to hide the user-interface of the PDF viewer when the document is first opened
`viewer-fitwindow`	true / false	Whether to resize the PDF viewer to fit the document size
`viewer-centerwindow`	true / false	Whether to center the PDF viewer window on the screen
`security-password`	a password	The password (if any) required to change the password of the document

Bookmarks

The documents “bookmarks” are the tree-like structure displayed in a pane on the left in Acrobat Reader. Sometimes called “outlines”, these are an excellent way to provide easy navigation around larger documents.

The Report Generator controls bookmarks through the BOOKMARKLIST element, which contains one or more BOOKMARK elements. These can themselves contain BOOKMARK elements, to create the tree structure. Each bookmark has a “name”, which is the name displayed to the user in the PDF, and an optional “href”, which is the hyperlink to follow if the user clicks on the bookmark - usually, but not necessarily, to a location in the document.

We'll cover more on Hyperlinks in a later section. For the moment, it's enough to know that linking to a specific location in the report is done by setting `href="#id"`, where "id" is the ID of the element you want to link to. Here's an example:

```
<pdf>
  <head>
    <bookmarklist>
    <bookmark name="Chapter 1" href="#ch1"/>
    <bookmark name="Chapter 2" href="#ch2">
      <bookmark name="Chapter 2 part 2" href="#ch2pt2">
    </bookmark/>
    <bookmark name="Chapter 3" href="#ch3"/>
    </bookmarklist>
  </head>
  <body>
    <h1 id="ch1">

      Chapter one here

    <h1 id="ch2">

      Chapter two part one here

      <h2 id="ch2pt2">

        Chapter two part two here

    <h1 id="ch3">

      Chapter three here

  </body>
</pdf>
```

The "expanded" attribute can be set to "true" to cause the specified bookmark tree to be opened by default. The "color", "font-style" and "font-weight" attributes may also be set to set the look of the bookmark entry, although this feature is ignored by PDF viewers before PDF 1.4 (Acrobat 5.x)

Box Model

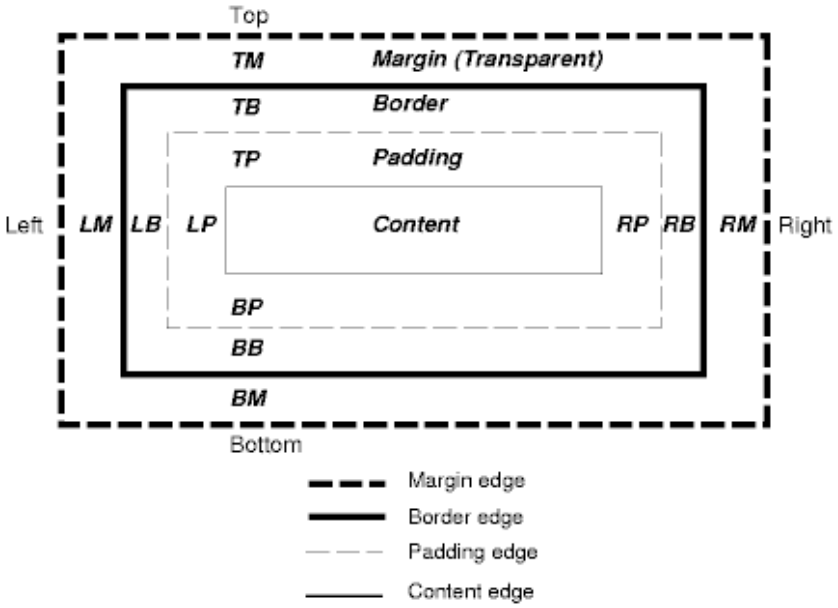
The "box model" is the name given to the layout model used by both CSS2 and the Report Generator. Coming to grips with how it works will help you to control the layout of your reports.

Every element that is displayed in the body of the report is a box - be it a paragraph of text, a table, a bitmap image or even a page itself. These boxes are usually positioned one after another down the page to make up the report.

All these elements have certain properties in common, which can be set by the various *block attributes* in the report generator. We'll cover some of these attributes now.

Padding, Margins and Borders

Every "box" placed in the document takes up a certain amount of space. As well as the obvious space required to display the content of the box, such as the dimensions of an image, there is the space around the content as well, which separates it from it's neighbors.



The diagram above shows the various “shells” around the content of a box. Starting with the content and moving out, we have:

1. Padding - the space between the content of the block and the border, this has the same background color or image as the content of the block.
2. Border - the optional border line surrounding the content of the block.
3. Margin - the space outside the border between this block and it’s neighbors. It’s always transparent.

The “padding”, “border” and “margin” attributes can be set to set the attribute for all four sides of the box, or “padding-top”, “padding-right”, “padding-bottom” or “padding-left” etc. can be set to set the border, padding or margin for just one side.

The Report Generator also supports the “border-color” attribute to set the color of the border, the “border-style” attribute to set the border line to solid, dotted, dashed and so on, and the custom “corner-radius” attribute, which allows the corners of the border to be rounded. Border colors and styles can be set seperately for each side - for example

```
div { border-top: dotted red; border-bottom: thick solid black; }
```

will draw a dotted red border above the DIV tag, and a thick solid black one below it.

Drawing the Background

Both the content and the padding of a box can optionally be drawn over a background. This can either be a color, by setting the “background-color” attribute, or a bitmap image as set by the “background-image” attribute.

The background image can be drawn in one of several positions, as set by the “background-position” attribute. By default this is set to “stretch”, which means the image is drawn once and stretched to fit the box. Other options are “repeat”, where the image is tiled repeatedly to fill the box, or any combination of “top”, “middle”, “bottom”, “left”, “center” or “right” to draw the image once. PDF is not as efficient as HTML at rendering background images, so the “repeat” setting should be used with care as it can result in long delays for those viewing the document.

Unlike HTML, PDF images don’t have a fixed size. Instead, the size of the bitmap image on the page depends on the dots-per-inch, or DPI of the image. For background images, this can be set using the “background-image-dpi” or “background-image-width” and “background-image-height” attributes. These have the same function for background images as the “image-dpi”, “width” and “height” attributes do for normal images - see the section on Images for more information.

Here are some examples showing the effects of the different settings

background-image-position="stretch"

background-image-position="repeat"

background-image-position="center middle"

Building on an existing PDF

A feature of the **Extended Edition** of the Report Generator is the ability to use a page from an existing PDF document as a background, in the same was as you could use a `background-image` or a `background-color`. This is done using the `background-pdf` attribute, which can be set to the URL of a PDF to include. Here's an example:

```
<?xml version="1.0"?>
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">

<pdf>
  <body background-pdf="original.pdf#page=2" font-size="18">
    Hello, World!
  </body>
</pdf>
```

This simple example would create a single page document, with the words "Hello, World!" placed on top of the second page of the "original.pdf" document. The pagenumber is specified by the "#2" in the URL - it can be left out, in which case the page that's used will be the same page as that in the current document - the first page is overlaid on page 1, the second is overlaid on page 2, and so on. When the source document is out of pages it starts again at the beginning.

A useful example of this is a multi page invoice. Imagine you want to create an invoice, which will run over several pages. The first page has the company logo and space for an address, whereas the remaining pages just have space for the invoice details. To do this with the report generator, create a two page template using your favorite tool - Quark Express or MS Word, for example - and then do something like the following example:

```
<pdf>
  <head>
    <style>
      #page1 { background-pdf:original.pdf#page=1 }
      body   { background-pdf:original.pdf#page=2 }
    </style>
  </head>
  <body>
    <p padding-left="1in" padding-top="1in">
      <!-- Address goes here -->
    </p>
    <table>
      <!-- Invoice details go here, covering as many pages as necessary -->
    </table>
  </body>
</pdf>
```

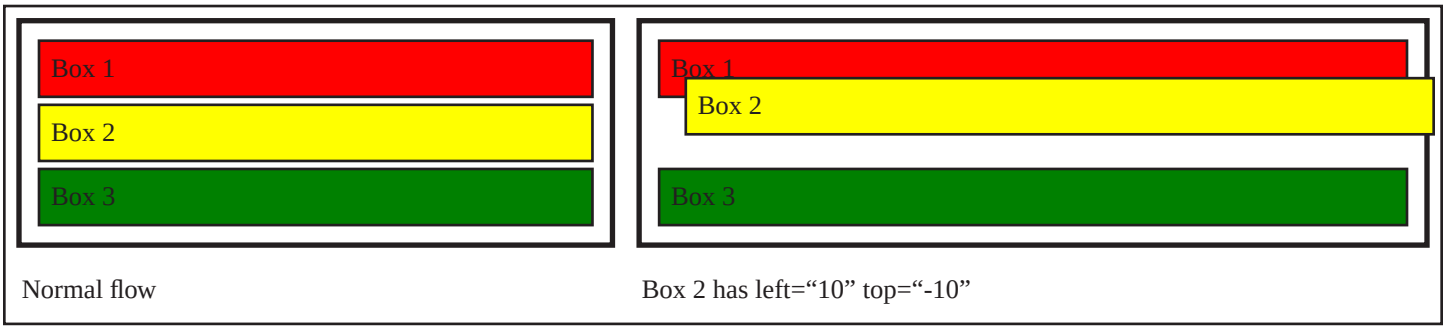
Note that this feature is not limited to pages! Theoretically an existing PDF could be used as the background for a table, a paragraph or any other box.

Extended edition pricing information is available from the product homepage.

Positioning

As mentioned above, most of the time the “boxes” containing the XML elements are placed on the page, each one following the next with no overlap between them - a procedure known as *relative positioning*. The distance between the blocks can be controlled to a degree using the “padding” and “margin” attributes discussed above - for most layout requirements, just these attributes are enough.

For more control, the “position”, “left” and “top” attributes can be set to change the way boxes are laid out. By default, the position is “relative”, which means the box is positioned normally and then offset by the “left” and “top” attributes - these default to zero. The position of the following box is calculated as if the box was not offset. Here’s an example:

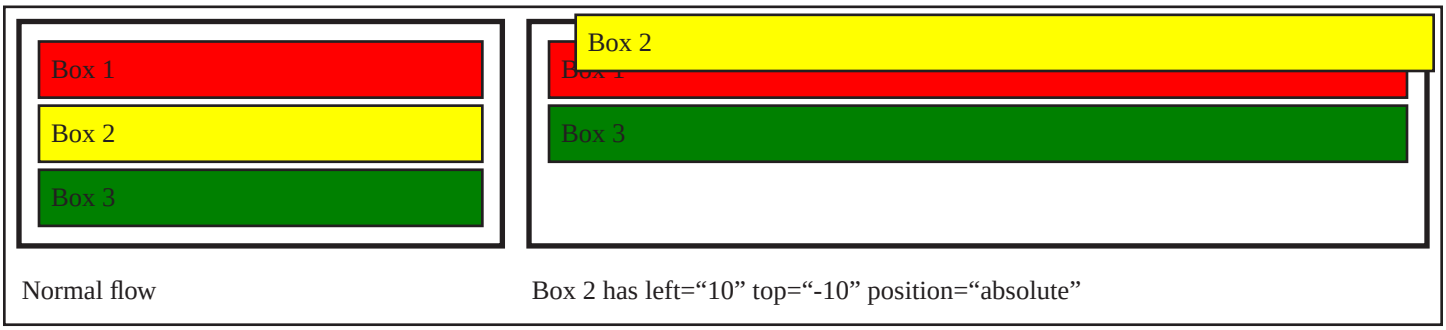


Normal flow

Box 2 has left=“10” top=“-10”

Sometimes this isn’t flexible enough - for example, if you want to place a paragraph of text on top of an image, or at a specific position on the page. In this case you can set the “position” attribute to “absolute”. This causes the box to be “taken out” of the normal flow and positioned relative to it’s parent *only* - i.e. completely independent of it’s siblings.

Here’s the above example again, but with the second box positioned absolutely. Notice how the left and top offsets are now relative to it’s parent, and how the third box is positioned as if the second didn’t exist.



Normal flow

Box 2 has left=“10” top=“-10” position=“absolute”

There is one critical condition when using absolutely positioned elements; the element cannot be the child of the BODY element. This is because unlike HTML, elements must be assigned to a page before they can be positioned, but as absolutely positioned items are independent of their siblings, there’s no way to decide which page they go on. To position an item at an absolute position on a specific page, it can be placed in a “background-macro” which is then assigned to the page.

Clipping and Visibility

In the above examples you will probably have noticed that the boxes overlap.In the case of the absolutely positioned example, it spills outside the bounds of it’s parent. This can be controlled by setting the “overflow” attribute, which can be set to “visible” (the default) or “hidden”. This determines whether an elements children are “clipped” at it’s edges or not.

Here’s the second example above, but with the “overflow” attribute of the parent element set to “hidden”. The element is clipped at the edge of the parents “content” box - because the parent has “padding” set to 4 this is 4 points inside the border.

Normal flow

Box 2 has left=“10” top=“-10” position=“absolute” overflow=“hidden”

The “overflow” attribute can be used to interesting effect with the CIRCLE, ELLIPSE and SHAPE elements.

There are two other attributes which will be familiar to HTML JavaScript programmers, but which aren’t as useful in PDF owing to the static nature of a PDF page - although we do support them. The `visibility` and `display` attributes affect whether an element on the page is displayed or not. The value of “visibility” defaults to “visible”, but can be set to “hidden” to prevent display of an element and it’s children, leaving the space it would have taken on the page empty. Alternatively, to remove an element altogether, set the “display” attribute to “none”, which will prevent the element both from being displayed and from having space allocated for it on the page.

Text and Fonts

Text Elements

The text handling in the report generator revolves around the idea of a *paragraph* - a rectangular block of text. Every line of text in the document is inside a paragraph - either an explicit one caused by the P, PRE, BLOCKQUOTE or H1 to H4 elements, or an “anonymous” paragraph (more on these below).

Inside a paragraph of text, the current font style may be changed by using *inline* elements, like B, I, A and SPAN. Inline elements may only be used inside a paragraph, but other than that are treated as normal blocks and may have a border, padding, background color or image as usual. Here’s a simple example.

```
<body>
  <p>This is a paragraph, <b>this is in bold</b> and this is back to normal</p>
</body>
```

Here’s a table summarizing the various text elements and what they’re intended for. More complete information is available in the Element reference.

Element	*Type*	*Purpose*
`P`	paragraph	A general purpose text container
`PRE`	paragraph	A type of paragraph that preserves whitespace and newlines
`H1 - H4`	paragraph	Used for headings
`BLOCKQUOTE`	paragraph	Used for quotes - indented in from the margins to the left and right
`SPAN`	inline	A general purpose inline element
`B`	inline	Set the font weight to **bold**

Element	*Type*	*Purpose*
I	inline	Set the font style to *italic*
U	inline	Set the text decoration to underlined
O	inline	Set the font style to outlined
A	inline	Set the text decoration to underlined
SUP	inline	Set the text to superscript
SUB	inline	Set the text to subscript
BIG	inline	Set the text to use a font size 1¼ times normal size
SMALL	inline	Set the text to use a font size ¾ times normal size
STRIKE	inline	Set the text decoration to ~~strike-out~~
TT	inline	Set the text to use a "typewriter" font, e.g. Courier
ZAPF	inline	Set the text to use the Zapf-Dingbats font
SYMBOL	inline	Set the text to use the Symbol font
NOBR	inline	Set the text to turn off automatic linewrapping
CODE	inline	Set the text to use a "typewriter" font, turn of line wrapping etc.
EM	inline	Identical to I
STRONG	inline	Identical to B

Anonymous Paragraphs

Under certain circumstances, the report generator will create "anonymous" paragraphs - basically it inserts a P element for you into the document where required. It will do this automatically if it finds text or inline elements directly inside a BODY, LI or TD element. Taking the example above, this could have been written as follows:

```
<body>
  This is a paragraph, <b>this is in bold</b> and this is back to normal
</body>
```

The Report Generator will automatically add the surrounding <P> and </P>, so internally this is converted to

```
<body>
  <p>This is a paragraph, <b>this is in bold</b> and this is back to normal</p>
</body>
```

If the parser is having trouble parsing a document, a good first step is to replace all the anonymous paragraphs with actual paragraphs, so you can see more clearly where the problem lies.

Making block elements inline

Since version 1.1 it's also possible to display block elements like images, tables and so on inside a paragraph. This can be done by setting the `display` attribute to "inline", rather than the default value of "block" (this is a break with the CSS2 standard, where all elements default to inline - we hope to fix this in a future release). Here's an example.

```
<p>
 This paragraph has an
 <img display="inline" src="images/logo.png"/>
 image in the middle.
</p>
```

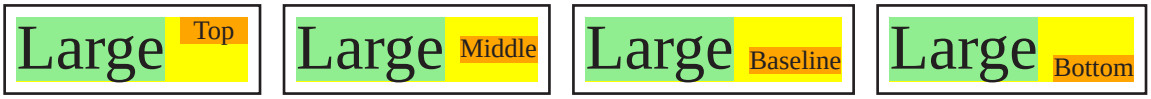
and here's the result

This paragraph has an  image in the middle

Vertical Alignment

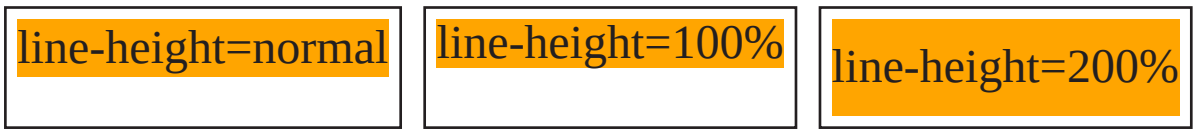
When mixing elements of differing heights in a paragraph, like the example above, there are several options available for vertical positioning. First, there are two definitions we need to make. The **Inline Box** is a box equivalent to the size of the inline item itself - usually a word or phrase, but as we saw above it's sometimes an image or similar. The above example contains three inline boxes, one for the text before the image, one for the image and one for the text after it. Each inline box is the same size or smaller than the **Line Box**, which is simply the box representing the physical line, and is always just big enough to fit it's inline boxes.

In the example below, the line box is in yellow, the larger text-box is in green and the smaller of the two text-boxes is shown in orange.



This example shows the four different options for vertical alignment within a line box, which is set with the `vertical-align` or `valign` attribute. "Top" places the top of the inline box at the top of the line box, "middle" places the middle of the inline box at the middle of the line box. "baseline", the default, places the baseline of the inline box at the baseline of the line box. Finally, "bottom" places the bottom of the text box at the bottom of the line box. There are two other values which can be used - "super" and "sub" - which place the text in the super or subscript position. These are not demonstrated here.

The height of each inline box depends on both the size of the font used, and it's *leading*, or white space between lines. This is set with the `line-height` attribute. Each font has a preferred leading set by the font author, which is equivalent to setting `line-height` to "normal" - usually equivalent to between 100% and 120% of the font size. The `line-height` can also be set to a percentage, in which case it's a percentage of the current `font-size`.



As you can see, any leading that is applied is split evenly above and below the text, as required by CSS2.

Float positioning

You've seen how to add blocks in the middle of a paragraph using the `display="inline` attribute, but there's one more common type of placement - known as *float positioning*. This causes the inline box to "float" to the left or right of the paragraph, and allows text to wrap around it.

"The following text will be drawn around the box to the right. When it grows beyond that box, it will automatically fill the full width of the line."

```
<p requote="true" text-align="justify" border="1" font-size="22pt" padding="4">
 <div float="right" width="120" height="50" background-color="blue"/>
 "The following text will be drawn around the box to the right. When it
 grows beyond that box, it will automatically fill the full width of the line."
</p>
```

Any inline elements can be floated to the left or right (setting the `float` attribute causes `display` to be automatically set to "inline"), and floating blocks can be started anywhere in a paragraph, not just at the start. Usually the floating block will start at the current line, but this depends on the value of the `clear` attribute. This attribute can be set to "none", "left", "right" or "both", to cause a floating block to be displayed only when the left margin is clear of any other floating blocks, the right margin is clear, or both are clear. The default is "none", which effectively says "it doesn't matter if there is another floating box to the left or right - put me on the first line you can". Here's an example showing the various different settings in combination.

Text text text text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text text
text text text text text text text More more more more more more
more more more more more more more more more more more more
more more more more more more more more more more more more more more
more more more more more more more more more more more more more more more more more
more more more more more more more more more more more more more more more more more
more more more more more more more more more more more more more more more more more
more more more more more more more more more more more more more more

```
<p border="1" font-size="12pt" padding="4">
 <div float="right" clear="right" width="40" height="40" background-color="yellow"/>
 <div float="right" clear="right" width="40" height="40" background-color="orange"/>
 <div float="right" clear="none" width="40" height="40" background-color="lightgreen"/>
 Text text text text...
 <div float="left" clear="none" width="40" height="40" background-color="pink"/>
 <div float="left" clear="none" width="40" height="40" background-color="lightblue"/>
 <div float="left" clear="none" width="40" height="40" background-color="khaki"/>
 <div float="left" clear="left" width="40" height="40" background-color="salmon"/>
 More more more more...
</p>
```

Text Attributes

There a several attributes that can be set to control how text is displayed in the document. Most of the “inline” elements defined above set one of these attributes to alter the style of text - for example, the `<b>` element is identical to `<span font-weight="bold">`. Almost all of these are taken from CSS2, and are in many cases identical to the values used in HTML. Full details for each attribute are defined in the Attribute reference section.

Attribute name	*Values*	*Description*
`font-family`	name of a font	Set the font face, e.g. “Times”, “Helvetica”, “monospace” or a user defined font. The CSS2 generic fonts “serif”, “sans-serif” and “monospace” are also recognised, and mapped to Times, Helvetica and Courier by default. Since version 1.0.14, it’s possible to specify more than one font-family, seperated by spaces or commas. This is commonly done in HTML to say “use the first font in this list that’s available”, but the actual meaning is “display each characters using the first font in the list that contains it”. This is particularly useful with PDF fonts - for example, setting `font-family="Times, Symbol"` would mean that text will be displayed in the Times Roman font if the character is available, otherwise the Symbol font will be used. This makes it easy to mix text from different fonts, eg. abcαβγ.
`font-style`	normal / italic / outline	Set the style of the font face - italic, outline or a combination, e.g “italic outline”.
`font-weight`	normal or bold	Set the weight of the font. Only two weights are recognized, normal and bold
`font-size`	size of the font	Set the size of the font. Can be “absolute”, (e.g. “12pt”) or “relative”, (e.g. “1.5em”, where 1em is the current size of the font). Other valid values, as defined in CSS2, are “larger” and “smaller”, as well as “xx-small”, “x-small”, “small”, “medium”, “large”, “x-large” and “xx-large”. “medium” is equivalent to 11pt.
`font-variant`	normal / small-caps	Set the font-variant - either `normal` (the default) or `small-caps`. The small-caps font is synthesized, so no explicit small-caps font is required. THIS TRANSFORMATION IS QUITE TIME CONSUMING, SO AVOID USING IT FOR LONG PHRASES.
`font-stretch`	normal / ultra-condensed / extra-condensed / condensed / semi-condensed / semi-expanded / expanded / extra-expanded / ultra-expanded	Set the horizontal stretching of the font. Note this attribute is not typographically correct, in that it simply stretches the text rather than choosing a variant of the typeface. This will result in wider or narrower vertical stems.
`line-height`	number	Set the spacing between successive lines of text - either “normal” to choose the spacing the font-designer recommended, a percentage (100% for line-height=font-size), or explicitly, eg “14pt”
`font`	font description	This shorthand property allows you to set the font family, size, style, weight, variant and line spacing with one attribute, eg. `"bold 12/14pt Times"`. See the CSS2 specification for a full description of this attribute.
`color`	color	Set the color of the font
`outline-color`	color	Set the color of the outline of the font, if it’s drawn

Attribute name	*Values*	*Description*
`outline-width`	number	Set the width of the outline of the font, if it's drawn
`text-indent`	number	Set the indentation of the first line of text in a paragraph. A positive number indents the first line to the right, a negative number to the left.
`text-decoration`	underline or line-through	Set the text decoration - underlined or struck out
`text-transform`	`normal` / `capitalize` / uppercase / lowercase	Set the text transformation - "capitalize" capitalizes the first letter of each word, and "uppercase" and "lowercase" transform the whole phrase accordingly.
`text-align`	left / right / center / justify	Set the alignment of the text within it's paragraph box. This is a standard CSS2 attribute, unlike it's HTML counterpart `align`. However, in an effort to preserve HTML compatibility, both parameters are accepted - if `text-align` isn't set, the value of `align` is used instead.
`letter-spacing`	number	Set the space between letters. A positive number moves letters further apart while a negative number moves them together. The default is zero
`justification-ratio`	number from 0 to 1	When text is justified, extra space is placed between letters and words to increase the overall length of the line. This parameter controls how much space is added between letters, and how much between words. A value of 0 means "only extend the spacing between words", while a value of 1 means "only extend the spacing between letters. The default is 0.5, which means add a bit to each. Note this setting has no effect if text is not justified - in that case, see the `letter-spacing` attribute.
`requote`	true or false	Whether to use "curly" quotes or "plain" quotes.
`suppress-ligatures`	true or false	Whether to automatically use the "fi", "fl" and "ffi" ligatures

Fonts

Built-in fonts

Every report created by the Report Generator can display the standard 5 fonts available in all PDF documents - **Times**, **Helvetica** and **`Courier`**, as well as the "Symbol" and "ZapfDingbats" fonts. Times, Helvetica and Courier can also be referred to by the generic CSS2 names of "serif", "sans-serif" and "monospace". The following two lines give identical results:

```
<body>
  <p>This is in <span font-family="Helvetica">Helvetica</span></p>
  <p>This is in <span font-family="sans-serif">Helvetica</span></p>
</body>
```

As well as the standard 5 fonts, users with the appropriate language version of Acrobat can access up to 7 further fonts to display Chinese, Japanese and Korean text. The names for these fonts are "**stsong**" (STSong-Light, simplified Chinese), "**msung**" (MSung-Light, traditional Chinese), "**mhei**" (MHei-Medium, traditional Chinese), "**heiseimin**" (HeiseiMin-W3, Japanese), "**heiseikakugo**" (HeiseiKakuGo-W5, Japanese), "**hygothic**" (HYGoThic-Medium, Korean) and "**hysmyeongjo**" (HYSMyeongJo-Medium, Korean).

Thanks to the native Unicode support of Java, XML and the Report Generator, creating reports with non-latin characters is easy. We'll cover more on this later, but here's a quick example of how to use a JSP to create a document showing the current date in Japanese

```
<?xml version="1.0"?>
<%@ page language="java" import="java.text.*"  contentType="text/xml; charset=UTF-8"%>
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">

<% DateFormat f = DateFormat.getDateInstance(DateFormat.FULL, Locale.JAPANESE)); %>

<pdf>
  <body font-family="HeiseiMin" font-size="18">
    Today is <%= f.format(new java.util.Date()) %>
  </body>
</pdf>
```

OpenType and WOFF fonts

One of the strengths of PDF documents is their ability to embed fonts into the document - both OpenType™ (also known as TrueType™) and Type 1 fonts can easily be embedded.

When embedding fonts, it's important to remember a key point about the PDF specification. Each font variation (there are four - normal, *italic*, **bold** and ***bold-italic***) is treated as a completely separate font. For the built in fonts, this isn't important, but when embedding a font authors need to remember that if even one letter is to be displayed in italic, two fonts will need to be embedded instead of one - the normal version and the italic.

OpenType fonts can be embedded using one or two bytes per glyph. Two bytes are recommended for any fonts that will be used to display glyphs outside the 8859-1 character set - Japanese, Chinese, Russian, Czech, Arabic and so on. The "bytes" attribute on the LINK element sets how many bytes are used - if not specified, it defaults to 1.

So how do you embed a font? Let's take as an example the Times Roman font, supplied with Microsoft Windows. It's an OpenType font, and there are four files that make up the font, one for each variation as described above.

```
<pdf>
  <head>
    <link name="mytimes" type="font" subtype="opentype" src="times.ttf" bytes="1"/>
  </head>
  <body font-family="mytimes" font-size="18">
     Hello in an embedded OpenType font
  </body>
</pdf>
```

This shows the basic setup embedding a single font variation (the value "truetype" can also be used as a synonum for "opentype"). Notice that when we link in the font we set the "name" attribute, which we then reference later in the document. But what do we do if we want it in italic as well?

```
<pdf>
  <head>
    <link name="mytimes" type="font" subtype="opentype"
          src="times.ttf" src-italic="timesi.ttf"/>
  </head>
  <body font-family="mytimes" font-size="18">
     Hello in an embedded, <i>italic</i> OpenType font
  </body>
</pdf>
```

By setting the “src”, “src-italic”, “src-bold” and “src-bolditalic” attributes in the LINK element we can have access to the entire range of styles in the font. If a variation isn’t used, it isn’t embedded in the document, so it doesn’t hurt to link in all the variations - the size of the document won’t be increased.

Two additional aspects of OpenType fonts can be set, both of which default to true. Whether the font is embedded in the document or just referenced by name is controlled by the “embed” attribute, and whether the font is subset or not is controlled by the “subset” attribute. Generally it’s best to leave these untouched.

OpenType Collections and WOFF fonts

WOFF fonts are a variation of the standard OpenType format, and since 1.1.63 we can load them in the same way as an OpenType. No change to the XML is required, although you can use “woff” as a subtype if you prefer - it is a synonym for “opentype”. The WOFF 2.0 format is **not yet supported** - we support WOFF 1.0 only.

We also support loading a particular font from an *OpenType Collection*. A collection is a number of fonts bundled into one file - as many of the datastructures are shared, this can sometimes save significant space overall and we see it most with fonts for the East Asian languages such as Chinese and Japanese.

To reference a particular font in an OpenType collection, add a fragment identifier of the form ‘#font=*n*’ to the font identifying which item in the collection you want, with the first font at index 1. For example, to load the second font in the collection:

```
<pdf>
  <head>
    <link name="mingliu" type="font" subtype="opentype" src="mingliu.ttf#font=2"/>
  </head>
  <body font-family="mytimes" font-size="18">MingLiU Proportional</body>
</pdf>
```

Type 1 fonts

Similar to OpenType fonts above, Type 1 fonts can be used too. These usually come as two separate files - an “AFM” file, describing the size of the characters, and a “PFA” or “PFB” file describing the actual characters themselves.

The AFM file must always be available, as otherwise the Report Generator won’t know the size of the characters or which characters are available in the font. The PFB file *should* always be included, but isn’t mandatory. Leaving it has the same effect as turning off embedding for OpenType fonts - if the font isn’t installed on the viewers computer, it will be approximated.

Here’s an example of how to embed a Type 1 font in the document.

```
<pdf>
  <head>
    <link name="BitstreamCharter" type="font" subtype="type1"
          src="charter.afm" pfbsrc="charter.pfb"/>
  </head>
  <body font-family="BitstreamCharter" font-size="18">
     Hello in an embedded Type 1 font
  </body>
</pdf>
```

Like OpenTypes, the italic, bold and bold-italic variants must be included separately, using the “src-italic”, “src-bold” and “src-bolditalic” for the AFM files, and “pfbsrc-italic”, “pfbsrc-bold” and “pfbsrc-bolditalic” for the PFB or PFA files.

Tables

The table syntax is almost identical to HTML with a few added features. Each table is a block (as described in the “box model” section above), with one or more rows (the TR element) containing several columns (the TH and TD elements).

Cells can span several columns or rows by setting the “colspan” and “rowspan” attributes. As each row and cell are also blocks, their margin, padding, border and backgrounds can be set separately (in CSS2 a row cannot have padding, margin or border set. We allow this, but only the vertical components - e.g. setting `<tr border="1">` only sets the top and bottom borders to 1. This is necessary to lay the table out correctly). Here’s an example:

```
<table width="100%" border="2">
  <tr>
    <td colspan="2" align="center">Countries and their foods</td>
  </tr>
  <tr background-color="#D0D0D0">
    <th>Country</th>
    <th>Food</th>
  </tr>
  <tr>
    <td>Wales</td>
    <td>Leek</td>
  </tr>
  <tr>
    <td>Argentina</td>
    <td>Steak</td>
  </tr>
  <tr>
    <td>Denmark</td>
    <td>Herring</td>
  </tr>
</table>
```

And here’s what it looks like.

Countries and their foods	
Country	Food
Wales	Leek
Argentina	Steak
Denmark	Herring

When migrating from HTML tables, you need to remember that the `border` directive sets the border for the entire table, rather than the border around each of it’s cells. To draw a border around every cell, you can either set the `border` attribute for each of them or set the `cellborder` option for the table. Likewise, the HTML attribute “cellspacing”, which set the margin for each cell, has been renamed to “cellmargin”.

Pagination with tables - headers and footers

When a table is too long to fit on a page, it may be broken into smaller tables that do fit (this can be prevented by setting the “page-break-inside” attribute - see Pagination for more detail). A common requirement when this happens is to reprint a standard header or footer row in the table.

This can be done using the THEAD, TBODY and TFOOT elements - also part of HTML, although not commonly used. These elements allow rows in the table to be assigned to the header, the body or the footer of the table. If the table is all on one page, this distinction isn't important, but if it's split over several pages this allows the Report Generator to reprint the headers and footers on each sub-table as required. Here's an example.

```
<table>
  <thead>
    <tr><td>Animal name</td><td>Habitat</td></tr>
  </thead>
  <tbody>
    <tr><td>Aardvark</td><td>Africa</td></tr>
    <tr><td>Ant</td><td>My Kitchen</td></tr>
    <tr><td>Anteater</td><td>South America</td></tr>
    <tr><td>Antelope</td><td>Africa</td></tr>
    <tr><td>Armadillo</td><td>South America</td></tr>
  </tbody>
</table>
```

If a row is added to a table directly (without being placed inside a THEAD, TBODY or TFOOT element), it's assumed to be inside the TBODY. The TH element, meant to represent a table header, is purely stylistic and is treated no differently to the TD element in terms of layout.

Table Layout algorithms

A table is laid-out according to one of two algorithms - which one is controlled by the setting of the "table-layout" attribute. The default is "auto", which means the table is laid out according to the "automatic" layout algorithm recommended in the CSS2 specification. This is the same as that used by most web browsers, where each cell is sized based both on it's content, any width or height that is specified and the size of the other cells in it's row and column.

The other option is "fixed", where the table is laid out according to the "fixed" layout algorithm from the CSS2 specification. This algorithm is slightly faster, as it sizes each cell based only on the "width" attributes, not on the cell contents. The TABLE element must have an explicit "width" attribute set, otherwise the layout algorithm defaults to auto.

There are some other subtle differences between HTML, CSS and the table model we use here. See the "table" entry in the Element and Attribute Reference section for more detail.

Lists

The Report Generator supports two types of list - "ordered" (specified by the OL element) and "unordered" (specified by the UL element). Each list contains one or more "list elements", specified by the LI element. The elements are printed on the page one after the other, often indented slightly and with a "marker" next to it. The marker is the only real difference between the two types of list. Here are a couple of examples demonstrating this - the only difference are the UL and OL elements:

```
<ul>
  <li>Item 1</li>
  <li>
    <ul>
      <li>Item 2.1</li>
      <li>Item 2.2</li>
    </ul>
  </li>
  <li>Item 3</li>
</ul>
```

- Item 1
  - Item 2.1
  - Item 2.2
- Item 3

```
<ol>
  <li>Item 1</li>
  <li>
    <ol>
      <li>Item 2.1</li>
      <li>Item 2.2</li>
    </ul>
  </li>
  <li>Item 3</li>
</ul>
```

1. Item 1
    1. Item 2.1
    2. Item 2.2
3. Item 3

In the ordered list, the “marker” are the arabic numerals starting at 1. In the unordered list, they’re the small bullets, or “discs”. These can be changed by setting the “marker-type” attribute of the list itself. Valid values can be either a literal or one of the following values.

Marker name	*Description*
`disc`	A round bullet (character U+2022). Unordered
`middle-dot`	A small round bullet (character U+00B7). Unordered
`decimal`	The arabic numerals starting at “1”
`lower-roman`	Lowercase roman numerals starting at “i”
`upper-roman`	Uppercase roman numerals starting at “I”
`lower-alpha`	Lowercase latin letters starting at “a”
`upper-alpha`	Uppercase latin letters starting at “A”
`circled-number`	Circled numbers from 1 to 20 (character U+2460 to U+2473)

If the marker-type is not one of these values, it’s printed literally. This can be used with good effect with a “dingbats” font. The font for the marker can be set using the “marker-font-family”, “marker-font-style” and “marker-font-weight” attributes, which do the same job for markers as “font-family”, “font-style” and “font-weight” do for normal text.

Also of note are the “marker-prefix” and “marker-suffix” attributes, which can be used to display a literal immediately before or after the marker. Here are some examples:

(1) Item 1
(2) Item 2
(3) Item 3

```
marker-type="decimal"
marker-prefix="("
marker-suffix=")"
```

a. Item 1
b. Item 2
c. Item 3

```
marker-type="lower-alpha"
```

i. Item 1
ii. Item 2
iii. Item 3

```
marker-type="lower-roman"
```

✗ Item 1
✗ Item 2
✗ Item 3

```
marker-type="&#x2717;"
marker-font-family="ZapfDingbats"
```

① Item 1
② Item 2
③ Item 3

```
marker-type="circled-number"  marker-suffix=""
marker-font-family="ZapfDingbats"
```

A useful feature which is missing in HTML is the ability to create *hierarchical* lists. This is most easily demonstrated.

```
<ol marker-type="upper-alpha" marker-hierarchy="true">
  <li>Item 1</li>
  <li>
    <ol marker-type="lower-roman">
      <li>Item 2.1</li>
      <li>Item 2.2</li>
    </ul>
  </li>
  <li>Item 3</li>
</ol>
```

A. Item 1
   B.i. Item 2.1
   B.ii. Item 2.2
C. Item 3

As you can see, the “marker-hierarchy” attribute allows nested lists to refer to the value of the parent list. The value specified by the “marker-hierarchy-separator” attribute is the literal (if any) to place between the nested elements, performing a similar job to “marker-prefix” and “marker-suffix”. It defaults to “.”.

The final setting relating to lists is the “marker-offset”. This is the distance away from the left edge of the list element to place the marker. Generally this is the same as the list elements “padding-left” attribute (which defines how far in the list element is nested), but it can be made smaller to indent the marker as well.

Images

The Report Generator can embed several different bitmap image formats - PNG, JPEG, GIF, PBM, PGM and TIFF. There are some restrictions however:

- Progressive JPEG images can only be read in Acrobat 4.x and greater, and as they’re larger as well they should be avoided and standard baseline JPEG’s used.
- Animated GIF images can be used, but only the first frame will be shown. GIF images may use any number of colors.
- Old-style JPEG, NeXT and Thunderscan TIFF image sub-formats are not supported, but these variations have been been seen in decades now.
- Transparency, including alpha transparency, is fully supported in the GIF and PNG image formats.
- Only the first image of multi-image PGM and PBM images will be used. ASCII encoded PNMs cannot be parsed.

The size of the image depends on the size in pixels of the bitmap, and the dots-per-inch (or DPI) it’s rendered at. As PDF is a print-based medium, there is no fixed “pixel size” which determines the size of the image. A 200 x 200 pixel bitmap at 200dpi will only take up one square inch - at 600dpi it takes a third of an inch.

The image DPI can be set by the “dpi” attribute, and defaults to the DPI set in the image. GIF, most PNG and the occasional TIFF image don’t specify a default DPI, in which case it defaults to 72 - which conveniently means that a 200 x 200 pixel bitmap takes 200 x 200 points on the page. Depending on the type of image this may not be high enough - it’s probably OK for photographs but hi-resolution line-art generally requires 200 to 300dpi to avoid appearing blocky when printed.

As well as using the “dpi” attribute, the width and height of an image can be set directly in the same way as for any other block - using the “width” and “height” attributes. The document author is responsible for making sure there is no change in aspect ratio.

```
<img src="canoe.jpg" dpi="200"/>
```

```
<img src="canoe.jpg" dpi="600"/>
```

The TIFF image format allows multiple pages as part of a single image. To select a specific page of a TIFF image, simply add a “#n” to the image URL. For example, to load the second page of a multi-page TIFF image, add the following XML to your document:

```
<img src="file:/path/to/myimage.tif#2"/>
```

URLs for the the image may be absolute or relative, in which case they’re relative to the base URL of the source file. Technically this is the System-ID of the `InputSource` the XML is being parsed from: this is generally the URL the XML is loaded from, although this is under the control of the software

The “alt” attribute can be set on an image, as in HTML, although this is typically not necessary. The only time it is required is if a PDF/UA document is being created.

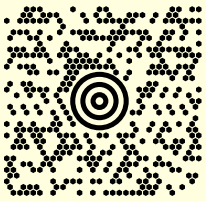
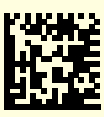
Barcodes

The Report Generator can print barcodes directly to the document using one of several barcode algorithms. This is generally more convenient than including bitmap representations of the barcode, and always results in smaller files. The size of the barcode depends on the value to be printed and the “bar-width” attribute (the width of the narrowest bar - may be set to values between 0.6 and 1). This means the the “width” element is ignored, and the height may be set within the limits imposed by the barcode algorithm - the minimum height is 15% of the width or 18 points, whichever is greater.

```
<barcode codetype="code128" showtext="true" value="My Value"/>
```

The value of the barcode is set by the mandatory “value” attributes, and the “showtext” attribute (which may be true or false) determines whether a human readable version of the value is printed below the code. The actual bar code algorithm used is set by the mandatory “codetype” attribute, and may be one of the following values. If a value contains characters outside the range that can be displayed by the selected code type, an error occurs.

Code name	*Description*	
`code128`	Code 128, a modern variable-width code. Can display ASCII values from 0x00 to 0xFF. Code128 has several variations, the package chooses the most compact one based on the data.	 Code 128
`code39`	Code 3 of 9. An older code, widely used but not terribly compact. Can display the symbols A to Z and digits, space - + $. % / *. May use the “bar-ratio” attribute.	 CODE 39
`code39checksum`	Code 3 of 9 with checkdigit. Identical to code39 but with a checkdigit added.	 CODE 39
`code25`	Interleaved Code 2 of 5. Can display digits only, but is fairly compact. May use the “bar-ratio” attribute.	 0123456789
`code25checksum / code25deutschenpost`	Interleaved Code 2 of 5 with checkdigit. Identical to `code25` but with a checkdigit added. The “code25deutschenpost” value can be used to select the checksum algorithm used by Deutschen Post in Germany for the Leitcode and Identcode symbols.	 0123456789

Code name	*Description*
`codabar`	CodaBar algorithm. Variable-width code used by Fed-Ex amongst others, the first and last symbols must be a stop code from A to D, and the symbols in the middle must be a digit or one of + - $ / : or the decimal point “.” 
`ean13 / upca`	EAN-13 - the international variable-width barcode used on groceries and books, always 13 digits long. The last digit is a checkdigit, which may or may not be specified. Generally EAN-13 codes should have their bar-width attribute set to 0.75, which makes the whole code one inch wide. The codetype “upca” may also be used to generate US-format UPC-A barcodes. These are identical except that the value must be 10 or 11 digits long. 
`ean8`	EAN-8 is an 8-digit barcode which is very similar to EAN-13 in design and purpose. It’s typically used where an EAN-13 barcode would be too large. 
`postnet`	PostNet algorithm, used by the US Postal Service to encode ZIP codes, so it only represents digits. The height and width of this code are fixed according to the specification, so the “width” and “height” attributes are ignored. 
`rm4scc`	Royal Mail 4-state Customer Code. A 4-state code used by the Royal Mail in the UK to encode postcodes. Like PostNet, the width and height of this code are fixed. This algorithm can encode digits and the upper-case letters A-Z. 
`intelligentmail`	The IntelligentMail® barcode, introduced in 2008 by the USPS to replace Postnet. It takes a 20, 25, 29 or 31 digit value and has a fixed width and height. 
`maxicode`	MaxiCode symbol. MaxiCode is a 2-D barcode invented by UPS but now in the public domain. These codes are different to the other barcode types in that they are always 80x80 points (the “width” and “height” attributes should be set to 80), and “showtext” is ignored. A MaxiCode symbol can encode up to 183 ISO-8859-1 characters of general text (extended error correction is used if space permits it), or for addressing a “Structured Carrier Message” can be specified. For an SCM the value must begin with “])>”, and the format must be as specified in section B.2 of the MaxiCode specification. 
`pdf417`	PDF417 is a “stacked” 2D barcode - probably the most common one. It’s used for a wide variety of purposes (for instance, paper archives of electronic invoices in Spain must use PDF417). The “width” and “height” attributes must be set but other attributes will be ignored. To target the earlier (partly incompatible) version of the spec, use a type of “pdf417:2001”, and to use ECI markers to identify the encoding explicitly (which is not supported by all readers), use “pdf417:eci” 
`qrcode`	QR-Code is a 2D barcode, invented and commonly used in Japan but making headway elsewhere too, due to it’s ability to store Kanji and it’s incredible density - the largest version can store over 6000 digits. The “width” and/or “height” attributes must be set but other attributes will be ignored. 
`datamatrix`	Data Matrix is another commonly used 2D barcode. By default a square datamatrix will be used, but if the width is a multiple of the height a rectangular code will be produced. 

Code name	*Description*
`databar`	GS1 Databar (formerly known as RSS-14) is a very compact, fixed size barcode that can represent 14 digits. Uniquely it requires no quiet zone, so you may want to consider adding margin or padding to barcodes using this symbol.
`aztec`	Aztec Code is a modern, compact 2D barcode that is visually quite similar to QR Code.
`deutchepostmatrix`	Deutsche Post have their own “post matrix” code, which is a variation on DataMatrix. The code has a fixed size and the “width” attribute should be left unspecified.

Two of the codes listed above - Code 3/9 and Interleaved 2/5 - are not “variable width” codes, and use just two bars - a thick bar and a thin bar. These algorithms may optionally use the “bar-ratio” attribute to specify the ratio between the width of thick and thin bars. Some knowledge of the algorithms limits are recommended if altering this value, which defaults to 2.8. If this attribute is specified for a variable-width barcode, it’s ignored.

Generic Blocks and Vector Graphics

Sometimes the need arises to group elements together inside a block - for example, to set the language or class for a number of elements, or to position several absolutely-positioned elements relative to the same point. There are several generic elements in the XML syntax, the most familiar one being DIV (short for *division*), which is also used in HTML.

The DIV element can contain other blocks as children - tables, paragraphs or other divs are common. A plain DIV by itself has no appearance on the page (although one can obviously be given by setting the background color and border, as for any block).

Performing an identical function to DIV but with a slight twist are the CIRCLE, ELLIPSE and SHAPE elements. As a block is just a rectangle on a page, these elements can be used to define the shape that’s drawn within this rectangle.

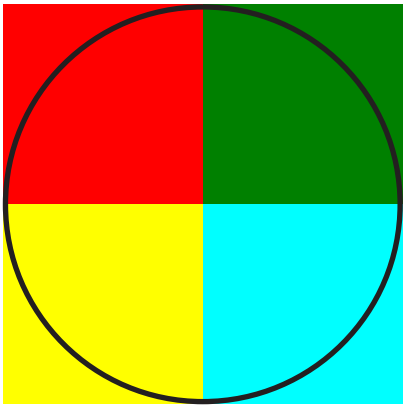
These can be used for interesting effect, especially as like a DIV they can also contain other blocks as children. For instance, a paragraph or image could be placed inside an ELLIPSE whose “overflow” attribute was set to “hidden” to give a porthole-like view on the contents, like the example to the right.

Note that the children of these shapes are not shaped to fit, merely trimmed - for example, a paragraph of text will still be rectangular, but this method allows only a portion of that rectangle to be seen.

These elements can also be used to draw diagrams, often by setting the “position” attribute to “absolute”. Here’s an example:

```
<style>
.pic div { position:absolute; width:75; height:75 }
</style>

<div class="pic" width="150" height="150">
  <div background-color="red"/>
  <div left="75" background-color="green"/>
  <div top="75" background-color="yellow"/>
  <div left="75" top="75" background-color="cyan"/>
  <ellipse width="150" height="150" border="2"/>
</div>
```



Ellipses

The ELLIPSE element takes the same attributes as a DIV - a “width” and “height” to specify the width and height of the ellipse.

Circles

The CIRCLE element is an alternative to the ELLIPSE. Instead of specifying the width and height, the mandatory “radius” attribute must be set to the radius of the circle. Unlike the ellipse, the “left” and “top” attributes specify the location for the center of the circle, not the top-left corner of the rectangle containing it. This can be confusing when the circle is relatively positioned, as it will appear to be misplaced - in this case the “left” and “top” attributes need to be set to the same value as the “radius”, or the ELLIPSE element used instead.

Shapes

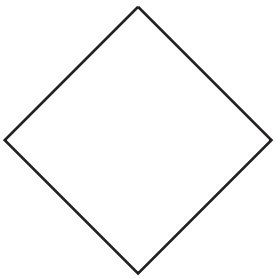
The SHAPE element allows a custom shape to be defined by a drawing lines, arcs and bezier curves. This shape may then be painted and/or may contain other blocks, which will be painted inside the shape - usually with the shapes “overflow” attribute set to “hidden” to clip it’s children to the bounds of the shape.

Each SHAPE element must contain a SHAPEPATH which defines the shape, and then may optionally contain other blocks like the DIV element. The SHAPEPATH defines the outline to draw, and may contain the following elements.

Element	Example	Description
moveto	`<moveto x="20" y="20"/>`	Moves the cursor to the specified location without marking the page.
lineto	`<lineto x="20" y="20"/>`	Draws a straight line to the specified location
arcto	`<arcto width="100" height="100" startangle="0" endengle="90"/>`	Draws an arc from an ellipse. The size of the ellipse is specified by the “width” and “height” attributes, and the section to draw is specified by the “startangle” and “endangle”. This example would draw an arc from the current cursor position to a position 50 points to the right and 50 points down the page.
bezierto	`<bezierto x="100" y="100" cx1="50" cy1="0" cx2="50" cy2="100"/>`	Draws a bezier curve to the location specified by “x” and “y”. cx1,cy1 is the location of the first control point and cx2,cy2 is the location of the second.

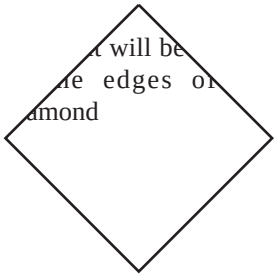
This is difficult to visualize so here are some examples. The first shows how to draw a diamond.

```
<shape width="100" height="100" border="1">
  <shapepath>
    <moveto x="50%" y="0%"/>
    <lineto x="100%" y="50%"/>
    <lineto x="50%" y="100%"/>
    <lineto x="0%" y="50%"/>
    <lineto x="50%" y="0%"/>
  </shapepath>
</shape>
```



If you then wanted to place some text inside this diamond, clipped to it's edges, you could do this:

```
<shape width="100" height="100" overflow="hidden">
  <shapepath>
    <moveto x="50%" y="0%"/>
    <lineto x="100%" y="50%"/>
    <lineto x="50%" y="100%"/>
    <lineto x="0%" y="50%"/>
    <lineto x="50%" y="0%"/>
  </shapepath>
  <p>
    This text will be clipped at the edge of the diamond.
  </p>
</shape>
```



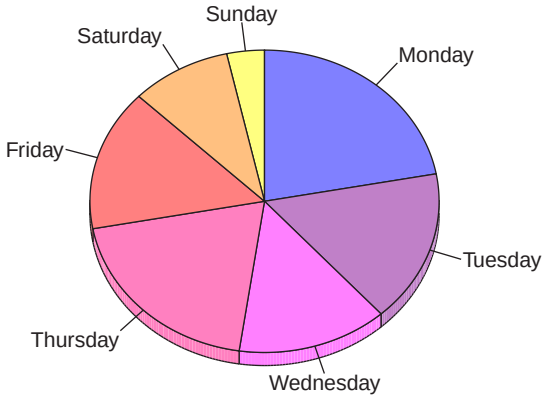
Graphs

The ability to plot inline graphs is a key feature of the Report Generator. The usual method of including graphical information (creating the graph as a bitmap using a separate package then including it as an image) has the disadvantages of increasing the size of the document and giving poor results, especially when compared to a vector based language like PDF.

With the Big Faceless Report Generator, graphs can be created using the same methods you would normally use to create a dynamic table (for example) - with a JSP or similar. The graphs are built using our Graph Library, which uses a 3D engine to create fully shaded, realistic graphs.

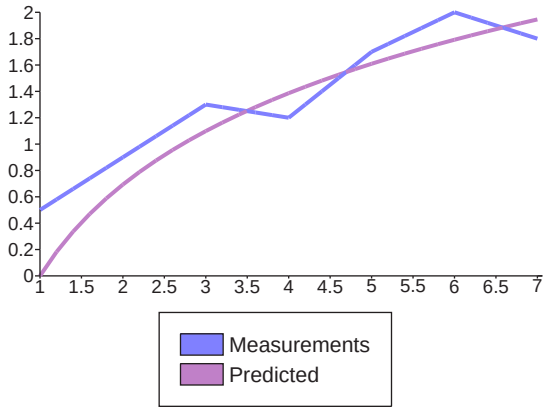
So how do you create a graph? Here's a simple Pie Graph to get you started.

```
<piegraph width="200" height="150"
 yrotation="30" display-key="flat-outer">
  <gdata name="Monday" value="19"/>
  <gdata name="Tuesday" value="14"/>
  <gdata name="Wednesday" value="12"/>
  <gdata name="Thursday" value="17"/>
  <gdata name="Friday" value="13"/>
  <gdata name="Saturday" value="8"/>
  <gdata name="Sunday" value="3"/>
</piegraph>
```



The library supports two broad categories of graph - those plotting *discrete* data, including Pie Graphs and Bar Graphs, and *continuous* data, ie; Line Graphs and Area Graphs. The discrete graphs all use the same pattern shown above, a graph element with one or more GDATA elements describing a (name, value) pair, whereas the continuous graphs focus on "curves" - a mathematical function created either from sampled values (stock prices over the year, for example) or pure functions (like a sine curve). These use either a DATACURVE or a SIMPLECURVE. Here's an example of both.

```
<linegraph width="200" height="150">
  <datacurve name="Measurements">
    <sample x="1" y="0.5"/>
    <sample x="2" y="0.9"/>
    <sample x="3" y="1.3"/>
    <sample x="4" y="1.2"/>
    <sample x="5" y="1.7"/>
    <sample x="6" y="2"/>
    <sample x="7" y="1.8"/>
  </datacurve>
  <simplecurve name="Predicted"
   method="java.lang.Math.log"/>
</linegraph>
```



The DATACURVE is made up of two or more SAMPLE elements, which have an “x” and “y” attribute relating to the point on the graph. The SIMPLECURVE takes the full name of a java method in it’s “method” attribute - this method must meet three criteria or an exception will be thrown:

1. It must be static
2. It must take a single double as its parameters
3. It must a return a double as its result

General Graph attributes

There are a very large number of attributes that can be set to control how the graphs appear - more than those used by all the other elements combined! These are detailed separately in the reference section, but we’ll go over some of them here too. The best way to try them out is to experiment, and to have a look at the **“graphs.xml”** example, supplied in the `example/samples` directory.

First, every graph is a block element, which means it can have padding, borders, a background color and all the other attributes appropriate for a block.

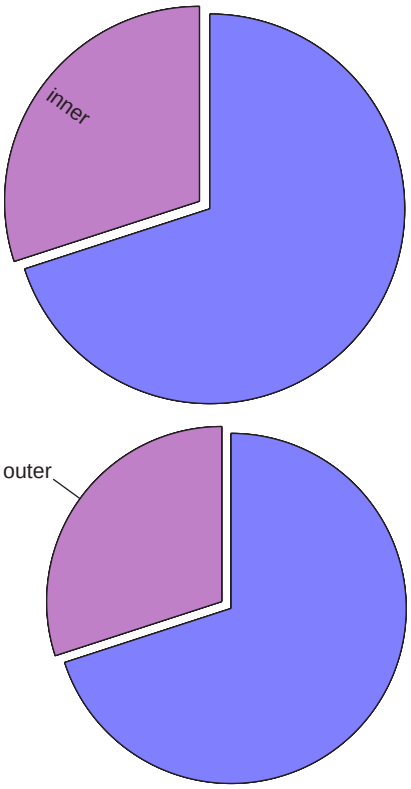
In addition, every graph can have the following attributes set (these are covered in more detail in the reference section).

Attribute	*value*	*Description*
`default-colors`	list of colors	The colors to use to display the graph, as a comma separated list. These are used in the order specified, and when the list is exhausted the sequence starts again from the beginning.
`xrotation`	an angle in degrees	The angle to rotate the graph around the X-axis. The X axis runs horizontally through the graph, from left to right.
`yrotation`	an angle in degrees	The angle to rotate the graph around the Y-axis. The Y axis runs vertically down the graph, from top to bottom.
`zrotation`	an angle in degrees	The angle to rotate the graph around the Y-axis. The Z axis goes “into” the document
`display-key`	none / right / bottom / top / left	Where to place the key relative to the graph. Pie Graphs have even moreoptions to choose from. The default is “bottom”.
`key-`*`attributename`*	font style	The style to give the font used to display the key. The attributes have the same names as those used for normal text (e.g color, font-family, font-size), but are prefixed with “key-” to make “key-color”, “key-font-family”, “key-font-size”, and so on.
`keybox-color`	color	The color to fill the box containing the key with. If the “display-key” value is not “top”, “right”, “bottom” or “left”, this value is ignored.

Attribute	*value*	*Description*
`keybox-border-color`	color	The border color to outline the box containing the key with. If the “display-key” value is not “top”, “right”, “bottom” or “left”, this value is ignored.
`light-level`	0 to 100	The intensity of the light used to simulate the shading on the graph. A value of 0 gives no shading at all, a value of 100 gives deep shadows. The default is 70.
`light-vector`	a vector, e.g. “(1,0,0)”	The direction of the light used to determine the shadows on the graph. The vector is specified as a vector of the form (X,Y,Z). The default is “(1,0,0)” which causes the light to appear to come from the right side of the graph.

Pie Graphs

The PIEGRAPH element is the only type of graph that isn’t plotted using axes. Pie graphs have a wider range of key types than the other graphs - as well as having the key placed in a box around the graph, pie graphs can have *inner* keys, where the name of the value is written directly on the slice, *outer* keys, where it’s written next to the relevant slice, or a combination.



The examples to the right are “rotated-inner-flat-outer” and “flat-outer” (in total there are 6 different options for “display-key” that are specific to Pie Graphs, so we won’t demonstrate them all here). Below are the list of valid options for “display-key” that are specific to Pie Graphs.

“display-key” value	*Description*
`flat-inner-flat-outer`	Put the label on the slice if it fits, or next to the slice if it doesn’t.
`flat-inner-rotated-outer`	Put the label on the slice if it fits, or rotate it and put it next to the slice if it doesn’t.
`rotated-inner-rotated-outer`	Put the label on the slice if it fits, or next to the slice if it doesn’t. Rotate it to the same angle as the slice regardless.
`rotated-inner-flat-outer`	Put the label on the slice and rotate it to the same angle if it fits: otherwise, put it next to the graph and don’t rotate it.
`flat-outer`	Put the label next to the slice
`rotated-outer`	Put the label next to the slice and rotate it to the same angle as the slice

When we say “if it fits” above, this is not to be taken literally (after rotation on three axes the math to determine this is well beyond us). Instead, the “outer-key-percentage” attribute can be set to the minimum percentage of the pie a slice can be before it is considered too narrow for an inner key.

Slices from the pie can be “extended” away from the center of the graph, like we’ve done above. This can be done by setting the “extend” attribute on the GDATA element to the percentage of the radius of the pie to extend the slice. The examples here have the purple slice set to “10”.

Another useful feature of Pie graphs is the “other” slice. The Report Generator can automatically group values below a certain size, to prevent the graph becoming too cluttered. The “other-percentage” sets the threshold (it defaults to zero) and the “other-label” is the label to use, which defaults to the word “other”.

A further problem wih Pie Graphs is what to do when the value to be plotted is zero. Of course, having a slice of Pie that’s 0% of the whole doesn’t make sense in the real world, but occasionally it’s useful to be able to indicate that the value *might* have been present.

To control this, the “display-zeros” attribute can be set to `true` or `false` - if true, the zero values will be displayed as infinitely small slices. If false, they will be skipped completely (the default)

Axes Graphs

Every graph other than the Pie Graph is plotted against two or more *axes* - and consequently they all have several attributes in common. First, we need to define some terms.

- A **formatter** determine how the values printed on an axis are displayed - as currencies, dates, integers and so on.
- A **label** is the name of the axis, like “day of week” or “number of units”. Labels on axes are optional, and off by default.
- A **style** is the name given to a group of attributes which together define how a text element in the graph appears. Like the “key” attributes in the general graph attributes section, these always have a common prefix followed by the name of a text attribute - for example “xaxis-font-family” and “xaxis-color” set the style of the values printed on the X axis, in the same way that “font-family” and “color” set the style of normal text in the document. Valid suffixes are:
  - **color** - the color of the text
  - **font-family** - the font family of the text
  - **font-style** - the font style of the text (“normal” or “italic”)
  - **font-weight** - the font weight of the text (“normal” or “bold”)
  - **font-size** - the font size of the text, in points
  - **align** - the horizontal alignment of the text (“left”, “center” or “right”)
  - **valign** - the vertical alignment of the text (“top”, “middle” or “bottom”)
  - **rotate** - the angle to rotate the text, in degrees clockwise.

With these definitions out of the way, we can list several attributes which are common to all graphs plotted on an axis.

Attribute	*Value*	*Description*
`xaxis`	style	The “xaxis-” group of attributes set the style to display the values plotted on the X axis - e.g. “xaxis-color” or “xaxis-font-family”. The default is black 7pt Helvetica.
`yaxis`	style	The “yaxis-” group of attributes set the style to display the values plotted on the Y axis - e.g. “yaxis-color” or “yaxis-font-family”. The default is black 7pt Helvetica.
`xaxis-formatter`	formatter	The formatter to use to display the values on the X axis. See below for more on formatters
`yaxis-formatter`	formatter	The formatter to use to display the values on the Y axis. See below for more on formatters
`xaxis-formatter-density`	`normal` `sparse` minimal	The “density” of the X axis formatter. See below for more on formatters
`yaxis-formatter-density`	`normal` `sparse` minimal	The “density” of the Y axis formatter. See below for more on formatters
`xaxis-label`	style	The “xaxis-label-” group of attributes set the style to display the label given to the X axis - e.g. “xaxis-label-color” or “xaxis-label-font-family”. The default is black 10pt Helvetica.
`yaxis-label`	style	The “yaxis-label-” group of attributes set the style to display the label given to the Y axis - e.g. “yaxis-label-color” or “yaxis-label-font-family”. The default is black 10pt Helvetica.

Attribute	*Value*	*Description*
`floor-color`	color	Set the color to draw the floor of the graph. The *floor* is the plane where *y=0* or where *y=min(y)* - for most graphs this is where *y=min(y)* but for line graphs this depends on the value of the “xaxis-at-zero” attribute. Defaults to “none”.
`floor-border-color`	color	Set the color to draw the grid on the floor of the graph. Defaults to “none”
`floor-grid`	color	Set which lines to draw on the grid on the floor of the graph. Valid values are `horizontal`, `vertical` or a combination of the two, e.g. `horizontal+vertical` (the default).
`ywall-color`	color	Set the color to draw the Y wall of the graph. The *Y wall* is the plane where *x=0* or where *x=min(x)* - for most graphs this is where *x=min(x)* but for line graphs this depends on the value of the “yaxis-at-zero” attribute. Default to “none”.
`ywall-border-color`	color	Set the color to draw the grid on the Y wall of the graph. Defaults to “none”
`ywall-grid`	color	Set which lines to draw on the grid on the Y wall of the graph. Valid values are `horizontal`, `vertical` or a combination of the two, e.g. `horizontal+vertical` (the default).
`zwall-color`	color	Set the color to draw the Z wall of the graph. The *Z wall* is the “back wall” of the graph. Defaults to “none”.
`zwall-border-color`	color	Set the color to draw the grid on the Z wall of the graph. Defaults to “none”
`zwall-grid`	color	Set which lines to draw on the grid on the Z wall of the graph. Valid values are `horizontal`, `vertical` or a combination of the two, e.g. `horizontal+vertical` (the default).
`axes-color`	color	Set the color to draw the axes lines in. Default is black
`box-color`	color	Set the color to draw the (optional) box around the entire graph. The default is “none”, so no box is drawn.
`min-y`	number	The minimum value to plot on the Y axis. Can be used to just display the top of bar graphs or area graphs.
`max-y`	number	The maximum value to plot on the Y axis. Can be used to increase the space above the top of the bars in a bar graph, for example.

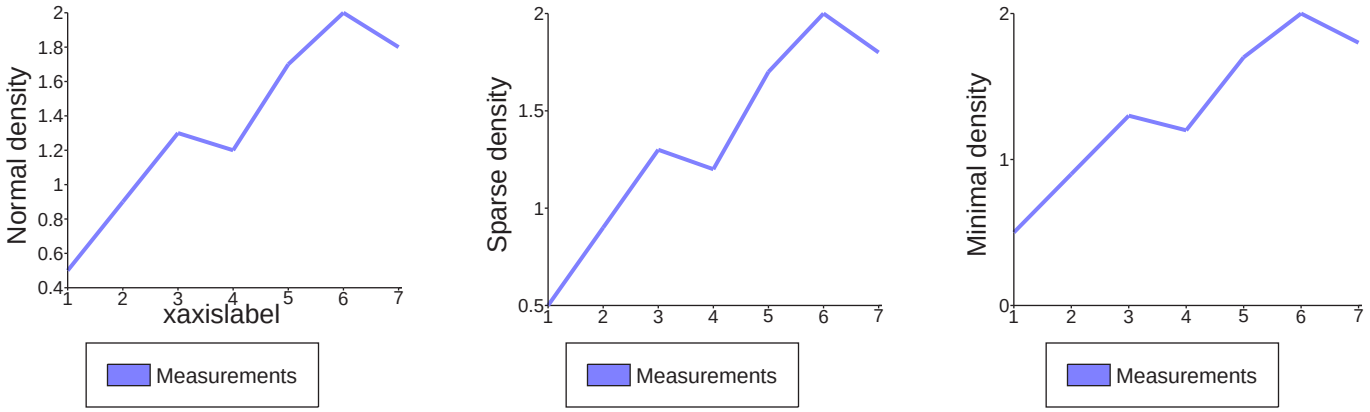
Formatting values on the axes

The X and Y axis values are displayed using a *formatter*. The default depends on the data being plotted, but is always either “integer()” or “floatingpoint()”. The “xaxis-formatter” and “yaxis-formatter” can be set to one of the following values:

Formatter	*Description*
`none`	Don’t plot any values on this axis
`integer()`	Plot the values on the axis as integers
`percentage()`	Plot the values on the axis as percentages
`floatingpoint()`	Plot the values on the axis as floating point values
`floatingpoint(`*`min`*`,`*`max`*`)`	Plot the values on the axis as floating point values. The *min* and *max* values are the minimum and maximum number of decimal places to display.
`percentage(`*`numdp`*`)`	Plot the values on the axis as percentages. The number of decimal places is specified by the *numdp* attribute.
`currency()`	Plot the values as currency values. The currency format depends on the “country” part of the locale of the graph, as set by the “lang” attribute.
`currency(`*`locale`*`)`	Plot the values as currency values. The locale of the currency format is specified explicitly - e.g. “en_GB” or “de_DE”.
`simple(`*`format`*`)`	Plot the values using a `java.text.DecimalFormat`. The *format* attribute specifies the format to use - e.g. “#0.0” to plot values that always have one decimal place.
`date()`	Plot values on the axis as a date, using the format “dd MMM yyyy” (only used with Line and Area graphs - see their entries for more information).
`date(`*`format`*`)`	Plot values on the axis as a date, using the specified `java.text.SimpleDateFormat` (only used with Line and Area graphs - see their entries for more information).

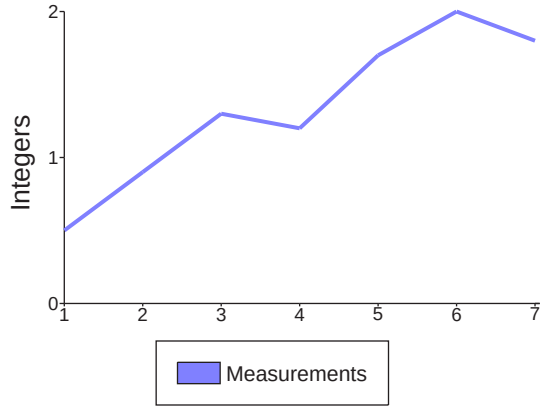
If these aren’t enough, a *custom formatter* can be written in java and referenced from the Report Generator by specifying it’s full class name. For example, `<bargraph xaxis-formatter="com.mycompany.CustomFormatter()">`. All formatters are subclasses of the `org.faceless.graph.Formatter` class, which is described more fully in the API documentation.

Additionally, the “density” of the formatter can be specified. This is an indication of how many values are to be plotted on the graph. This is set using the “xaxis-formatter-density” and “yaxis-formatter-density” attributes - values can be “normal”, for between 8 and 14 values on the axis, “sparse” for between 4 and 7 values on the axis, and “minimal” for either 3 or 4 values on the axis, depending on data. Here’s an example showing the differences on the Y-axis.

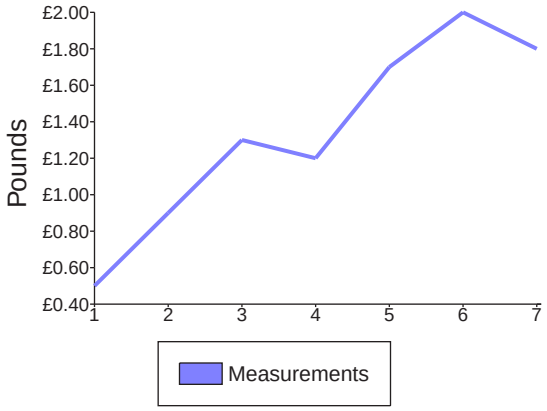


And here are some examples of the different types of formatters. The various `date()` formatters will be dealt with separately below, as they are specific to line and area graphs.

```
<linegraph width="200" height="150"
 yaxis-formatter="integer()"
 yaxis-label="Integers">
  <datacurve name="Measurements">
    <sample x="1" y="0.5"/>
    <sample x="2" y="0.9"/>
    <sample x="3" y="1.3"/>
    <sample x="4" y="1.2"/>
    <sample x="5" y="1.7"/>
    <sample x="6" y="2"/>
    <sample x="7" y="1.8"/>
  </datacurve>
</linegraph>
```



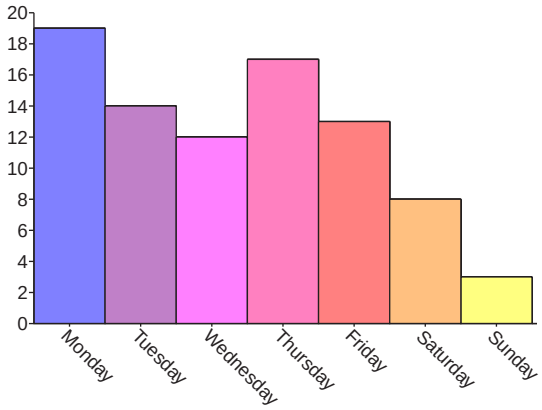

```
<linegraph width="200" height="150"
 yaxis-formatter="currency(en_GB)"
 yaxis-label="Pounds">
  <datacurve name="Measurements">
    <sample x="1" y="0.5"/>
    <sample x="2" y="0.9"/>
    <sample x="3" y="1.3"/>
    <sample x="4" y="1.2"/>
    <sample x="5" y="1.7"/>
    <sample x="6" y="2"/>
    <sample x="7" y="1.8"/>
  </datacurve>
</linegraph>
```



Bar Graphs

The simplest and most familiar type of bar graph can be created using a BARGRAPH element, which creates a single row of simple bars. Here's an example:

```
<bargraph width="200" height="150" xaxis-rotate="45">
  <gdata name="Monday" value="19"/>
  <gdata name="Tuesday" value="14"/>
  <gdata name="Wednesday" value="12"/>
  <gdata name="Thursday" value="17"/>
  <gdata name="Friday" value="13"/>
  <gdata name="Saturday" value="8"/>
  <gdata name="Sunday" value="3"/>
</bargraph>
```



If you compare this with the pie graph example shown above, you'll notice that other than the name of the element and a couple of attributes, the XML is almost identical. We've set the "xaxis-rotate" attribute to rotate the values on the X axis to 45 degrees - useful for longer values.

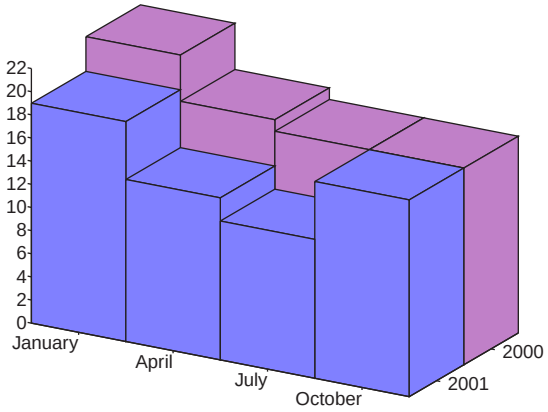
Four attributes common to all variants of bar graphs are "bar-depth", "bar-width", "round-bars" and "display-barvalues". The first two set the size of the bar relative to the square that it sits on, and both default to 100%. The "round-bars" attribute can be set to true or false, and if true turns each bar from a box into a cylinder - a nice effect, although it takes a little longer to draw. The "display-barvalues" attribute allows the value of the bar to be plotted directly on or above the bar - values can be either "top" to display the

value above the bar, “middle” to display it in the middle of the bar, “insidetop” to display the value at the end of but just inside the bar, or “none” to not display it at all (the default).

Depth Bar Graphs

For more than one set of data, the simple BARGRAPH shown above can’t cope, and it’s necessary to turn to one of the other three options. The first is a DEPTHBARGRAPH, which plots the different sets behind each other. To be effective, this graph really needs to be shown in 3D. We’ve also set the “xaxis-align” attribute to “right”, which is effective with 3D rotation. Here’s an example: Notice the “name2” attribute on the GDATA elements. This sets the name of the values on the second axes, and is used with TOWERBARGRAPH and MULTIBARGRAPH graphs as well.

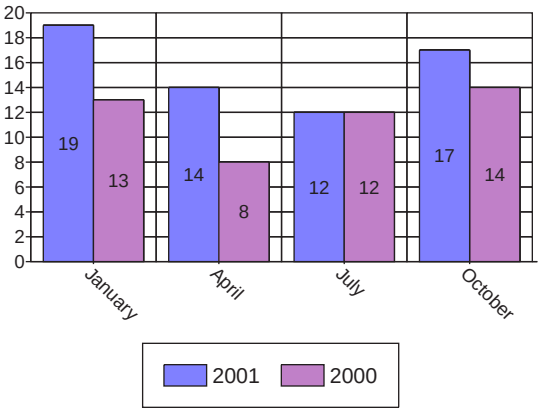
```
<depthbargraph width="200" height="150"
 xaxis-align="right" xrotation="30" yrotation="30">
  <gdata name="January" name2="2001" value="19"/>
  <gdata name="April" name2="2001" value="14"/>
  <gdata name="July" name2="2001" value="12"/>
  <gdata name="October" name2="2001" value="17"/>
  <gdata name="January" name2="2000" value="22"/>
  <gdata name="April" name2="2000" value="18"/>
  <gdata name="July" name2="2000" value="17"/>
  <gdata name="October" name2="2000" value="17"/>
</depthbargraph>
```



Multi Bar Graphs

When 3D isn’t an option the DEPTHBARGRAPH isn’t very effective, and a MULTIBARGRAPH is a better choice. This plots several narrow columns next to each other on the one axis, but other than that is identical in function to the DEPTHBARGRAPH. We’ve set the “zwall-border-color” so you can see more clearly where the divisions between values are.

```
<multibargraph width="200" height="150"
  xaxis-rotate="45" zwall-border-color="black"
  bar-width="80%" display-barvalues="middle">
   <gdata name="January" name2="2001" value="19"/>
   <gdata name="April" name2="2001" value="14"/>
   <gdata name="July" name2="2001" value="12"/>
   <gdata name="October" name2="2001" value="17"/>
   <gdata name="January" name2="2000" value="22"/>
   <gdata name="April" name2="2000" value="18"/>
   <gdata name="July" name2="2000" value="17"/>
   <gdata name="October" name2="2000" value="17"/>
</multibargraph>
```

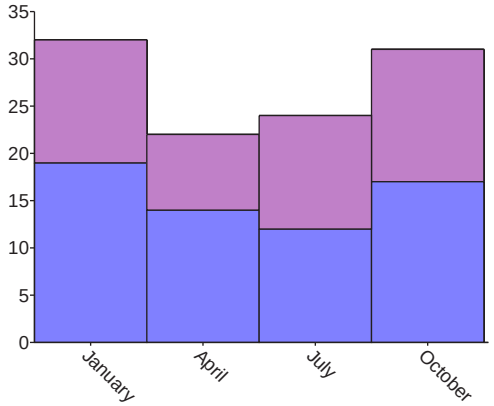


Notice the “name2” attribute on the GDATA elements. This sets the name of the values on the second axes, and is used with TOWERBARGRAPH and MULTIBARGRAPH graphs as well. We also set the “display-barvalues” attribute to middle and slightly reduced the bar-width, which can help the legibility of this type of graph.

Tower Bar Graphs

The third option for plotting bargraphs is a TOWERBARGRAPH, which is more useful for showing cumulative values than DEPTHBARGRAPH or MULTIBARGRAPH. Again note the “name2” attribute on the GDATA elements sets the second axes.

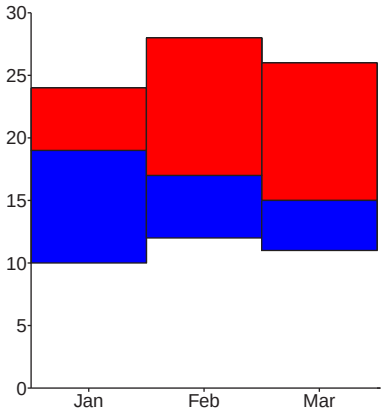
```
<towerbargraph width="200" height="150" xaxis-rotate="45">
  <gdata name="January" name2="2001" value="19"/>
  <gdata name="April" name2="2001" value="14"/>
  <gdata name="July" name2="2001" value="12"/>
  <gdata name="October" name2="2001" value="17"/>
  <gdata name="January" name2="2000" value="13"/>
  <gdata name="April" name2="2000" value="8"/>
  <gdata name="July" name2="2000" value="12"/>
  <gdata name="October" name2="2000" value="14"/>
</towerbargraph>
```



Floating Bar Graphs

The final option for plotting bargraphs is a FLOATINGBARGRAPH. Each bar in a floating bar-graph has two halves - the intention is to show a minimum, a middle value (often an average) and a maximum. The positions on the bar are specified with the `min-value`, `mid-value` and `max-value` attributes.

```
<floatingbargraph width="140" height="150">
  <gdata name="Jan" min-value="10" mid-value="19" max-value="24"/>
  <gdata name="Feb" min-value="12" mid-value="17" max-value="28"/>
  <gdata name="Mar" min-value="11" mid-value="15" max-value="26"/>
</floatingbargraph>
```



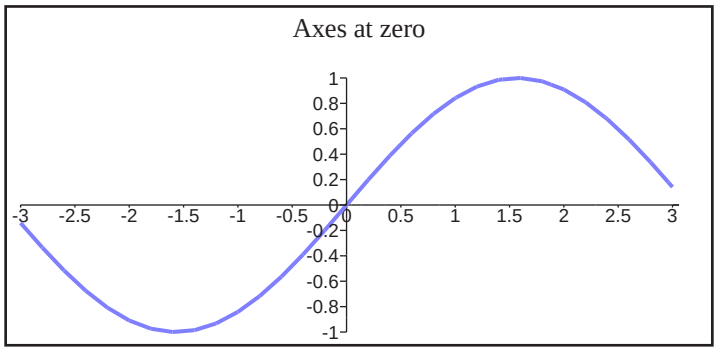
Bar graphs are unique amongst the different graphs in that they can use a “gradient fill” to display the colors. See the Colors section for more information

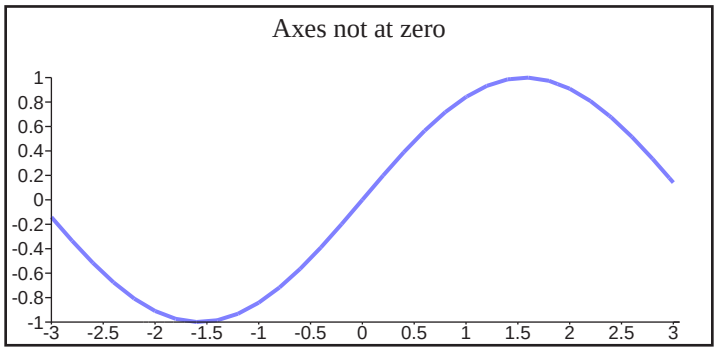
Line Graphs

The LINEGRAPH element allows one or more “curves” to be plotted against an X and Y axis. We’ve already described the curves above, so in this section we’ll focus on attributes specific to the LINEGRAPH element.

First up is an attribute specific to the LINEGRAPH element - the “line-thickness” attribute, which sets the thickness of the line used to draw each curve. This attribute is unique in that it only has an effect if the graph is plotted in 2D (i.e. xrotation, yrotation and zrotation are all zero). The default is 1.5. Similar in purpose but for 3D graphs is the “curve-depth” attribute, which controls how “deep” into the page the curve is drawn. This defaults to 1, and applies to both LINEGRAPH and AREAGRAPH.

There are several attributes which control the range of the axes on a LINEGRAPH. The “xaxis-at-zero” and “yaxis-at-zero” attributes control where the X and Y axis values (and the “ywall” and “floor” attributes discussed in “Axes Graphs”, above) are drawn. These boolean attributes both default to “true”, which means that the axes are drawn where *x=0* and *y=0*, even though this may be in the middle of the graph. Notice the difference with this sine curve.

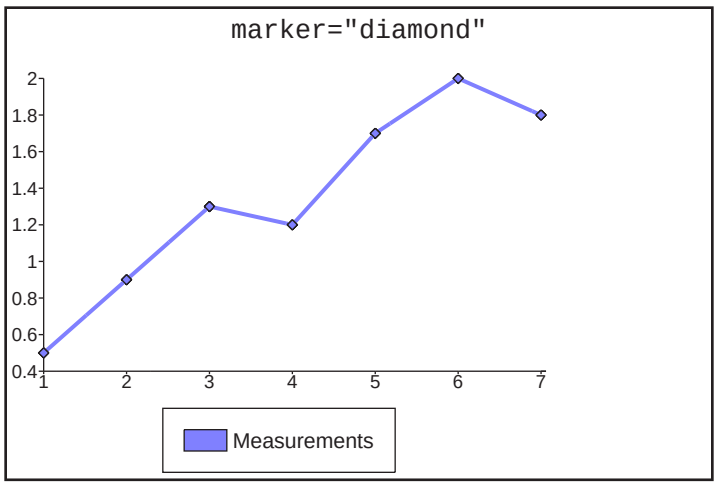


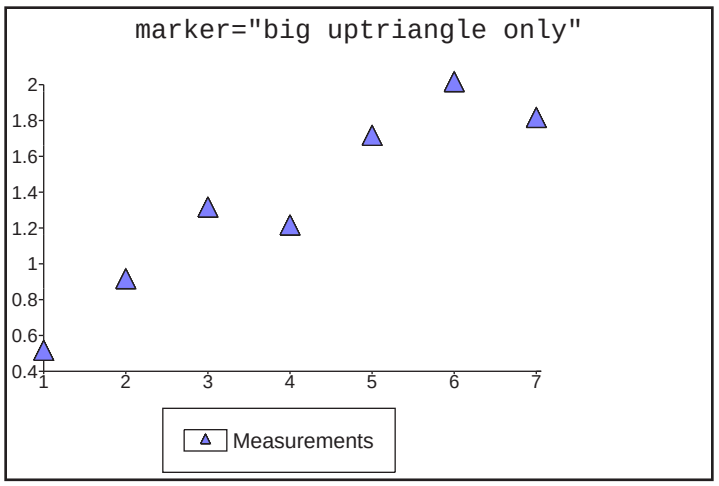


Next, an attribute which applies to LINEGRAPH and AREAGRAPH and is specific to plotting DATACURVES. The “max-data-points” attribute allows the number of points actually plotted to be limited to a fixed number. This is most commonly done for speed - if your database query returns 1000 elements to be plotted on the graph, but it’s only 2 inches wide, this could be set to a value (say 100) which would cause only every tenth data sample to be retained. By default this is set to 100.

Likewise for SIMPLECURVE elements, the “function-smoothness” attribute can be used to set the number of samples taken when drawing a SIMPLECURVE. This defaults to 30, which is generally adequate, but may be set to any value.

When plotting DATACURVE curves on a LINEGRAPH, markers may be placed at each data sample by setting the “marker” attribute of the DATACURVE. This can be set to either “none” (no marker, the default), “line” (which simply draws a line across the curve where the value is, or “circle”, “square”, “diamond”, “octagon”, “uptriangle”, or “downtriangle”, which place the specified marker at each sample as you’d expect. These values can optionally be prefixed with “big”, to double the size of the marker, “small” to reduce the size, or suffixed with “noborder” to remove the black border around the markers or “only”, to draw *just the marker*, not the lines connecting them. Example combinations include “circle”, “circle only”, “small diamond noborder only” or “big uptriangle only” - here’s what that looks like.





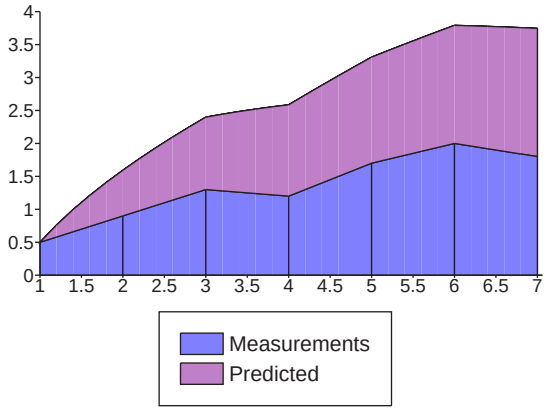
The “line” option is really only useful for 3D line graphs, and results in a black line across the curve where the sample is - similar to the “segments” described in the AREAGRAPH section below.

When a marker is used on a curve, it may optionally be added to the key by adding the “-with-markers” suffix to the key type. The graph on the right shown above has the “display-key” value set to “bottom-with-markers”, while the graph on the left has the attribute simply set to “bottom”.

Area Graphs

The AREAGRAPH is very similar to the LINEGRAPH, but is more appropriate for displaying cumulative data as the curves are stacked on top of each other. Many of the AREAGRAPH attributes are described in the LINEGRAPH section above, as they apply to both.

```
<areagraph width="200" height="150">
  <datacurve name="Measurements">
    <sample x="1" y="0.5"/>
    <sample x="2" y="0.9"/>
    <sample x="3" y="1.3"/>
    <sample x="4" y="1.2"/>
    <sample x="5" y="1.7"/>
    <sample x="6" y="2"/>
    <sample x="7" y="1.8"/>
  </datacurve>
  <simplecurve name="Predicted"
   method="java.lang.Math.log"/>
</areagraph>
```



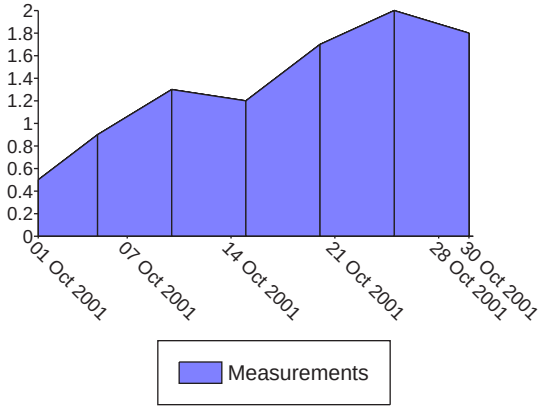
As you can see, both DATACURVE and SIMPLECURVE can be mixed on the same graph. Noticable on the DATACURVE though are the black lines dividing the curve into “segments”, and showing where the sample values are. These may not always be desirable, and can be removed by setting “draw-segments” attribute to “false”.

The other difference between this and the LINEGRAPH is that the data values at each point are added together. This is the result of the “cumulative” attribute, which defaults to true. Occasionally you may be working with pre-accumulated values, in which case setting this attribute to “false” turns off this behaviour.

Plotting Dates

Both the LINEGRAPH and AREAGRAPH support plotting dates on the X axis, instead of numeric values. This can be done by setting the `xaxis-formatter` to “date()” and the “x” attribute for the SAMPLE element to a valid date - recognized formats include RFC822 (e.g. `"Mon, 18 Feb 2002 17:26:18 +0100"`) and ISO8601 (e.g. `"2001-02-18"`, `"2001-02-18T17:26"` or `"2001-02-18T17:26:18+0100"`), although the recommended format is ISO8601. In this example we’ve also set the “xaxis-formatter-density” to “sparse”, although this is optional. Here’s how:

```
<areagraph width="200" height="150" xaxis-rotate="45"
 xaxis-formatter="date()"
 xaxis-formatter-density="sparse">
  <datacurve name="Measurements">
    <sample x="2001-10-01" y="0.5"/>
    <sample x="2001-10-05" y="0.9"/>
    <sample x="2001-10-10" y="1.3"/>
    <sample x="2001-10-15" y="1.2"/>
    <sample x="2001-10-20" y="1.7"/>
    <sample x="2001-10-25" y="2"/>
    <sample x="2001-10-30" y="1.8"/>
  </datacurve>
</areagraph>
```



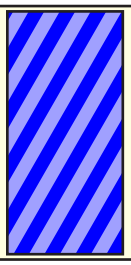
Internally the date is converted to a number value, so if the “xaxis-formatter” isn’t set the results will look fairly strange. The smallest unit of resolution with dates is a second, and using this method date ranges of between 2 seconds and 100 years can be plotted.

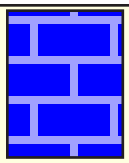
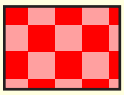
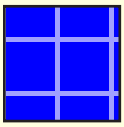
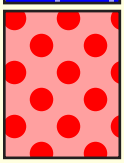
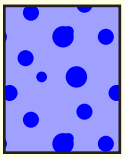
Specifying Colors

Colors can be specified in the XML in a number of ways. Most people are familiar with the #RRGGBB notation, which specifies a color in the default RGB colorspace used by the Report Generator. Colors can be specified in the following ways:

Color Example	*Description*
`none or transparent`	Specify no color is to be used
`#FF0000`	Specify a color in the documents RGB colorspace by setting the red component to 0xFF and the green and blue components to 0x00. Each component is on a scale from 0 to 255, so 0xFF is 100% red.
`rgb(100%, 0, 0)`	Another method of specifying an RGB color, this is identical to #FF0000. May also be specified as `rgb(255,0,0)`
`gray(100%)`	Specify a color in the documents GrayScale colorspace. `gray(0%)` is black and `gray(100%)` is white
`cmyk(100%, 0%, 0%, 0%)`	Specify a color in the documents CMYK colorspace. The example here would set the color to cyan.
`black`	Specify a named color - one of the list of 140 named colors in the current RGB colorspace (the list is the same list as used by HTML). The full list of named colors is in the reference section and also in the “colors.pdf” document in the `docs` directory of the package.
`spot(“name”, cmyk(100%,72%,0%,6%))`	Specify a spot color. Requires the name of the color, and the fallback color to use if it’s unavailable
`spot(“name”, cmyk(100%,72%,0%,6%), 50%)`	Specify a spot color and the intensity of that color. Requires the name of the color, and the fallback color to use if it’s unavailable, and the “intensity”, or how much of that ink to use. Values can range from 100% (which is the same as the 2 argument form above) to 0% (which is the same as transparency).
`alpha(50%, #FF0000)`	Specify a semi-transparent version of a color (since 1.1.10). The first parameter is the alpha value to use - 100% for a completely opaque color, 0% for a completely transparent one. The second parameter may be any one of the types of color listed above - so `alpha(50%, spot("Blue", cmyk(100%,72%,0%,6%)))` is a valid color, if a somewhat extreme example. Translucent colors will only work with Acrobat 5 or later.

As well as the “plain” colors defined above, the Report Generator can use “pattern” colors when drawing text or as the background color of an element. Document authors can choose from one of 8 different predefined patterns. Each pattern has a name, a foreground color (indicated by “fg” in the following table) and a background color (indicated by “bg”), and then possibly more attributes depending on the pattern. Here’s the list.

`pattern(stripe,`*`fg`*`,`*`bg`*`,`*`fgwidth`*`,`*`bgwidth`*`,`*`ang`*`)`		The `stripe` pattern creates a striped color pattern. The angle and width of each stripe can be set separately - the width of the stripe in the foreground color is set by *fgwidth* and the width in the background color is set by *bgwidth*. The angle is set by *ang*, and is specified in degrees clockwise from 12 o’clock.

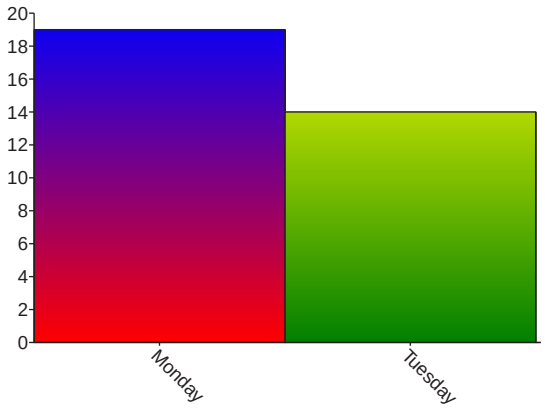
`pattern(brick,`*`fg`*`,`*`bg`*`,`*`width`*`,`*`height`*		This `brick` pattern creates a brickwork pattern (using the "running bond" style of bricklaying, for what it's worth). The width and height of each brick must be specified.
`pattern(check,`*`fg`*`,`*`bg`*`,`*`size`*`)`		The `check` pattern requires the size of each square in the check to be specified.
`pattern(grid,`*`fg`*`,`*`bg`*`,`*`linewidth`*`,`*`spacewidth`*`)`		The `grid` pattern creates a gridded pattern as shown here. The width of the line and the width of the space between the lines must be specified.
`pattern(spot,`*`fg`*`,`*`bg`*`,`*`size`*`)`		A "spot" pattern similar to the pattern used for halftoning in newspapers can be created with the `spot` pattern. The size of each spot must be specified.
`pattern(polka,`*`fg`*`,`*`bg`*`,`*`size`*`)`		A different kind of "spot" pattern, containing a number of different size random spots can be created with the `polka` pattern. The average size of the spots must be specified.
`pattern(star,`*`fg`*`,`*`bg`*`,20)`		Finally, a pattern of repeating 5-pointed stars (like those on the US flag) can be created with the `star` pattern. The size of each star must be specified.

Here's an example of how to use a pattern as the background color, and a spot color as the foreground for a page header.

```
<body background-color="pattern(stripes,#FFF0F0,#F0E0E0,5,5,45)">
  <h1 color="spot('PANTONE Reflex Blue CVC', cmyk(100%,72%,0%,6%))">
    Heading in Reflex Blue
  </h1>
  <p>
    The pages of this document have light pink stripes
  </p>
</body>
```

Finally, for Bar Graphs only, it's possible to fill a bar using a *gradient* fill. Specifying a color as `gradient(red, blue)` would cause the bar in the graph to smoothly change from red to blue as the value increased. Here's an example:

```
<bargraph width="200" height="150" xaxis-rotate="45">
  <gdata color="gradient(red,blue)"
         name="Monday" value="19"/>
  <gdata color="gradient(green,yellow)"
         name="Tuesday" value="14"/>
</bargraph>
```



Color Spaces

Each item in the document has three types of colorspace it can work with - RGB, CMYK and a GrayScale colorspace. By default, the RGB colorspace is the sRGB calibrated colorspace used by Java, and the CMYK and GrayScale spaces are device-dependent. Any RGB colors that are specified will use the RGB colorspace of the object, CMYK colors will use the CMYK colorspace, and grayscale colors will use the GrayScale colorspace.

Any one of these colorspaces can be replaced by setting the “colorspace” attribute. If the value of the colorspace is a 3 component colorspace, the elements RGB colorspace will be set. If the specified colorspace has 4 components, the elements CMYK space will be set, and so on. This means that only one non-standard colorspace can be used per element - or, put another way, no element can use both a calibrated CMYK and calibrated RGB colorspace (other than sRGB) at the same time. We doubt this will cause a problem for many.

Valid values are “sRGB”, “DeviceCMYK” and “DeviceGray”, which set the colorspaces to the default values, or the URL of an ICC color profile file. Here’s an example which calibrates the document to use the NTSC color profile. Note that although in theory any element can have a different colorspace set, in practice it keeps everything simpler if you set the colorspace on the BODY element and leave it. The one exception to this is images, which we’ll cover below.

```
<body colorspace="http://path/to/NTSCspace.icc">
  <h1 color="#FF0000">
    This heading is in bright-red, according to the NTSC color profile
  </h1>
</body>
```

PNG, TIFF and JPEG images may optionally have an ICC color profile embedded in the image - if one is found it will be used automatically. The ability to override the colorspace for an image has been removed in version 1.1.10 - if the image needs to use a particular colorspace, it should be embedded in the file.

Hyperlinks

Hyperlinks can be used within PDF documents to navigate around the document, to load web pages in whatever web browser is installed on the users system, and to allow a limited level of interaction between the document and it's environment.

The familiar <A> element from HTML is a part of the Report Generators XML syntax, but unlike HTML it's significance is limited to stylistic changes only. Instead, the "href" attribute, which signifies a hyperlink, may be added to *any* element in the document. For example, the following two lines are equivalent.

```
<a href="http://bfo.com">go to website</a>
<span href="http://bfo.com">go to website</span>
```

This opens up some possibilities not available in HTML - for example, a TABLE or PIEGRAPH element could be turned into a hyperlink, simply by adding an "href" attribute. Be warned that the PDF specification is quiet about what happens if two hyperlink areas overlap..

So what values can the "href" attribute take? This tables lists the possibilities:

Example	*Description*
`#elementid`	Jump to the specified element in the report. The "elementid" is the ID of the destination element.
`#elementid?zoom=fit`	Jump to the specified element in the report, and zoom the page so that just that element is visible
`http://domain.com`	Any URL may be specified to jump to an external document. This functionality requires a web browser to be installed, and the exact form of the URL depends on the capabilities of that browser
`pdf:playsound(soundurl)`	Play an audio sample from the specified URL. The PDF specification can *in theory* handle Sun .AU, Macintosh AIFF and AIFF-C and Windows RIFF (.WAV) files, although RIFF support seems to be slightly more capable in our tests. This requires sound support from the PDF viewer application, and may not work on all operating systems.
`pdf:show(form element)`	Show the specified form element if it is hidden. The *form element* must be the name of a form element. See the Forms section.
`pdf:hide(form element)`	Hide the specified form element if it is visible. The *form element* must be the name of a form element. See the Forms section.
`pdf:reset()`	Reset the documents form to it's default values. See the Forms section.
`pdf:submit(url [, method])`	Submit the contents of the documents form to a URL. The *method* is optional - it defaults to "POST" - but if specified must be one of the following values: `POST` Post the form using the standard HTTP POST method `FDF` Post the form in Adobes Form Description Format (FDF) `XML` Post the form as XML (requires Acrobat 5.0) `PDF` Post the entire document (requires Acrobat 5.0) See the Forms section.
`javascript:code`	Run a section of JavaScript code. See the Forms section.
`pdf:action`	Run a "named" action - PDF viewer dependent, see below

The facility to run “named” actions can be very useful, provided you know which PDF viewer application your target audience is running. For the vast majority who run Acrobat 4.0 or greater, the following named actions may be used - they loosely correspond to the equivalent actions which can be run from the drop down menus in Acrobat. These values are case-sensitive.

Action name	*Description*
`Open`	Open the “open file” dialog
`Close`	Close the current document
`Print`	Print the current document
`GeneralInfo`	Bring up the “general information” dialog
`FontsInfo`	Bring up the “fonts information” dialog
`SecurityInfo`	Bring up the “security information” dialog
`Quit`	Quit the PDF viewer
`NextPage`	Go to the next page
`PrevPage`	Go to the previous page
`FirstPage`	Go to the first page
`LastPage`	Go to the last page
`GoToPage`	Bring up the “Go to page” dialog

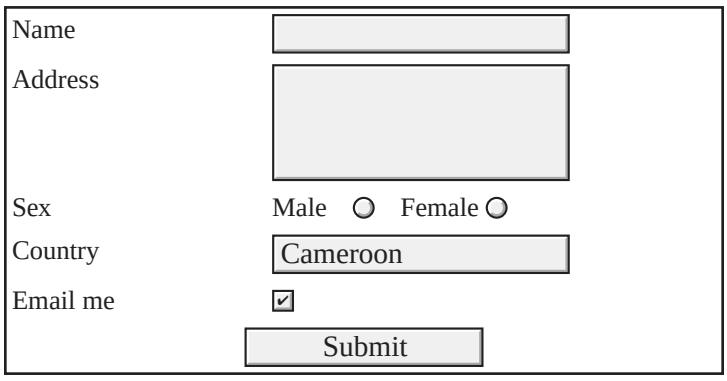
Action name	*Description*
`Find`	Bring up the “Find” dialog
`FindAgain`	Repeat the last search
`SelectAll`	Select the entire page
`Copy`	Copy the selection to the clipboard
`FullScreen`	Switch the document to fullscreen mode
`FitPage`	Zoom the document to fit the page
`ActualSize`	Zoom the document to actual size
`FitWidth`	Zoom the document to fit the width
`FitVisible`	Zoom the document to fit the entire page
`SinglePage`	Set the document to “Single page” mode
`OneColumn`	Set the document to “One column” mode
`TwoColumns`	Set the document to “Two columns” mode

Interactive Forms support

New in version 1.1 is the ability to include Form elements in the document. Interactive forms are one of the more underused aspects of PDF, but certainly one of the more interesting. There's a great deal more to forms than we cover here - we highly recommend purchasing a book on the subject, and experimenting with a copy of Acrobat to see what's possible. This section will document only the syntax used to add form elements to the Report Generator, not the reasons why you would. The use of interactive forms (specifically, the <INPUT> tag) requires the *Extended Edition* of the product.

Like HTML forms, a document can contain text boxes, drop down lists, radio buttons, check boxes, regular "submit" buttons and even JavaScript! The main differences are that the JavaScript object model is radically different, that each PDF only has a single form (unlike HTML), and that the form isn't tied to a single "submit" URL - instead, each submit button (there may be more than one) specifies the URL to submit to. Form elements are not covered by CSS2, so we've based our implementation fairly closely on HTML4.0, with a couple of simplifications.

```
<table><tr>
  <td>Name</td>
  <td><input type="text" name="name" width="10em"/></td>
 </tr><tr>
  <td>Address</td>
  <td><input type="text" name="address" lines="3" width="10em"/></td>
 </tr><tr>
  <td>Sex</td>
  <td>
   Male <input display="inline" type="radio" name="sex" value="male" padding-right="0.2in"/>
   Female <input display="inline" type="radio" name="sex" value="female"/>
  </td>
 </tr><tr>
  <td>Country</td>
  <td>
   <input type="select" name="country" value="Cameroon">
    <option>Cameroon</option>
    <option>Lebanon</option>
    <option>Other</option>
   </input>
  </td>
 </tr><tr>
  <td>Email me</td>
  <td><input type="checkbox" name="email" checked="true"/></td>
 </tr><tr>
  <td colspan="2" align="center">
   <input type="button" name="submit" onClick="pdf:submit(http://localhost, POST)"/>
  </td>
</tr></table>
```



Much of this should look fairly familiar to HTML authors. The key differences here are:

- The "name" and "type" values are mandatory, and each name must be unique across the entire document.

- As well as the “value” attribute, which can be set for every type of form field, the fields can take a “default-value” attribute, which controls what the field is reset to when a `pdf:reset()` action is run.
- Multiline text boxes don’t use the <textarea> tag, but are identical to normal text boxes - just set the “lines” attribute to the number of lines that are required. You can also optionally set the “scrollable” attribute to false, to prevent users from scrolling the box to enter more text. Initial values can be set either in the “value” attribute, or between the `<input>` and `</input>` tags.
- Drop down lists don’t use the <select> tag, but instead use a regular INPUT element with a “type” of “select”. It does use the OPTION elements to list the options, but the selected option is chosen by setting the “value” attribute on the INPUT. It’s not currently possible to select more than one option in a list. For multi-line lists just add a “lines” attribute, in the same way as the multiline text-boxes. Another variation is to set the “editable” attribute to “true”, which will turn the drop-down into a combo box - the user can type their own values into the box as well as choose one from the list.
- There’s no submit button - instead, a regular button with an “onClick” attribute does the same job. In fact, the “onClick” action may be any type of hyperlink that’s supported by the Report Generator. Buttons can also take a “src” attribute, which can be the URL of an image which will be pasted onto the button.
- Every form element recognises the boolean attributes “readonly”, which prevents the form from interacting with the user, “required” - which means the field must have a value in it befor the form is submitted - and “submitted”, which is on by default, but may be turned off to prevent the field from being submitted to the server, and is useful for fields used only for cosmetic or temporary purposes.

JavaScript

One of the aspects we haven’t demonstrated is JavaScript, which is supported in Acrobat 4.0 and 5.0, both the Reader and the full version of Acrobat. Although the syntax is identical to the JavaScript used in web browsers, the Document Object Model is radically different - do **not** expect your HTML JavaScript to work in Acrobat. Acrobats object model is documented in the `AcroJS.pdf` document supplied with retail versions of Acrobat. We won’t go into too much detail about the syntax, but will limit our discussion to showing you how to embed JavaScript code into your report.

Like HTML, we use the SCRIPT tag in the HEAD of the document to embed JavaScript. However, due to JavaScript code commonly containing the < and > characters, we recommend embedding it inside a “CData” block, like so:

```
<pdf>
 <head>
  <script>
  <![CDATA[
   function dumpForm()
   {
     var s="";
     s += "Your name is '"+this.getField("name").value+"'\n";
     s += "Your country is '"+this.getField("country").value+"'\n";
     s += "Your sex is '"+this.getField("sex").value+"'\n";
     app.alert(s);
   }
  ]]>
  </script>
 </head>
```

Then to call this function, simply create a link or button that runs the action **`javascript:dumpForm()`**. Click here to see what we mean.

Two actions we haven’t demonstrated yet are the “pdf:show” and “pdf:hide” actions. These can only be used with form fields, and although not terribly useful they’re interesting enough to demonstrate here. Roll your mouse over this link and keep an eye on the “address” box on the previous page.

This example is useful because it demonstrates an “event” handler. We’ve seen one example of these already - the “onClick” attribute on the “submit” button in the previous example. In fact, there are several to choose from, but although the “onMouseOver”,

“onMouseOut” and “onClick” handlers can be used with any element in the same way as the “href” attribute, the rest are limited to use with form fields.

Attribute	*Description*
`onClick`	The action to perform when the element of form field is clicked. Identical in function to “href”, the two attributes can be used interchangeably
`onMouseOver`	The action to perform when the mouse moves over the element or form field.
`onMouseOut`	The action to perform when the mouse moves out of the element or form field.
`onMouseDown`	The action to perform when the mouse button is clicked in the form fields focus area.
`onMouseUp`	The action to perform when the mouse button is released in the form fields focus area.
`onFocus`	The action to perform when the field gains focus (text elements only)
`onBlur`	The action to perform when the field loses focus (text elements only)
`onChange`	The action to perform when the value of the field has changed
`onKeyPress`	The action to perform when a key is pressed in the form field (text elements only). Use for limiting input into the field to digits (for example).
`onFormat`	The action to perform when the contents of the field is about to be redisplayed.
`onOtherChange`	The action to perform when the value of *one of the other fields* has changed. This is commonly used in conjunction with read-only fields, to show a value based on the contents of other fields.

Digital Signatures

Although still a type of form field, Digital Signatures are handled quite differently from the other fields - so we’ll cover them seperately. Digital signatures allow a PDF report to be signed before distribution. Like the other form fields, using this feature requires the *Extended Edition* of the product.

These are useful for two main purposed - one, to identify the author of the document, and two, to provide notice if the document has been altered after it was signed. This is done by calculating a checksum of the document, and then encrypting that checksum with the “private key” of the author, which can later be verified by a user with the full version of Adobe Acrobat or Acrobat Approval™, although *not* the free Acrobat Reader, by comparing it with the corresponding public key.

Digital Signatures are implemented in Acrobat via a plug-in or “handler”. As of 2018 the vast majority of signatures use the standard Adobe signature model and require the “handler” attribute to be set to “acrobat”. We also support “globalsign” as a value since 1.1.63, to use the GlobalSign signing service.

Currently signatures are limited in that only one signature may be applied to a document, otherwise an Exception is thrown.

Digital signatures are defined using an INPUT tag - the same tag used in HTML to define form elements (this is because digital signatures are part of the PDF version of a form, which we’ll be adding more support for in later releases). The tag can be placed anywhere in the body of the document, and doesn’t require a <FORM> like HTML. Here’s a quick look at a typical signature tag:

```
<input type="signature" name="sig1" keystore="mystore" password="secret" handler="acrobat"/>
```

and here’s an example showing how to use the “globalsign” handler for signing.

```
<input type="signature" name="sig1" keystore="mystore" password="secret" handler="globalsign"
    login="0123456789012345 0123456789..." identity="OU=Test Unit"/>
```

There are a number of attributes that apply only to digital signatures.

Attribute	*Description*
`type`	(Mandatory) The type of INPUT tag. Must be “signature” for digital signatures
`name`	(Mandatory) The name to give the form field.
`keystore`	(Mandatory) The URL of the keystore containing the private key to sign the document with.
`handler`	(Mandatory) The Digital Signature handler that will be used to verify the document. The value should be “acrobat” for most signatures, or “globalsign” for the Globalsign signing service. The values “acrobat6”, “verisign” and “selfsign” are all legacy synonyms for “acrobat”.
`password`	(Optional) The password required to open the keystore.
`alias`	(Optional) The alias or “friendly name” given to the private key in the keystore. Defaults to “mykey”.
`keypassword`	(Optional) The password to open the private key in the keystore. Defaults to the value of the `password` attribute.
`keystoretype`	(Optional) The type of keystore. Usually will be either “JKS” for “Java Keystore” or some other value like “pkcs12”, which depends on what JCE providers are available. May optionally include a hyphen, followed by a provider name - for example “JKS-SUN” to load the Sun implementation of the JKS keystore, or “pkcs12-BC” for the PKCS#12 implementation by The Legion of the Bouncy Castle. Defaults to “JKS”.
`signer`	(Optional) The name of the person or entity signing the document. For informational purposes only. Defaults to the name on the signing certificate.
`location`	(Optional) The location where the document was signed. For informational purposes only.
`reason`	(Optional) The reason why the document was signed. For informational purposes only.
`ocsp`	(Optional) If true, the certificates used for signing will be verified against their OCSP and CRL responders at the time of signing. This is required for “long-term validation” of signatures.
`timestampurl`	(Optional) May be set to the URL of a RFC3161 time stamp server to time-stamp the signature. This is required for “long-term validation” of signatures.
`background-image / background-pdf`	(Optional) An image to display as the content of the signature annotation. This should be used carefully - in particular, the image should ideally be transparent enough that it doesn’t completely mask out the area it covers. Alternatively a “background-pdf” may be used instead - it functions the same way.
`login`	(Required for Globalsign signatures). When handler is “globalsign”, this attribute must be set to the login details for the Globalsign service; either the *apikey* and *apisecret*, seperated by a space, or the path to the encrypted file containing this information which is supplied by Globalsign
`identity`	(Required for Globalsign signatures). When handler is “globalsign”, this attribute must be set to the identity details for the Globalsign service; either an X.500 principal, the path to an X.509 certificate to extract the identity from, or a JSON object describing the identity, the structure of which is described in the Globalsign documentation.

We’re only skimming the surface of this topic for now. A considerably more in-depth coverage of digital signatures (including how to generate test keys for signing and verify the signed document using Acrobat) is in the PDF library user guide.

PDF/A and PDF/UA Support

Since 1.1.47 the Report Generator can generate PDF/A documents - PDF/A-1, A-2 or A-3. These are the same as regular PDFs, with a few restrictions:

- All fonts must be embedded. This includes fonts in graphs, and fonts used as the markers in unordered lists (eg to draw the bullet)
- An Output Intent must be specified and an ICC profile embedded - either RGB or CMYK
- All colors must match this ICC profile - so if an sRGB profile is embedded, all colors in the PDF must be RGB. Transparency is not allowed.
- All images must match the embedded ICC profile, or the image must include an embedded ICC profile itself. Transparency is not allowed in PDF/A-1, but is in PDF/A-2 and A-3
- Form fields are problematic, and should be avoided in PDF/A-1.
- JavaScript is not allowed.

Converting a document that meets these requirements to PDF/A should be as simple as setting the `output-profile` meta-tag to "PDF/A3b", then setting the `output-intent-icc` meta-tag to the URI of the ICC profile to embed amd the `output-intent-identifier` meta-tag to the name of the profile. So, for example

```
<meta name="output-intent-identifier" value="sRGB"/>
<meta name="output-intent-icc" value="resources/sRGB.icc" />
<meta name="output-profile" value="PDF/A1b"/>
```

The value for "output-profile" can choose from the different *revisions* of the PDF/A specification (1 to 3 at the time of writing) and the different *conformance levels*, which are currently "a", "b" and "u". Examples are "PDF/A1b", "PDF/A2u", "PDF/A3a" etc. There are two example supplied with the Report Generator in the `samples` folder, one for CMYK and one for RGB, to get you started.

PDF/UA is a similar concept to PDF/A, but is focused on *Accessibility* rather than long-term archiving like PDF/A. Because of this it imposes restrictions on the structure of the document; for example, a <td> must be contained inside a <table/>, and all <img> elementa must have an "alt" attribute. Many of these restrictions are simply formalising the common structure of HTML, which is largely the same as the requirements for a the Report Generator. Where that's not the case, the Report Generator will throw an exception when it hits the invalid XML. With luck the text of the exception will tell you what you need to know, but please do email support@bfo.com if that's not the case.

Again, as for PDF/A there are examples in the "samples" folder which show how to create a simple PDF/UA document.

Finally, it is possible to create a document that meets the requirements of both PDF/A and PDF/UA - in fact, if you're already targetting PDF/UA it's is a good idea, as you've already done most of the work. Just add "+PDF/UA1" to the value of any other `output-profile` value. For example:

```
<meta name="output-intent-identifier" value="sRGB"/>
<meta name="output-intent-icc" value="resources/sRGB.icc" />
<meta name="output-profile" value="PDF/A3a+PDF/UA1"/>
```

Migrating from HTML

Migrating a document from HTML to XML may be easy or difficult, depending on how the HTML has been written. Following these four steps will account for 95% of the changes that are required.

1. The first, and probably most painful step is to ensure that all tags are closed and all attributes are quoted, so that the document meets the XML specification. Elements that don't officially require closure, like TD, LI, P, as well as tags with no content like BR and IMG are likely to be the cheif cause of problems. We find the insistance on quoting even unambiguous attributes annoying, and we're pleased to see that at least one XML parser (that supplied with Resin 2.0) has the option of being lax about this requirement.
2. Second, if any non-CSS legacy attributes are used (e.g. "bgcolor" to set the background color), these should be converted to their CSS equivalent (e.g. "background-color", and placed in a stylesheet in the head of the document (either embedded or external). Versions of the Report Generator since 1.0.11 recognise the `style="background-color:red"` method of defining attributes, although we still recommend the XML equivalent of `background-color="red"`.
3. Third, check the document for inline images, tables, lists or other blocks inside paragraphs. We've found this to be a common occurance, due to HTML not requiring a closing </P> tag.
4. Fourth and finally, change the tags that have a different syntax. These are:

    - TABLE - the HTML attributes "border" and "cellmargin" should be renamed "cellborder" and "cellmargin".
    - The legacy FONT element should be replaced with an equivalent SPAN
    - The various different styles of paragraph and span available in HTML - ADDRESS, CITE etc. should be replaced with a P or SPAN, setting the "class" attribute to control the style.
    - Definition lists using the DL, DT and DT elements aren't supported, and should be replaced wither either a normal UL list with the "value" attribute set to the definition, or a TABLE.

Provided that no JavaScript, forms or frames are used, these steps should result in a report that is legible and ready to be tailored for it's eventual destination as a PDF document.

Internationalization

The vast majority of the worlds languages can be used with the Report Generator. Thanks to XML's natural character set of UTF-8, as well as it's ability to use "preferred" native character sets like Shift-JIS or EUC-KR, specifying the actual characters to display is not a problem. When editing your XML just remember to save the file in the correct encoding - UTF-8 unless you've specified otherwise.

When it comes to actually displaying the characters, the key is to use the right font. The standard fonts (Helvetica, Times and Courier) will, as far as we know, display the following languages correctly:

English, French, German, Portuguese, Italian, Spanish, Dutch (no "ij" ligature), Danish, Swedish, Norwegian, Icelandic, Finnish, Polish, Croatian, Czech, Hungarian, Romanian, Slovak, Slovenian, Latvian, Lithuanian, Estonian, Turkish, Catalan (although the "L with dot" character is missing), Basque, Albanian, Rhaeto-Romance, Sorbian, Faroese, Irish, Scottish, Afrikaans, Swahili, Frisian, Galician, Indonesian/Malay and Tagalog.

For Chinese, Japanese and Korean the obvious choice is to use the standard east asian fonts like "hygothic", "heiseimin" and "mhei" (the full list is in the Fonts section).

For other languages like Czech, Slovenian, Russian or Hebrew that require characters not directly supported by the PDF specification, the best method is to embed an appropriate OpenType or Type 1 font using the LINK element. Provided the font contains the character, and the "embed" attributes is left at it's default values of "true", the characters should display correctly.

Right-to-left languages (arabic, hebrew, syriac and urdu) are supported. The "direction" attribute controls the overall flow of the text and defaults to "rtl" for these languages, and can also be set manually. The "unicode-bidi" CSS property is not supported, so for further control it is necessary to embed the correct byte-order marks in the text, eg `&#x202b;`.

Every element in the document can have a language set using the "lang" attribute, which defaults to the current locale of the PDF generation process. This attribute affects a few things - the style of quote substitution if the "requote" attribute is true, the type of currency format to use when a "currency()" formatter is used with graphs, default text direction, default font (if the language is Chinese, Japanese or Korean) and default page size - for `en_US`, `en_CA` and `fr_CA` the default is Letter, for everyone else it's A4.

Examples would be "de" for German and "en_GB" for British English. Generally it is enough to set the "lang" attribute of the `<pdf>` element, which sets the language for the entire document.

When creating documents from JSP pages, remember to set the character set to match the `<?xml?>` declaration. This also applies to pages included via the `<jsp:include>` method. The following examples are all valid:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<%@ page language="java" contentType="text/xml"%>
<!-- document follows in ISO-8859-1 -->
```

```
<?xml version="1.0"?>
<%@ page language="java" contentType="text/xml; charset=UTF-8"%>
<!-- document follows in UTF-8 -->
```

```
<?xml version="1.0" encoding="ShiftJIS"?>
<%@ page language="java" contentType="text/xml; charset=ShiftJIS"%>
<!-- document follows in Shift JIS -->
```

Word and line-breaking rules for east-asian text follow the rules outlined in css-text-3. The `line-break` and `word-break` CSS properties are available as of release 1.1.63 (although the ‘break-word’ value of word-break is not supported). If these properties are unset, for compatibility with previous releases we default to the behaviour prior to 1.1.62 However if either is specified, the other will default to “normal”. In addition, when the line-break property is set to “loose” or “normal”, the language of the content will also affect the algorithm: as defined in css-text-3, text in Japanese or Chinese is split differently. Some examples:

```
// legacy line break behaviour
<body> ... </body>...

// line-break and word-break both set to "normal"
<body style="word-break: normal"> ... </body>...

// line-break set to loose, word-break set to "normal"
<body style="line-break: loose"> ... </body>...

// line-break set to loose, word-break set to "normal", and the special rules
// for breaking Japanese characters defined in css-text-3 will be applied
<body lang="ja" style="line-break: loose"> ... </body>...

// the special value of "-bfo-legacy" can be used to reset the behaviour to 1.1.62
// and earlier
<body style="line-break: -bfo-legacy; word-break: -bfo-legacy"> ... </body>...
```

Problems, Limitations and Future Direction

Several changes are planned for the next release, listed here in no particular order. We'd greatly appreciate additions or feedback on this list.

- The collapsing border model for tables needs to be implemented
- Running headers and footers are planned
- We plan to support "subpages" - dividing a page up into smaller sections like columns or quarters.

Following is a list of known problems and limitations with the generator.

Limit on nesting the "overflow" attribute

Due to the way the "overflow" attribute is implemented and to limitations in the PostScript language, elements with the "overflow" attribute set to "hidden" shouldn't be nested more than 12 deep for compatibility with Acrobat 4.0, or 28 deep for Acrobat 5.0 or later

Differences with CSS2 specification

For those that take specification compliance seriously, here is a list of the differences between the CSS2 specification and our implementation of it. Numbers refer to the section of the spec at http://http://www.w3.org/TR/REC-CSS2. Over time the length of this list will shorten as we head towards full CSS2 compliance.

```
4.1.5  At rules (like @import or @charset) are not supported
4.2    Parsing errors throw an error, and are not ignored.
5.11   Only the :lang and :first-child (and the custom :last-child) pseudo selectors are used.
7.     Media types are not used (we've only got the one media - PDF)
8.5.3  Border style - double, groove, ridge, inset and outset are not used
9.2.5  The "display" property defaults to "block", not "inline".
9.4    "bottom" and "right" aren't used for positioning, only "top" and "left"
9.9.1  "z-index" isn't supported. Later elements will always be drawn over earlier ones.
9.10   Changing text-direction in the middle of a word won't work. Ensure there is always
       a space between phrases with different directions.
10.4   max-width is ignored.
10.7   max-height is ignored
11.1.1 The clip rectangle value is always "auto"
12.    The generated content section is pretty different. We don't use this
       method for inserting quotes around text or numbers before lists. The way we do
       it is shown in the "lists.xml" example.
13.1   We don't use the @page rule.
13.2.3 We don't do the crop marks.
13.3   The "page-break" attributes are only recognised on elements that are the direct
       child of the BODY element (the exception to this is the TR tag, which recognises
it when these attributes is set to "join"). All elements other than P, PRE, H1-H4,
BLOCKQUOTE and TABLE have the "page-break-inside" attribute set to "avoid".
13.3.2 Named pages are not used.
15.2.4 font-size-adjust is ignored.
14.2.1 background-attachment doesn't apply to PDF
15.3   We use a much simpler method for selecting fonts - you either embed it or you don't!
16.3   overline and blink are not valid text-decorations
16.4   text-shadow is ignored
16.5   word-spacing isn't used (see justification-ratio though)
17.6   The collapsing border model is not supported, and borders are always drawn around empty cells.
18.    User interface doesn't apply to PDF
19.    Aural style sheets doesn't apply to PDF
```

Reference Section

Named Colors

The following named colors can be used in the Report Generator. Their equivalent RGB values are listed below.

white #FFFFFF	whitesmoke #F5F5F5	ghostwhite #F8F8FF	snow #FFFAFA	gainsboro #DCDCDC
lavender #E6E6FA	aliceblue #F0F8FF	lavenderblush #FFF0F5	seashell #FFF5EE	lightcyan #E0FFFF
azure #F0FFFF	floralwhite #FFFAF0	oldlace #FDF5E6	mintcream #F5FFFA	honeydew #F0FFF0
linen #FAF0E6	ivory #FFF0F0	lightyellow #FFFFE0	beige #F5F5DC	antiquewhite #FAEBD7
cornsilk #FFF8DC	lemonchiffon #FFFACD	lightgoldenrodyellow #FAFAD2	mistyrose #FFE4E1	bisque #FFE4C4
papayawhip #FFEFD5	blanchedalmond #FFFFCD	peachpuff #FFEFD5	palegoldenrod #EEE8AA	wheat #F5DEB3
moccasin #FFE4B5	navajowhite #FFDEAD	khaki #F0E68C	yellow #FFFF00	gold #FFD700
goldenrod #DAA520	darkkhaki #BDB76B	tan #D2B48C	peru # CD853F	darkgoldenrod #B8860B
rosybrown #BC8F8F	burlywood #DEB887	sandybrown #F4A460	saddlebrown #8B4513	lightsalmon #FFA07A
salmon #FA8072	coral #FF7F50	peachpuff #FFEFD5	darksalmon #E9967A	lightcoral #F08080
darkorange #FF8C00	orange #FFA500	mistyrose #FFE4E1	lightpink #FFB6C1	pink #FFC0CB
hotpink #FF69B4	palevioletred #DB7093	mediumvioletred #C71585	thistle #D8BFD8	plum #DDA0DD
mediumorchid #BA55D3	mediumpurple #9370DB	darkmagenta #8B008B	navy #000080	mediumslateblue #7B68EE
cornflowerblue #6495ED	mediumblue #0000CD	deepskyblue #00BFFF	lightskyblue #87CEFA	powderblue #B0E0E6
skyblue #87CEEB	lightblue #ADD8E6	darkcyan #008B8B	cadetblue #5F9EA0	steelblue #4682B4
lightsteelblue #B0C4DE	teal #008080	lightseagreen # 20B2AA	mediumaquamarine #66CDAA	darkseagreen #8FBC8F
darkturquoise # 00CED1	mediumturquoise #48D1CC	turquoise #40E0D0	paleturquoise #AFEEEE	aquamarine #7FFFD4
aqua # 00FFFF	cyan #00FFFF	lightcyan #E0FFFF	mediumspringgreen #00FA9A	springgreen #00FF7F
greenyellow #ADFF2F	lawngreen #7CFC00	lime #00FF00	limegreen #32CD32	chartreuse #7FFF00
lightgreen #90EE90	palegreen #98FB98	yellowgreen #9ACD32	lightgrey #D3D3D3	gainsboro #DCDCDC
silver #C0C0C0	darkgray #A9A9A9	gray #808080	black #000000	indianred #CD5C5C
tomato #FD6347	orangered #FF4500	red #FF0000	maroon #800000	darkred #8B0000
crimson #DC143C	deeppink #FF1493	brown #A52A2A	chocolate #D2691E	sienna #A0522D
firebrick #B22222	magenta #FF00FF	fuchsia #FF00FF	violet #EE82EE	orchid #DA70D6
indigo #4B0082	slateblue #6A5ACD	darkslateblue #483D8B	midnightblue #191970	darkblue #00008B
blue #0000FF	royalblue #4169E1	dodgerblue #1E90FF	mediumseagreen #3CB371	seagreen #2E8B57
green #008000	darkgreen #006400	olive #808000	darkolivegreen #556B2F	olivedrab #6B8E23
forestgreen #228B22	darkslategray #2F4F4F	dimgray #696969	slategray #708090	lightslategray #778899

Named Entities

The following named entities are understood by the Report Generator.

Name	*Symbol*	*CodePoint*	*Description*
zwnbsp		U+feff	zero width non breaking space. Best use NOBR instead.
nbsp		U+00a0	no-break space = non-breaking space ISOnum
iexcl	¡	U+00a1	inverted exclamation mark ISOnum
cent	¢	U+00a2	cent sign ISOnum
pound	£	U+00a3	pound sign ISOnum
curren	¤	U+00a4	currency sign ISOnum
yen	¥	U+00a5	yen sign = yuan sign ISOnum
brvbar	¦	U+00a6	broken bar = broken vertical bar ISOnum
sect	§	U+00a7	section sign ISOnum
uml	¨	U+00a8	diaeresis = spacing diaeresis ISOdia
copy	©	U+00a9	copyright sign ISOnum
ordf	ª	U+00aa	feminine ordinal indicator ISOnum
laquo	«	U+00ab	left-pointing double angle quotation mark = left pointing guillemet ISOnum
not	¬	U+00ac	not sign = discretionary hyphen ISOnum
shy	-	U+00ad	soft hyphen = discretionary hyphen ISOnum. This character is used in the Unicode (as opposed to the HTML sense), which means it's only displayed if a word break occurs at the specified position.
reg	®	U+00ae	registered sign = registered trade mark sign ISOnum
macr	¯	U+00af	macron = spacing macron = overline = APL overbar ISOdia
deg	°	U+00b0	degree sign ISOnum
plusmn	±	U+00b1	plus-minus sign = plus-or-minus sign ISOnum
sup2	²	U+00b2	superscript two = superscript digit two = squared ISOnum
sup3	³	U+00b3	superscript three = superscript digit three = cubed ISOnum
acute	´	U+00b4	acute accent = spacing acute ISOdia
micro	µ	U+00b5	micro sign ISOnum
para	¶	U+00b6	pilcrow sign = paragraph sign ISOnum
middot	·	U+00b7	middle dot = Georgian comma = Greek middle dot ISOnum
cedil	¸	U+00b8	cedilla = spacing cedilla ISOdia
sup1	¹	U+00b9	superscript one = superscript digit one ISOnum
ordm	º	U+00ba	masculine ordinal indicator ISOnum
raquo	»	U+00bb	right-pointing double angle quotation mark = right pointing guillemet ISOnum
frac14	¼	U+00bc	vulgar fraction one quarter = fraction one quarter ISOnum

Name	*Symbol*	*CodePoint*	*Description*
`frac12`	½	U+00bd	vulgar fraction one half = fraction one half ISOnum
`frac34`	¾	U+00be	vulgar fraction three quarters = fraction three quarters ISOnum
`iquest`	¿	U+00bf	inverted question mark = turned question mark ISOnum
`Agrave`	À	U+00c0	latin capital letter A with grave = latin capital letter A grave ISOlat1
`Aacute`	Á	U+00c1	latin capital letter A with acute ISOlat1
`Acirc`	Â	U+00c2	latin capital letter A with circumflex ISOlat1
`Atilde`	Ã	U+00c3	latin capital letter A with tilde ISOlat1
`Auml`	Ä	U+00c4	latin capital letter A with diaeresis ISOlat1
`Aring`	Å	U+00c5	latin capital letter A with ring above = latin capital letter A ring ISOlat1
`AElig`	Æ	U+00c6	latin capital letter AE = latin capital ligature AE ISOlat1
`Ccedil`	Ç	U+00c7	latin capital letter C with cedilla ISOlat1
`Egrave`	È	U+00c8	latin capital letter E with grave ISOlat1
`Eacute`	É	U+00c9	latin capital letter E with acute ISOlat1
`Ecirc`	Ê	U+00ca	latin capital letter E with circumflex ISOlat1
`Euml`	Ë	U+00cb	latin capital letter E with diaeresis ISOlat1
`Igrave`	Ì	U+00cc	latin capital letter I with grave ISOlat1
`Iacute`	Í	U+00cd	latin capital letter I with acute ISOlat1
`Icirc`	Î	U+00ce	latin capital letter I with circumflex ISOlat1
`Iuml`	Ï	U+00cf	latin capital letter I with diaeresis ISOlat1
`ETH`	Ð	U+00d0	latin capital letter ETH ISOlat1
`Ntilde`	Ñ	U+00d1	latin capital letter N with tilde ISOlat1
`Ograve`	Ò	U+00d2	latin capital letter O with grave ISOlat1
`Oacute`	Ó	U+00d3	latin capital letter O with acute ISOlat1
`Ocirc`	Ô	U+00d4	latin capital letter O with circumflex ISOlat1
`Otilde`	Õ	U+00d5	latin capital letter O with tilde ISOlat1
`Ouml`	Ö	U+00d6	latin capital letter O with diaeresis ISOlat1
`times`	×	U+00d7	multiplication sign ISOnum
`Oslash`	Ø	U+00d8	latin capital letter O with stroke = latin capital letter O slash ISOlat1
`Ugrave`	Ù	U+00d9	latin capital letter U with grave ISOlat1
`Uacute`	Ú	U+00da	latin capital letter U with acute ISOlat1
`Ucirc`	Û	U+00db	latin capital letter U with circumflex ISOlat1
`Uuml`	Ü	U+00dc	latin capital letter U with diaeresis ISOlat1
`Yacute`	Ý	U+00dd	latin capital letter Y with acute ISOlat1
`THORN`	Þ	U+00de	latin capital letter THORN ISOlat1

Name	*Symbol*	*CodePoint*	*Description*
szlig	ß	U+00df	latin small letter sharp s = ess-zed ISOlat1
agrave	à	U+00e0	latin small letter a with grave = latin small letter a grave ISOlat1
aacute	á	U+00e1	latin small letter a with acute ISOlat1
acirc	â	U+00e2	latin small letter a with circumflex ISOlat1
atilde	ã	U+00e3	latin small letter a with tilde ISOlat1
auml	ä	U+00e4	latin small letter a with diaeresis ISOlat1
aring	å	U+00e5	latin small letter a with ring above = latin small letter a ring ISOlat1
aelig	æ	U+00e6	latin small letter ae = latin small ligature ae ISOlat1
ccedil	ç	U+00e7	latin small letter c with cedilla ISOlat1
egrave	è	U+00e8	latin small letter e with grave ISOlat1
eacute	é	U+00e9	latin small letter e with acute ISOlat1
ecirc	ê	U+00ea	latin small letter e with circumflex ISOlat1
euml	ë	U+00eb	latin small letter e with diaeresis ISOlat1
igrave	ì	U+00ec	latin small letter i with grave ISOlat1
iacute	í	U+00ed	latin small letter i with acute ISOlat1
icirc	î	U+00ee	latin small letter i with circumflex ISOlat1
iuml	ï	U+00ef	latin small letter i with diaeresis ISOlat1
eth	ð	U+00f0	latin small letter eth ISOlat1
ntilde	ñ	U+00f1	latin small letter n with tilde ISOlat1
ograve	ò	U+00f2	latin small letter o with grave ISOlat1
oacute	ó	U+00f3	latin small letter o with acute ISOlat1
ocirc	ô	U+00f4	latin small letter o with circumflex ISOlat1
otilde	õ	U+00f5	latin small letter o with tilde ISOlat1
ouml	ö	U+00f6	latin small letter o with diaeresis ISOlat1
divide	÷	U+00f7	division sign ISOnum
oslash	ø	U+00f8	latin small letter o with stroke, = latin small letter o slash ISOlat1
ugrave	ù	U+00f9	latin small letter u with grave ISOlat1
uacute	ú	U+00fa	latin small letter u with acute ISOlat1
ucirc	û	U+00fb	latin small letter u with circumflex ISOlat1
uuml	ü	U+00fc	latin small letter u with diaeresis ISOlat1
yacute	ý	U+00fd	latin small letter y with acute ISOlat1
thorn	þ	U+00fe	latin small letter thorn with ISOlat1
yuml	ÿ	U+00ff	latin small letter y with diaeresis ISOlat1
OElig	Œ	U+0152	latin capital ligature OE ISOlat2

Name	*Symbol*	*CodePoint*	*Description*
`oelig`	œ	U+0153	latin small ligature oe ISOlat2
`Scaron`	Š	U+0160	latin capital letter S with caron ISOlat2
`scaron`	š	U+0161	latin small letter s with caron ISOlat2
`Yuml`	Ÿ	U+0178	latin capital letter Y with diaeresis ISOlat2
`circ`	ˆ	U+02c6	modifier letter circumflex accent ISOpub
`tilde`	˜	U+02dc	small tilde ISOdia
`zwnj`		U+200c	zero width non-joiner NEW RFC 2070
`zwj`		U+200d	zero width joiner NEW RFC 2070
`lrm`		U+200e	left-to-right mark NEW RFC 2070
`rlm`		U+200f	right-to-left mark NEW RFC 2070
`ndash`	–	U+2013	en dash ISOpub
`mdash`	—	U+2014	em dash ISOpub
`lsquo`	‘	U+2018	left single quotation mark ISOnum
`rsquo`	’	U+2019	right single quotation mark ISOnum
`sbquo`	,	U+201a	single low-9 quotation mark NEW
`ldquo`	“	U+201c	left double quotation mark ISOnum
`rdquo`	”	U+201d	right double quotation mark ISOnum
`bdquo`	„	U+201e	double low-9 quotation mark NEW
`dagger`	†	U+2020	dagger ISOpub
`Dagger`	‡	U+2021	double dagger ISOpub
`permil`	‰	U+2030	per mille sign ISOtech
`lsaquo`	‹	U+2039	single left-pointing angle quotation mark ISO proposed
`rsaquo`	›	U+203a	single right-pointing angle quotation mark ISO proposed
`euro`	€	U+20ac	euro sign NEW
`trade`	™	U+2122	trade mark sign ISOnum
`fnof`	*ƒ*	U+0192	latin small f with hook = function = florin ISOtech
`Alpha`	A	U+0391	greek capital letter alpha
`Beta`	B	U+0392	greek capital letter beta
`Gamma`	Γ	U+0393	greek capital letter gamma ISOgrk3
`Delta`	Δ	U+0394	greek capital letter delta ISOgrk3
`Epsilon`	E	U+0395	greek capital letter epsilon
`Zeta`	Z	U+0396	greek capital letter zeta
`Eta`	H	U+0397	greek capital letter eta
`Theta`	Θ	U+0398	greek capital letter theta ISOgrk3

Name	*Symbol*	*CodePoint*	*Description*
Iota	Ι	U+0399	greek capital letter iota
Kappa	Κ	U+039a	greek capital letter kappa
Lambda	Λ	U+039b	greek capital letter lambda ISOgrk3
Mu	Μ	U+039c	greek capital letter mu
Nu	Ν	U+039d	greek capital letter nu
Xi	Ξ	U+039e	greek capital letter xi ISOgrk3
Omicron	Ο	U+039f	greek capital letter omicron
Pi	Π	U+03a0	greek capital letter pi ISOgrk3
Rho	Ρ	U+03a1	greek capital letter rho
Sigma	Σ	U+03a3	greek capital letter sigma ISOgrk3
Tau	Τ	U+03a4	greek capital letter tau
Upsilon	Υ	U+03a5	greek capital letter upsilon ISOgrk3
Phi	Φ	U+03a6	greek capital letter phi ISOgrk3
Chi	Χ	U+03a7	greek capital letter chi
Psi	Ψ	U+03a8	greek capital letter psi ISOgrk3
Omega	Ω	U+03a9	greek capital letter omega ISOgrk3
alpha	α	U+03b1	greek small letter alpha ISOgrk3
beta	β	U+03b2	greek small letter beta ISOgrk3
gamma	γ	U+03b3	greek small letter gamma ISOgrk3
delta	δ	U+03b4	greek small letter delta ISOgrk3
epsilon	ε	U+03b5	greek small letter epsilon ISOgrk3
zeta	ζ	U+03b6	greek small letter zeta ISOgrk3
eta	η	U+03b7	greek small letter eta ISOgrk3
theta	θ	U+03b8	greek small letter theta ISOgrk3
iota	ι	U+03b9	greek small letter iota ISOgrk3
kappa	κ	U+03ba	greek small letter kappa ISOgrk3
lambda	λ	U+03bb	greek small letter lambda ISOgrk3
mu	μ	U+03bc	greek small letter mu ISOgrk3
nu	ν	U+03bd	greek small letter nu ISOgrk3
xi	ξ	U+03be	greek small letter xi ISOgrk3
omicron	ο	U+03bf	greek small letter omicron NEW
pi	π	U+03c0	greek small letter pi ISOgrk3
rho	ρ	U+03c1	greek small letter rho ISOgrk3
sigmaf	ς	U+03c2	greek small letter final sigma ISOgrk3

Name	*Symbol*	*CodePoint*	*Description*
sigma	σ	U+03c3	greek small letter sigma ISOgrk3
tau	τ	U+03c4	greek small letter tau ISOgrk3
upsilon	υ	U+03c5	greek small letter upsilon ISOgrk3
phi	ϕ	U+03c6	greek small letter phi ISOgrk3
chi	χ	U+03c7	greek small letter chi ISOgrk3
psi	ψ	U+03c8	greek small letter psi ISOgrk3
omega	ω	U+03c9	greek small letter omega ISOgrk3
thetasym	ϑ	U+03d1	greek small letter theta symbol NEW
upsih	ϒ	U+03d2	greek upsilon with hook symbol NEW
piv	ϖ	U+03d6	greek pi symbol ISOgrk3
bull	•	U+2022	bullet = black small circle ISOpub
hellip	…	U+2026	horizontal ellipsis = three dot leader ISOpub
prime	′	U+2032	prime = minutes = feet ISOtech
Prime	″	U+2033	double prime = seconds = inches ISOtech
oline	‾	U+203e	overline = spacing overscore NEW
frasl	⁄	U+2044	fraction slash NEW
weierp	℘	U+2118	script capital P = power set = Weierstrass p ISOamso
image	ℑ	U+2111	blackletter capital I = imaginary part ISOamso
real	ℜ	U+211c	blackletter capital R = real part symbol ISOamso
alefsym	ℵ	U+2135	alef symbol = first transfinite cardinal NEW
larr	←	U+2190	leftwards arrow ISOnum
uarr	↑	U+2191	upwards arrow ISOnum-->
rarr	→	U+2192	rightwards arrow ISOnum
darr	↓	U+2193	downwards arrow ISOnum
harr	↔	U+2194	left right arrow ISOamsa
crarr	↵	U+21b5	downwards arrow with corner leftwards = carriage return NEW
lArr	⇐	U+21d0	leftwards double arrow ISOtech
uArr	⇑	U+21d1	upwards double arrow ISOamsa
rArr	⇒	U+21d2	rightwards double arrow ISOtech
dArr	⇓	U+21d3	downwards double arrow ISOamsa
hArr	⇔	U+21d4	left right double arrow ISOamsa
forall	∀	U+2200	for all ISOtech
part	∂	U+2202	partial differential ISOtech
exist	∃	U+2203	there exists ISOtech

Name	*Symbol*	*CodePoint*	*Description*
`empty`	∅	U+2205	empty set = null set = diameter ISOamso
`nabla`	∇	U+2207	nabla = backward difference ISOtech
`isin`	∈	U+2208	element of ISOtech
`notin`	∉	U+2209	not an element of ISOtech
`ni`	∋	U+220b	contains as member ISOtech
`prod`	∏	U+220f	n-ary product = product sign ISOamsb
`sum`	∑	U+2211	n-ary sumation ISOamsb
`minus`	−	U+2212	minus sign ISOtech
`lowast`	∗	U+2217	asterisk operator ISOtech
`radic`	√	U+221a	square root = radical sign ISOtech
`prop`	∝	U+221d	proportional to ISOtech
`infin`	∞	U+221e	infinity ISOtech
`ang`	∠	U+2220	angle ISOamso
`and`	∧	U+2227	logical and = wedge ISOtech
`or`	∨	U+2228	logical or = vee ISOtech
`cap`	∩	U+2229	intersection = cap ISOtech
`cup`	∪	U+222a	union = cup ISOtech
`int`	∫	U+222b	integral ISOtech
`there4`	∴	U+2234	therefore ISOtech
`sim`	~	U+223c	tilde operator = varies with = similar to ISOtech
`cong`	≅	U+2245	approximately equal to ISOtech
`asymp`	≈	U+2248	almost equal to = asymptotic to ISOamsr
`ne`	≠	U+2260	not equal to ISOtech
`equiv`	≡	U+2261	identical to ISOtech
`le`	≤	U+2264	less-than or equal to ISOtech
`ge`	≥	U+2265	greater-than or equal to ISOtech
`sub`	⊂	U+2282	subset of ISOtech
`sup`	⊃	U+2283	superset of ISOtech
`nsub`	⊄	U+2284	not a subset of ISOamsn
`sube`	⊆	U+2286	subset of or equal to ISOtech
`supe`	⊇	U+2287	superset of or equal to ISOtech
`oplus`	⊕	U+2295	circled plus = direct sum ISOamsb
`otimes`	⊗	U+2297	circled times = vector product ISOamsb
`perp`	⊥	U+22a5	up tack = orthogonal to = perpendicular ISOtech

Name	*Symbol*	*CodePoint*	*Description*
sdot	⋅	U+22c5	dot operator ISOamsb
lceil	⌈	U+2308	left ceiling = apl upstile ISOamsc
rceil	⌉	U+2309	right ceiling ISOamsc
lfloor	⌊	U+230a	left floor = apl downstile ISOamsc
rfloor	⌋	U+230b	right floor ISOamsc
lang	〈	U+2329	left-pointing angle bracket = bra ISOtech
rang	〉	U+232a	right-pointing angle bracket = ket ISOtech
loz	◊	U+25ca	lozenge ISOpub
spades	♠	U+2660	black spade suit ISOpub
clubs	♣	U+2663	black club suit = shamrock ISOpub
hearts	♥	U+2665	black heart suit = valentine ISOpub
diams	♦	U+2666	black diamond suit ISOpub

Element and Attribute reference

In a future version of the documentation this section will contain a cross-referenced list of all the elements and attributes that can be used in the Report Generator.

For now, the Element and Attribute reference information is online at http://bfo.com/ products/report/docs/tags.