The JFreeReport Class Library

Version 0.8.3

Reference Guide

Written by David Gilbert

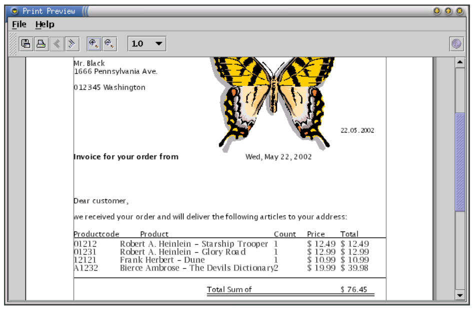
July 9, 2003

Contents

1 Introduction

1.1 What is JFreeReport?

JFreeReport is a free Java^{TM} class library for generating reports. JFreeReport data is sourced via Swing's `TableModel` interface,[1] and formatted according to an XML-based report definition file. Reports can be previewed on screen, sent to the printer or saved in various formats including PDF, HTML, Excel, CSV and plain text format.



Mr. Black
1666 Pennsylvania Ave.

012345 Washington

22.05.2002

Invoice for your order from Wed, May 22, 2002

Dear customer,

we received your order and will deliver the following articles to your address:

Productcode	Product	Count	Price	Total
01212	Robert A. Heinlein – Starship Trooper	1	$ 12.49	$ 12.49
01231	Robert A. Heinlein – Glory Road	1	$ 12.99	$ 12.99
12121	Frank Herbert – Dune	1	$ 10.99	$ 10.99
A1232	Bierce Ambrose – The Devils Dictionary	2	$ 19.99	$ 39.98

Total Sum of $ 76.45

JFreeReport is free under the terms of the GNU Lesser General Public Licence (LGPL)—see Appendix B for details.

1.2 The JFreeReport Project

The official home page of JFreeReport is:

`http://www.jfree.org/jfreereport/index.html`

Please visit this site for news, updates, and access to the JFreeReport forum.

Thomas Morgner is the JFreeReport Project Leader. Development activities are coordinated through the project site hosted at SourceForge:

`http://sourceforge.net/projects/jfreereport/`

New developers are always welcome! If you would like to help out, please visit the project site and sign up.

1.3 This Document

This document has been written (by David Gilbert) for version 0.8.3 of JFreeReport (actually, a lot of the content relates to versions 0.8.0 and 0.8.1—it hasn't been updated for the latest releases).

> *Note: this is the version of the guide that was previously available for purchase from Object Refinery Limited. It is currently available as a free download—this may change for future releases.*

[1] Used to obtain data for tables created with Swing's `JTable` class.

1.4 Disclaimer

Please note that I cannot guarantee that this document is error free. You must use this document *at your own risk* or *not use it at all.*

1.5 Acknowledgements

I would like to thank Thomas Morgner for his (major) contribution as the JFreeReport Project Leader.

JFreeReport has also been enhanced by the contributions of several other developers: Piotr Bzdyl, Heiko Evermann, Patrice Rolland and Joerg Schoemer. The work by these contributors is very much appreciated.

Also, I would like to thank Bruno Lowagie and Paulo Soares for developing iText, the class library used in JFreeReport to save reports in PDF format.

Finally, thanks to all the developers that have provided feedback (bug reports and feature requests) concerning JFreeReport.

1.6 Comments and Suggestions

If you have any comments or suggestions regarding this document, please send an e-mail to: `david.gilbert@object-refinery.com`

2 Downloading and Installing JFreeReport

2.1 Introduction

This section contains instructions for:

- downloading and installing (unpacking) JFreeReport;
- running the demonstration applications;
- compiling[2] the JFreeReport source code;
- generating the Javadoc HTML documentation from the source code;

2.2 Download

You can download the latest version of JFreeReport from the JFreeReport project page on SourceForge:

`http://sourceforge.net/projects/jfreereport`

For version 0.8.3, you can choose between two download archives:

File:	**Description:**
`jfreereport-0.8.3.zip`	A `zip` file containing all the files required for JFreeReport (recommended for users of Microsoft Windows).
`jfreereport-0.8.3.tar.gz`	A `tar.gz` file containing all the files required for JFreeReport (recommended for users of Linux or Unix operating systems).

Both archives contain exactly the same source code, except that the end-of-line markers have been encoded using `CR/LF` in the `zip` archive, and `LF` only in the `tar.gz` archive.

Whichever archive you choose to download, you should save it in the directory where you want the `jfreereport-0.8.3` directory (created during unpacking—see the next section) to reside.

2.3 Installation

2.3.1 Overview

Installing JFreeReport is simply a matter of unpacking the download archive. When you unpack the archive, a new directory (`jfreereport-0.8.3`) will be created at the same location in your filesystem as the archive file itself.

[2]Recompilation is optional, because the precompiled runtime jar files are included in the download.

2.3.2 Unpacking the ZIP Archive

If you chose to download the `jfreereport-0.8.3.zip` archive, you can "unzip" it using any ZIP utility. The `jar` utility included with Java will do the job—type the following command:

```
jar -xvf jfreereport-0.8.3.zip
```

This will extract all the files into a new directory `jfreereport-0.8.3`. After this is complete, you can delete the `jfreereport-0.8.3.zip` file (if you want to).

2.3.3 Unpacking the `tar.gz` Archive

If you chose to download the `jfreereport-0.8.3.tar.gz` archive, you can unpack it using the following command:

```
tar xvzf jfreereport-0.8.3.tar.gz
```

This will extract all the files into a new directory `jfreereport-0.8.3`. After this is complete, you can delete the `jfreereport-0.8.3.tar.gz` file (if you want to).

2.3.4 The Files

After unpacking the JFreeReport archive, you will see the following files and directories:

File:	**Description:**
`ChangeLog`	A text file containing notes on changes in each version of JFreeReport.
`README`	A text file containing important information about JFreeReport. Read this first!!!
`ant/`	A directory containing an Ant build script for JFreeReport.
`checkstyle/`	A directory containing a Checkstyle property file.
`jfreereport-0.8.3-demo.jar`	The runtime jar file for the JFreeReport demonstration applications.
`jfreereport-0.8.3.jar`	The runtime jar file for the JFreeReport class library.
`lib/`	A directory containing runtime jar files for the libraries that JFreeReport depends on.
`license-LGPL.txt`	The full text of the GNU Lesser General Public License.
`resource/`	JFreeReport sample reports, plus reference tables for the extended parser.
`source/`	A directory containing the JFreeReport source code.

You should take a little time to familiarise yourself with the contents of the distribution and, in particular, read the `README` file.

2.4 Running the Demonstration Application

Several demonstration applications are included in the JFreeReport distribution. These are intended to illustrate some of the features supported by JFreeReport.

To run the main demo, switch to the `jfreereport-0.8.3` directory and type the following command:

```
java -jar jfreereport-0.8.3-demo.jar
```

This automatically sets up the appropriate classpath, and invokes the main demo class (`com.jrefinery.report.demo.JFreeReportDemo`). An alternative way to run the same application is to specify the classpath manually using the following command (all on one line):

```
java -classpath jfreereport-0.8.3-demo.jar:jfreereport-0.8.3.jar
:lib/jcommon-0.8.2.jar:lib/itext-0.98.jar:lib/gnujaxp.jar
:lib/bsh-1.2b6.jar:lib/pixie.jar:lib/jakarta-poi-1.5.1-final-20020615.jar
com.jrefinery.report.demo.JFreeReportDemo
```

A special point to note: if you are using Windows, you should use a semi-colon rather than a colon as the path separator, and a backward slash rather than a forward slash within paths.

2.5 Compiling the Source

An Ant build script is included in the JFreeReport distribution, in the `ant` directory. This script has been tested using Ant version 1.5.1. For more information about Ant, refer to:

```
http://jakarta.apache.org/ant
```

To recompile the JFreeReport source code, you can use the following command:

```
ant compile
```

This creates a temporary `build` directory within the `jfreereport-0.8.3` directory, compiles the JFreeReport classes, then creates a new runtime jar file.

Similarly, to recompile the JFreeReport demonstration applications, you can use the following command:

```
ant compile-demo
```

This creates a temporary `build` directory within the `jfreereport-0.8.3` directory, compiles the JFreeReport demo classes, then creates a new runtime jar file.

2.6 Generating the Javadoc Documentation

The JFreeReport source code contains *Javadoc comments*. You can use the `javadoc` tool to generate HTML documentation files directly from the source code.[3]

To generate the documentation, you can use the Ant build script:

```
ant javadoc
```

This will create a new `javadoc` directory (located within the `jfreereport-0.8.3` directory) containing the Javadoc reference information for JFreeReport. To view the Javadocs, open the `javadoc/index.html` file in your favourite web browser.

[3]The Javadoc HTML files can be viewed on-line at the JFreeReport home page.

3 Getting Started

3.1 Introduction

In this section, two sample applications are presented as an introduction to using JFreeReport:

- `HelloWorld.java` – a minimal application that illustrates just the basic steps in creating and displaying a report;
- `SwingIconsDemo.java` – a more complete example that demonstrates the use of various formatting options provided by JFreeReport;

These examples should be sufficient to get you started using JFreeReport in your own applications.

3.2 The Basic Steps

3.2.1 Overview

In standard usage, there are three major tasks in generating reports with JFreeReport:

- arrange for some data that can be accessed via the `TableModel` interface (that is, the model used by Swing's `JTable` class);
- create a `JFreeReport` object that will control the formatting of the generated report;
- link the data (`TableModel`) with the `JFreeReport` instance and pass the report to a print preview frame for presentation to the user;

Other variations are possible (for example, sending a report directly to file *without* previewing it first) but the above steps represent the most common scenario.

3.2.2 Step 1: The Data

JFreeReport is designed to work with data that is accessible via the `TableModel` interface. If you are already familiar with using Swing's `JTable` class, then this will cause you no trouble. If not, I recommend that you find out more about Swing's `JTable` class before you tackle JFreeReport. Tutorials can be found at Sun's Java website:

```
http://java.sun.com/
```

If you want to generate reports using data accessed via JDBC, you will be pleased to know that JFreeReport includes code for generating a `TableModel` instance from a JDBC `ResultSet`.

3.2.3 Step 2: The Report Definition

All report formatting information is recorded by an instance of the `JFreeReport` class. You have two options for creating this instance:

- write a report definition in XML format, and get JFreeReport to parse the definition and create a corresponding `JFreeReport` object;
- create a `JFreeReport` instance in code, and populate the report bands and elements using the JFreeReport API;

Putting your report definitions into an XML file is a good idea, because it allows you to change the formatting of your reports without recompiling your application.

Creating reports in code can allow additional flexibility if you want to vary the format of your reports at run-time, but has the disadvantage that your report formatting is "hard-coded" into your application.

3.2.4 Step 3: Previewing the Report

The final step—previewing the report—is mostly taken care of by the JFreeReport library. The print preview frame will display the report on-screen, provide controls to page forward and backward through the report, zoom in and zoom out, print the report, or export to a number of different file formats including PDF, HTML, CSV, Excel and XML.

3.3 Sample Application: Hello World

3.3.1 Overview

The `HelloWorld.java` application (included in the distribution) provides a basic overview of the steps for creating and viewing reports with JFreeReport.

3.3.2 The Data

The data for this example is a simple table:

Column1:	**Column2:**
`Hello`	`World!`

It is created using the following code:

```
private TableModel createData()
{

    Object[] columnNames = new String[] { "Column1", "Column2" };
    DefaultTableModel result = new DefaultTableModel(columnNames, 1);
    result.setValueAt("Hello", 0, 0);
    result.setValueAt("World!", 0, 1);
    return result;

}
```

The column names are important, since they will be referenced by elements within the report definition (see the next section).

3.3.3 The Report

The `JFreeReport` instance used to control the format of the "Hello World" report is created in code, so that the demo is self-contained:

```
private JFreeReport createReportDefinition()
{

    JFreeReport report = new JFreeReport();
    report.setName("A Very Simple Report");

    TextElement t1 = ItemFactory.createStringElement(
        "T1",
        new Rectangle2D.Double(0.0, 0.0, 150.0, 20.0),
        Color.black,
        ElementAlignment.LEFT.getOldAlignment(),
        ElementAlignment.MIDDLE.getOldAlignment(),
        null, // font
        "-",  // null string
        "Column1"
    );

    report.getItemBand().addElement(t1);

    TextElement t2 = ItemFactory.createStringElement(
        "T2",
        new Rectangle2D.Double(200.0, 0.0, 150.0, 20.0),
        Color.black,
        ElementAlignment.LEFT.getOldAlignment(),
        ElementAlignment.MIDDLE.getOldAlignment(),
        null, // font
        "-",  // null string
        "Column2"
    );

    report.getItemBand().addElement(t2);
    return report;

}
```

To understand what is going on here, you need to know that the report layout is controlled by *elements* that are added to *bands* within the report. A report consists of a number of (possibly empty) bands, including:

- the *report header* – printed once at the beginning of the report;
- the *report footer* – printed once at the end of the report;
- the *page header* – printed at the top of each page;
- the *page footer* – printed at the bottom of each page;
- the *item band* – printed once for each row of data in the `TableModel`;
- an additional *group header* and/or *group footer* for each *group* defined in the report.

In the `HelloWorld.java` application, only the item band is populated—the other bands remain empty (the default). A text element is added for each column of data in the `TableModel`.

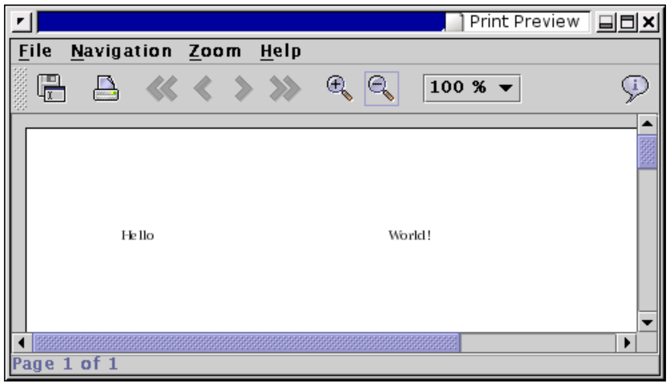
The `ItemFactory` class is used to create individual elements. Each element carries its own position, alignment, font settings and other presentation attributes. Notice also that each element in the example is tied back to the `TableModel` by a reference to a column name in the table.

3.3.4 Previewing the Report

To preview this report, the data is linked with the report (using the `setData(...)` method), and the report is passed to a print preview frame:

```
TableModel data = createData();
JFreeReport report = createReportDefinition();

report.setData(data);
try
{
    PreviewFrame preview = new PreviewFrame(report);
    preview.pack();
    preview.setVisible(true);
}
catch (ReportProcessingException e)
{
    Log.error("Failed to generate report ", e);
}
```

If you run the demo, you should see the following preview frame:



Note: to run the demo from the precompiled jar files in the JFreeReport distribution, you can use the following command:

```
java -classpath jfreereport-0.8.3-demo.jar
:jfreereport-0.8.3.jar:lib/jcommon-0.8.2.jar
:lib/gnujaxp.jar:lib/bsh-1.2b6.jar:lib/pixie.jar
:lib/itext-0.98.jar:lib/jakarta-poi-1.5.1-final-20020615.jar
com.jrefinery.report.demo.HelloWorld
```

As usual, colons should be replaced by semi-colons if you are using Windows.

3.4 Sample Application: SwingIconsDemo

3.4.1 Overview

The `SwingIconsDemo.java` application (included in the distribution) generates a report listing the icons included in Sun's *Java Look and Feel Graphics Repository*, a collection of standard icons distributed by Sun Microsystems for use with Java. You will need to download this icon set, and agree to Sun's terms and conditions, before you will be able to run the demo application.

This example uses an XML report definition file to create the `JFreeReport` instance that controls the final report output.

3.4.2 Running the Demo

You may find the material in the following sections easier to follow if you have already tried out the demo application. In this section I describe how to run the `SwingIconsDemo` application.

The `SwingIconsDemo.java` source file is included in the JFreeReport distribution, along with the report template file (`swing-icons.xml`).

As mentioned previously, running this demo requires you to first download the *Java Look and Feel Graphics Repository*, a collection of standard icons (provided by Sun) for use in Swing applications. These icons are contained in a jar file (`jlfgr-1_0.jar`) that you can download[4] from:

```
http://developer.java.sun.com/developer/techDocs/hi/repository/
```

When you run the `SwingIconsDemo` application, it will look for this file in a *directory* that is on the class path. I recommend that you place the file in the `jfreereport-0.8.3` directory, then use the following command to run the demo:

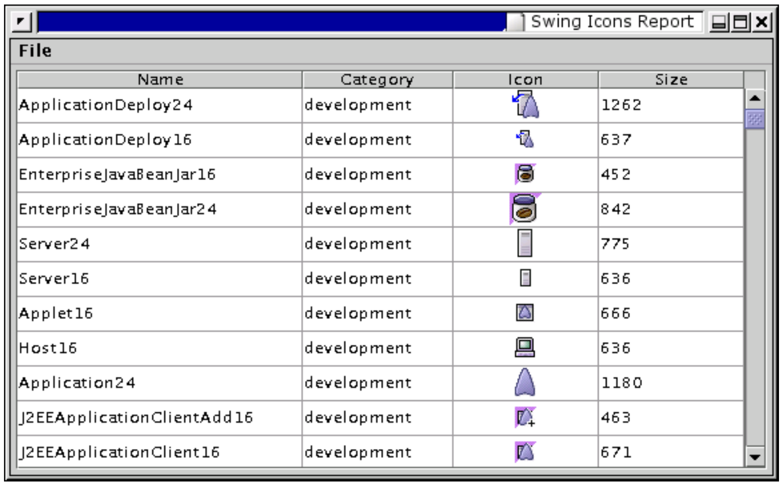
```
java -classpath .:jfreereport-0.8.3-demo.jar
:jfreereport-0.8.3.jar:lib/jcommon-0.8.2.jar
:lib/gnujaxp.jar:lib/bsh-1.2b6.jar:lib/pixie.jar
:lib/itext-0.98.jar:lib/jakarta-poi-1.5.1-final-20020615.jar
com.jrefinery.report.demo.SwingIconsDemo
```

Notice the addition of the "." at the beginning of the classpath to include the current directory in the classpath.

3.4.3 The Data

The `SwingIconsDemo` class uses the `SwingIconsDemoTableModel` class to store the data for the report. This table model contains code for reading the icons from the icon file (`jlfgr-1_0.jar`). The data is then displayed in a simple user interface using a `JTable`:

[4]Subject to acceptance of Sun's licence agreement.

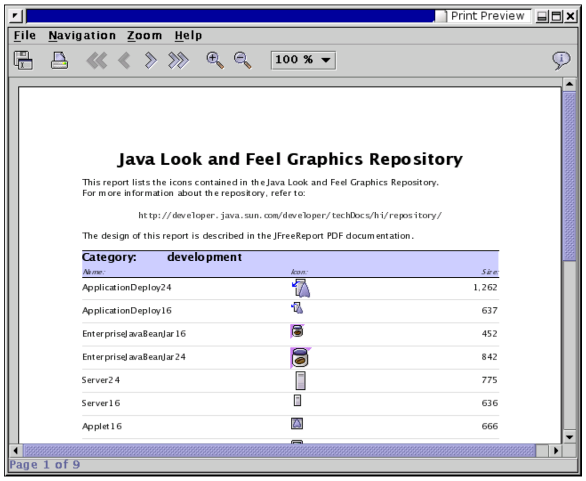


Name	Category	Icon	Size
ApplicationDeploy24	development		1262
ApplicationDeploy16	development		637
EnterpriseJavaBeanJar16	development		452
EnterpriseJavaBeanJar24	development		842
Server24	development		775
Server16	development		636
Applet16	development		666
Host16	development		636
Application24	development		1180
J2EEApplicationClientAdd16	development		463
J2EEApplicationClient16	development		671

Some important points to note about the dataset:

- the column names are *Name*, *Category*, *Icon* and *Size*. These names are used in the XML report template file to reference data items;
- the data is sorted by the *Category* column. This is important since the data will be grouped by *Category*. JFreeReport does not currently perform any sorting, so you need to provide the data pre-sorted.

When you run the `SwingIconsDemo` application, you will see a menu item that allows you to display a print preview window. When you preview the sample report, you will see output like this:



Java Look and Feel Graphics Repository

This report lists the icons contained in the Java Look and Feel Graphics Repository.
For more information about the repository, refer to:

http://developer.java.sun.com/developer/techDocs/hi/repository/

The design of this report is described in the JFreeReport PDF documentation.

Category: development

Name:	Icon:	Size:
ApplicationDeploy24		1,262
ApplicationDeploy16		637
EnterpriseJavaBeanJar16		452
EnterpriseJavaBeanJar24		842
Server24		775
Server16		636
Applet16		666

The format of the report is controlled by an XML report definition file, described in the next section.

3.5 The Report Definition File

3.5.1 Overview

The `swing-icons.xml` file contains a report definition in the format required by JFreeReport's "extended" parser. This parser is more powerful and flexible than the "simple" parser, at the expense of being more verbose. The old format is still in use, but doesn't support all the features of the JFreeReport engine. A utility is included with JFreeReport to convert reports from the "simple" format to the "extended" format.

Here is a skeleton version of the file (the missing pieces, marked with comments, are reproduced in later sections):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE report PUBLIC "-//JFreeReport//DTD report definition//EN//extended"
                        "http://jfreereport.sourceforge.net/extreport.dtd">

<report-definition name="First Report">

  <!-- INSERT PARSER CONFIGURATION HERE -->

  <!-- INSERT REPORT CONFIGURATION HERE -->

  <!-- INSERT STYLES HERE -->

  <!-- INSERT TEMPLATES HERE -->

  <!-- INSERT FUNCTIONS HERE -->

  <!-- ********************** -->
  <!-- * REPORT DESCRIPTION * -->
  <!-- ********************** -->

  <report-description>

    <!-- INSERT REPORT HEADER HERE -->

    <!-- INSERT REPORT FOOTER HERE -->

    <!-- INSERT PAGE HEADER HERE -->

    <!-- INSERT PAGE FOOTER HERE -->

    <!-- INSERT GROUP DEFINITIONS HERE -->

    <!-- INSERT ITEM BAND HERE -->

  </report-description>

</report-definition>
```

The report name is required, and is made available within the report as a *report property* (`report.name`).

3.5.2 The Parser Configuration

The *parser configuration* provides a mechanism for customising the behaviour of the report definition parser:

```
<!-- ************************ -->
<!-- * PARSER CONFIGURATION * -->
<!-- ************************ -->
<parser-config>
  <object-factory class="com.jrefinery.report.io.ext.factory.datasource.DataSourceCollector"/>
  <object-factory class="com.jrefinery.report.io.ext.factory.objects.DefaultClassFactory"/>
  <element-factory class="com.jrefinery.report.io.ext.factory.elements.DefaultElementFactory"/>
  <stylekey-factory class="com.jrefinery.report.io.ext.factory.stylekey.DefaultStyleKeyFactory"/>
  <stylekey-factory class="com.jrefinery.report.io.ext.factory.stylekey.PageableLayoutStyleKeyFactory"/>
  <template-factory class="com.jrefinery.report.io.ext.factory.templates.DefaultTemplateCollection"/>
  <datasource-factory class="com.jrefinery.report.io.ext.factory.datasource.DefaultDataSourceFactory"/>
</parser-config>
```

Most of the time, you will just copy and paste this default configuration.

3.5.3 The Report Configuration

The *report configuration* allows you to control certain aspects of the way that JFreeReport works. In the example, the default page format and the initial height and width of the preview frame is specified:

```
<!-- ************************ -->
<!-- * REPORT CONFIGURATION * -->
<!-- ************************ -->
<report-config>
  <defaultpageformat orientation="portrait"
                     pageformat="LETTER"
                     topmargin="72"
                     bottommargin="72"
                     leftmargin="72"
                     rightmargin="72"/>
  <configuration>
    <property name="com.jrefinery.report.preview.PreferredHeight">480</property>
    <property name="com.jrefinery.report.preview.PreferredWidth">640</property>
  </configuration>
</report-config>
```

All the valid `pageformat` options are specified in the `PageFormatFactory` class. The margins are specified in points (1/72 inch).

Additional details on the available configuration parameters can be found in the documentation for the `ReportConfiguration` class.

3.5.4 Styles

The *Styles* element is empty in this example:

```
<!-- ********** -->
<!-- * STYLES * -->
<!-- ********** -->
<styles>
</styles>
```

3.5.5 Templates

The *Templates* element is empty in this example:

```
<!-- ************* -->
<!-- * TEMPLATES * -->
<!-- ************* -->
<templates>
</templates>
```

3.5.6 Report Functions

The example uses several *report functions* to calculate summary information for display on the report. It is necessary to declare functions within the XML report template file—individual report elements can then reference particular functions by name.

When you declare a function, you give it a name and specify the Java class that implements the function. Most functions also require some properties to be specified (you should refer to the documentation or source code for the class that implements the function to find out which properties are required).

Here is the XML used to define the functions used in the sample report:

```
<!-- ************* -->
<!-- * FUNCTIONS * -->
<!-- ************* -->
<functions>

  <property-ref name="report.date"/>

  <function name="PageNumber" class="com.jrefinery.report.function.PageFunction">
  </function>

  <function name="GroupCount" class="com.jrefinery.report.function.ItemCountFunction">
    <properties>
      <property name="field">Name</property>
      <property name="group">Category</property>
    </properties>
  </function>

  <function name="GroupSum" class="com.jrefinery.report.function.ItemSumFunction">
    <properties>
      <property name="field">Size</property>
      <property name="group">Category</property>
    </properties>
  </function>

  <function name="OverallCount" class="com.jrefinery.report.function.ItemCountFunction">
    <properties>
      <property name="field">Name</property>
    </properties>
  </function>

  <function name="OverallSum" class="com.jrefinery.report.function.ItemSumFunction">
    <properties>
      <property name="field">Size</property>
    </properties>
  </function>
</functions>
```

The first entry in the function declarations marks a report property for access. The (`report.date`) property is automatically set up by JFreeReport at report generation time, and it returns the current date. In the example, the report date is displayed in the page header.

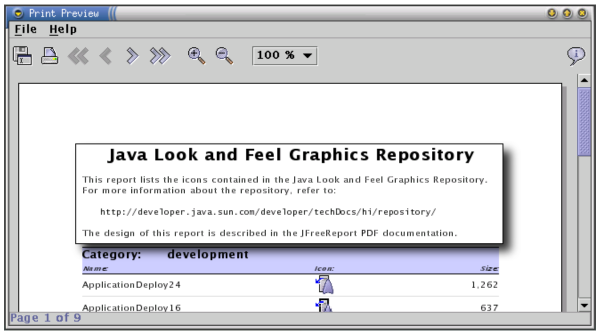
The first function (`PageNumber`) returns the current page number. It does not require any properties to be set. In the example report, the page number is displayed in the page footer.

There are two functions declared to count report items. The first (`GroupCount`) counts items within the `Category` group, and is used in the group footer. The second (`OverallCount`) counts items for the entire report (simply by omitting the `group` property), and is used in the report footer.

There are two functions declared to sum the icon size field. The first (`GroupSum`) calculates a running total within the `Category` group, and is used in the group footer. The second (`OverallSum`) calculates a running total for the entire report, and is used in the report footer.

3.5.7 The Report Header

The *report header* is a band that is printed once at the beginning of a report. In our example, the report header contains five labels (fixed text items), highlighted in the figure below:



Java Look and Feel Graphics Repository

This report lists the icons contained in the Java Look and Feel Graphics Repository.
For more information about the repository, refer to:

http://developer.java.sun.com/developer/techDocs/hi/repository/

The design of this report is described in the JFreeReport PDF documentation.

Category: development

Name:	Icon:	Size:
ApplicationDeploy24		1,262
ApplicationDeploy16		637

The minimum height of the band is 112 points. The default font is plain `SansSerif`, at 10 points. Here is the XML used to define the report header:

```
<!-- ================= -->
<!-- = REPORT HEADER = -->
<!-- ================= -->
<report-header name="report-header-band">

  <style>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">112.0</basic-object>
      <basic-object name="width" class="java.lang.Double">0.0</basic-object>
    </compound-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-size">10</basic-key>
    <basic-key name="font-bold">false</basic-key>
    <basic-key name="font-italic">false</basic-key>
    <basic-key name="pagebreak-after">false</basic-key>
  </style>

  <default-style>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-size">10</basic-key>
    <basic-key name="font-bold">false</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </default-style>

  <element name="title1" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">2.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">18.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="alignment">center</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="font-size">18</basic-key>
      <basic-key name="font-bold">true</basic-key>
      <basic-key name="font-italic">false</basic-key>
      <basic-key name="paint">black</basic-key>
    </style>
    <template references="label">
      <basic-object name="content">Java Look and Feel Graphics Repository</basic-object>
      <basic-object name="nullValue">null</basic-object>
    </template>
  </element>

  <element name="description1" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">32.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">10.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="paint">black</basic-key>
    </style>
    <template references="label">
```

```
      <basic-object name="content">This report lists the icons contained in the Java Look and Feel Graphics Repository.</basic-object>
      <basic-object name="nullValue">null</basic-object>
    </template>
  </element>

  <element name="description2" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">44.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">10.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="paint">black</basic-key>
    </style>
    <template references="label">
      <basic-object name="content">For more information about the repository, refer to:</basic-object>
      <basic-object name="nullValue">null</basic-object>
    </template>
  </element>

  <element name="url1" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">68.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">10.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="alignment">center</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="font">Monospaced</basic-key>
      <basic-key name="font-size">9</basic-key>
      <basic-key name="paint">black</basic-key>
    </style>
    <template references="label">
      <basic-object name="content">http://developer.java.sun.com/developer/techDocs/hi/repository/</basic-object>
      <basic-object name="nullValue">null</basic-object>
    </template>
  </element>

  <element name="description3" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">92.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">10.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="paint">black</basic-key>
    </style>
    <template references="label">
      <basic-object name="content">The design of this report is described in the JFreeReport PDF documentation.</basic-object>
      <basic-object name="nullValue">null</basic-object>
    </template>
  </element>

</report-header>
```

Special points to note:

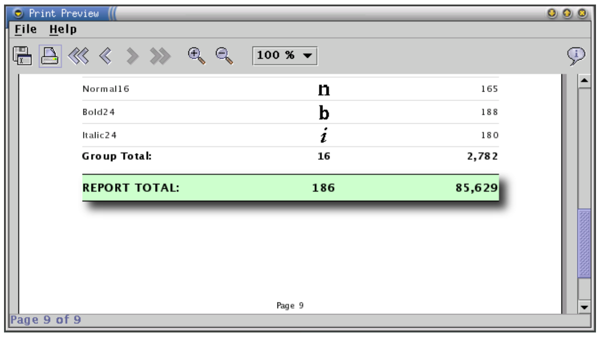
- the x and y coordinates for each element are measured in points (1/72 inch) relative to the band's origin;
- the y-values increase as you move down the band / page;
- the width of each label is 100 percent (percentages are coded as negative values), which means that the element area extends across the full width of the report. This is particularly important for the first label, which is centered within its area, and so appears at the center of the band;

- the default font is overridden for first and fourth labels. For the other labels, the default font specified for the header is used.

Although the example displays only labels, you can use any report element in a report header, including those that display function values.

3.5.8 The Report Footer

The *report footer* is a band that is printed once at the end of a report. In the example, the report footer contains a label and two number fields that get their values from specific report functions. It also contains two lines and a rectangle, use to enhance the appearance of the report. The footer is highlighted in the figure below:



```
Normal16        n       165
Bold24          b       188
Italic24        i       180
Group Total:    16    2,782

REPORT TOTAL:   186  85,629

Page 9
Page 9 of 9
```

Here is the XML used to define the report footer:

```
<!-- ================= -->
<!-- = REPORT FOOTER = -->
<!-- ================= -->
<report-footer name="report-footer-band">

  <style>
    <basic-key name="font-bold">true</basic-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">30.0</basic-object>
      <basic-object name="width" class="java.lang.Double">0.0</basic-object>
    </compound-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-size">12</basic-key>
    <basic-key name="font-italic">false</basic-key>
    <basic-key name="pagebreak-before">false</basic-key>
  </style>

  <default-style>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-size">12</basic-key>
    <basic-key name="font-bold">true</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </default-style>

  <element name="rect1" type="shape/generic">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">-100.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="stroke" class="java.awt.BasicStroke">0.0</basic-key>
      <basic-key name="paint">#ccffcc</basic-key>
      <basic-key name="draw-shape">false</basic-key>
      <basic-key name="fill-shape">true</basic-key>
      <basic-key name="scale">true</basic-key>
      <basic-key name="keepAspectRatio">false</basic-key>
    </style>
    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Rectangle2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="height">100.0</basic-object>
        <basic-object name="width">100.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-object>
    </datasource>
  </element>

  <element name="line1" type="shape/generic">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">0.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="stroke" class="java.awt.BasicStroke">0.5</basic-key>
      <basic-key name="draw-shape">true</basic-key>
      <basic-key name="fill-shape">false</basic-key>
      <basic-key name="keepAspectRatio">false</basic-key>
      <basic-key name="scale">true</basic-key>
    </style>
    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Line2D$Float">
        <basic-object name="x2">100.0</basic-object>
        <basic-object name="x1">0.0</basic-object>
        <basic-object name="y2">0.0</basic-object>
        <basic-object name="y1">0.0</basic-object>
      </compound-object>
    </datasource>
  </element>

  <element name="line2" type="shape/generic">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">30.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">0.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="stroke" class="java.awt.BasicStroke">0.5</basic-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="draw-shape">true</basic-key>
      <basic-key name="fill-shape">false</basic-key>
      <basic-key name="scale">true</basic-key>
      <basic-key name="keepAspectRatio">false</basic-key>
    </style>
    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Line2D$Float">
        <basic-object name="x2">100.0</basic-object>
        <basic-object name="x1">0.0</basic-object>
        <basic-object name="y2">30.0</basic-object>
        <basic-object name="y1">30.0</basic-object>
      </compound-object>
    </datasource>
  </element>

  <element name="reportTotalLabel" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">10.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">12.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-50.0</basic-object>
      </compound-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="paint">black</basic-key>
    </style>
    <template references="label">
      <basic-object name="content">REPORT TOTAL:</basic-object>
      <basic-object name="nullValue">null</basic-object>
    </template>
  </element>

  <element name="reportCountField" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">-50.0</basic-object>
```

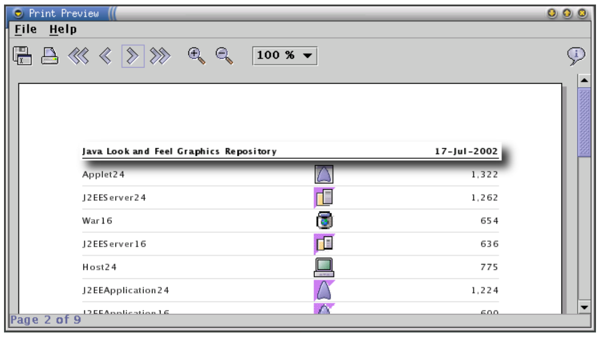
```
        <basic-object name="y">10.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-30.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">center</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="dynamic_height">false</basic-key>
    </style>
    <template references="number-field">
      <basic-object name="field">OverallCount</basic-object>
      <basic-object name="format">#0</basic-object>
      <basic-object name="nullValue">-</basic-object>
    </template>
  </element>

  <element name="reportSumField" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">-80.0</basic-object>
        <basic-object name="y">10.0</basic-object>
      </compound-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-20.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">right</basic-key>
      <basic-key name="valignment">bottom</basic-key>
      <basic-key name="dynamic_height">false</basic-key>
    </style>
    <template references="number-field">
      <basic-object name="field">OverallSum</basic-object>
      <basic-object name="format">#,##0</basic-object>
      <basic-object name="nullValue">-</basic-object>
    </template>
  </element>
</report-footer>
```

Special points to note:

- the lines and shapes are declared first, so that they are drawn first (to avoid obscuring the other report elements in the band);
- the coordinates for the `line` elements both represent a single point, in this example. This is a special case, where JFreeReport draws a horizontal line passing through the point and covering the full width of the band;
- the `number-field` elements specify the name of the function that supplies the element value. Each function must be declared elsewhere in the report definition (see section 3.5.6);
- for the `number-field` report elements, you can define a format string to format the output. Internally, this is passed to an instance of Java's `NumberFormat` class to control the formatting.

3.5.9 The Page Header

The *page header* appears at the top of every page, with the possible exception of the first and last pages. In the example, the page header contains a label and a date field that displays the current date. This is highlighted in the figure below:



Java Look and Feel Graphics Repository		17-Jul-2002
Applet24		1,322
J2EEServer24		1,262
War16		654
J2EEServer16		636
Host24		775
J2EEApplication24		1,224

Here is the XML used to define the page header:

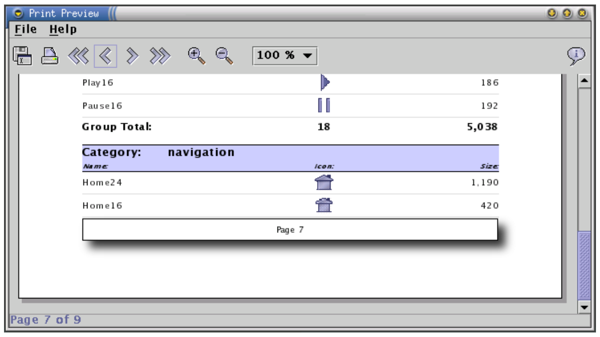
```
<!-- ============== -->
<!-- = PAGE HEADER = -->
<!-- ============== -->
<page-header name="anonymousBand@1319318">

  <style>
    <basic-key name="font-bold">true</basic-key>
    <basic-key name="display-on-lastpage">true</basic-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">18.0</basic-object>
      <basic-object name="width" class="java.lang.Double">0.0</basic-object>
    </compound-key>
    <basic-key name="font-size">9</basic-key>
    <basic-key name="display-on-firstpage">false</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </style>

  <default-style>
    <basic-key name="font-bold">true</basic-key>
    <basic-key name="font-size">9</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </default-style>

  <element name="@anonymousc" type="shape/generic">
    <style>
      <basic-key name="stroke" class="java.awt.BasicStroke">0.0</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-key>
      <basic-key name="draw-shape">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">-100.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="keepAspectRatio">false</basic-key>
      <basic-key name="scale">true</basic-key>
      <basic-key name="paint">#afafaf</basic-key>
      <basic-key name="fill-shape">true</basic-key>
    </style>
    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Rectangle2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="height">100.0</basic-object>
        <basic-object name="width">100.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-object>
    </datasource>
  </element>

  <element name="@anonymousd" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">14.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-50.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="label">
      <basic-object name="nullValue">null</basic-object>
      <basic-object name="content">Java Look and Feel Graphics Repository</basic-object>
    </template>
  </element>

  <element name="@anonymouse" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">-80.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">14.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-20.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">right</basic-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="date-field">
      <basic-object name="nullValue">-</basic-object>
      <basic-object name="format">d-MMM-yyyy</basic-object>
      <basic-object name="field">report.date</basic-object>
    </template>
  </element>

  <element name="@anonymousf" type="shape/generic">
    <style>
      <basic-key name="stroke" class="java.awt.BasicStroke">2.0</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">16.0</basic-object>
      </compound-key>
      <basic-key name="draw-shape">true</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">0.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="keepAspectRatio">false</basic-key>
      <basic-key name="scale">true</basic-key>
      <basic-key name="paint">#cfcfcf</basic-key>
      <basic-key name="fill-shape">false</basic-key>
    </style>
    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Line2D$Float">
        <basic-object name="x2">100.0</basic-object>
        <basic-object name="x1">0.0</basic-object>
        <basic-object name="y2">16.0</basic-object>
        <basic-object name="y1">16.0</basic-object>
      </compound-object>
    </datasource>
  </element>
</page-header>
```

Special points to note:

- the page header is suppressed on the first page, since it would conflict with the report header;
- the `date-field` element references (by name) the report date property. This property is declared elsewhere in the report definition (see section 3.5.6);
- the `date-field` element has a format attribute to control date formatting. This is passed to an instance of Java's `SimpleDateFormat` class to control the formatting.

3.5.10 The Page Footer

The *page footer* appears at the bottom of every page, with the possible exception of the first and last pages. In the example, the page footer contains just a number field that displays the page number. This is highlighted in the figure below:



Play16	▶	186
Pause16	❚❚	192
Group Total:	**18**	**5,038**

Category: navigation

Name:	Icon:	Size:
Home24		1,190
Home16		420

Page 7

Here is the XML used to define the page footer:

```
<!-- =============== -->
<!-- = PAGE FOOTER = -->
<!-- =============== -->
<page-footer name="anonymousBand@7205769">

  <style>
    <basic-key name="font-bold">false</basic-key>
    <basic-key name="display-on-lastpage">true</basic-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">14.0</basic-object>
      <basic-object name="width" class="java.lang.Double">0.0</basic-object>
    </compound-key>
    <basic-key name="font-size">9</basic-key>
    <basic-key name="display-on-firstpage">true</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </style>

  <default-style>
    <basic-key name="font-bold">false</basic-key>
    <basic-key name="font-size">9</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </default-style>

  <element name="@anonymous10" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">3.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">center</basic-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="number-field">
      <basic-object name="nullValue">null</basic-object>
      <basic-object name="format">Page #0</basic-object>
      <basic-object name="field">PageNumber</basic-object>
    </template>
  </element>
</page-footer>
```

Special points to note:

- the page footer can be suppressed on the first or last pages, to avoid conflict with the report header and/or the report footer. In this example, the page footer is displayed on all pages;
- the elements using `number-field` templates reference (by name) the function that returns the page number. This function is declared elsewhere in the report definition (see section 3.5.6);

- the `number-field` element has a format attribute to control numberformatting. This is passed to an instance of Java's `NumberFormat` class to control formatting.

3.5.11 Report Groups

Report groups are used to aggregate report items according to the values in certain fields. In this example, only one group is defined (on the `Category` field) in addition to the default group, but it is possible to create multiple (nested) groups with JFreeReport.

For each group, you can define a *group header* that is printed at the start of each group instance. Likewise, you can define a *group footer* that is printed at the end of each group instance.

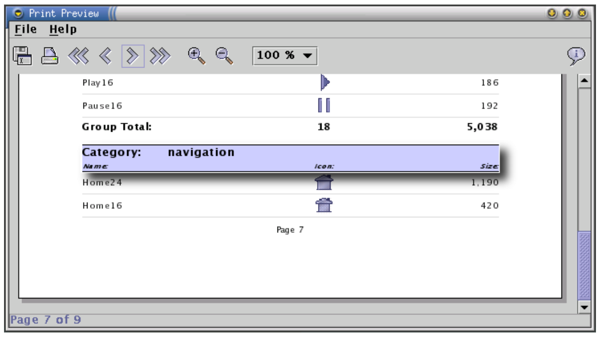
The XML used to define the `Category` group is:

```
<!-- ========== -->
<!-- = GROUPS = -->
<!-- ========== -->
<groups>

  <!-- default group -->
  <group name="default">
    <fields>
    </fields>
    <group-header name="anonymousBand@4732779">
      <style>
      </style>
      <default-style>
      </default-style>
    </group-header>
    <group-footer name="anonymousBand@6100951">
      <style>
      </style>
      <default-style>
      </default-style>
    </group-footer>
  </group>

  <!-- category group -->
  <group name="Category">
    <fields>
      <field>Category</field>
    </fields>

    <!-- INSERT GROUP HEADER HERE -->

    <!-- INSERT GROUP FOOTER HERE -->

  </group>
</groups>
```

An important point to note is that the data in the `TableModel` should be *presorted* according to the fields defined in the report groups, otherwise you will get some unusual results. JFreeReport does not perform any sorting itself.

As mentioned already, JFreeReport will print a *group header* every time a new group starts, and a *group footer* every time a group ends. The header and footer definitions should be inserted at the points indicated in the XML fragment above.

The group header is highlighted in the figure below:



```
Play16                          186
Pause16                         192
Group Total:          18      5,038

Category:   navigation
Name                 Icon      Size
Home24                        1,190
Home16                          420

              Page 7
```

Page 7 of 9

Here is the XML used to define the group header:

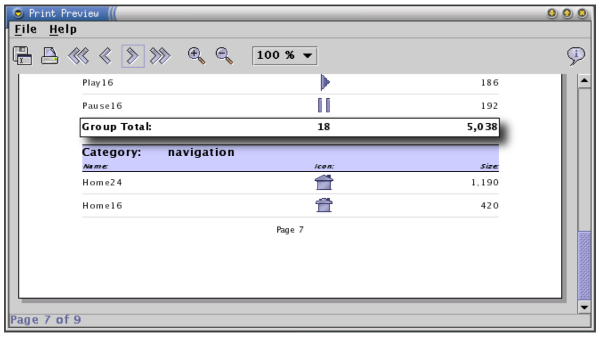
```
<group-header name="anonymousBand@3921842">
  <style>
    <basic-key name="font-bold">true</basic-key>
    <basic-key name="repeat-header">false</basic-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">30.0</basic-object>
      <basic-object name="width" class="java.lang.Double">0.0</basic-object>
    </compound-key>
    <basic-key name="font-size">12</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
    <basic-key name="pagebreak-before">false</basic-key>
  </style>

  <default-style>
    <basic-key name="font-bold">true</basic-key>
    <basic-key name="font-size">12</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </default-style>

  <element name="@anonymous11" type="shape/generic">
    <style>
      <basic-key name="stroke" class="java.awt.BasicStroke">0.0</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-key>
      <basic-key name="draw-shape">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">-100.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="keepAspectRatio">false</basic-key>
      <basic-key name="scale">true</basic-key>
      <basic-key name="paint">#ccccff</basic-key>
      <basic-key name="fill-shape">true</basic-key>
    </style>

    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Rectangle2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="height">100.0</basic-object>
        <basic-object name="width">100.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-object>
    </datasource>
  </element>

  <element name="@anonymous12" type="shape/generic">
    <style>
      <basic-key name="stroke" class="java.awt.BasicStroke">0.5</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">0.0</basic-object>
      </compound-key>
      <basic-key name="draw-shape">true</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">0.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="keepAspectRatio">false</basic-key>
      <basic-key name="scale">true</basic-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="fill-shape">false</basic-key>
    </style>
    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Line2D$Float">
        <basic-object name="x2">100.0</basic-object>
        <basic-object name="x1">0.0</basic-object>
        <basic-object name="y2">0.0</basic-object>
        <basic-object name="y1">0.0</basic-object>
      </compound-object>
    </datasource>
  </element>

  <element name="@anonymous13" type="shape/generic">
    <style>
      <basic-key name="stroke" class="java.awt.BasicStroke">0.5</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">30.0</basic-object>
      </compound-key>
      <basic-key name="draw-shape">true</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">0.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
      </compound-key>
      <basic-key name="keepAspectRatio">false</basic-key>
      <basic-key name="scale">true</basic-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="fill-shape">false</basic-key>
    </style>
    <datasource type="StaticDataSource">
      <compound-object name="value" class="java.awt.geom.Line2D$Float">
        <basic-object name="x2">100.0</basic-object>
        <basic-object name="x1">0.0</basic-object>
        <basic-object name="y2">30.0</basic-object>
        <basic-object name="y1">30.0</basic-object>
      </compound-object>
    </datasource>
  </element>

  <element name="@anonymous14" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">3.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">12.0</basic-object>
        <basic-object name="width" class="java.lang.Double">92.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="label">
      <basic-object name="nullValue">null</basic-object>
      <basic-object name="content">Category:</basic-object>
    </template>
  </element>

  <element name="@anonymous15" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">96.0</basic-object>
        <basic-object name="y">3.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">12.0</basic-object>
        <basic-object name="width" class="java.lang.Double">120.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="string-field">
      <basic-object name="nullValue">-</basic-object>
      <basic-object name="field">Category</basic-object>
    </template>
  </element>

  <element name="@anonymous16" type="text/plain">
    <style>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="font-size">8</basic-key>
      <basic-key name="font-bold">false</basic-key>
      <basic-key name="font-italic">true</basic-key>
      <basic-key name="paint">black</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">20.0</basic-object>
      </compound-key>
      <basic-key name="alignment">left</basic-key>
```

```
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-50.0</basic-object>
      </compound-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="label">
      <basic-object name="nullValue">null</basic-object>
      <basic-object name="content">Name:</basic-object>
    </template>
  </element>

  <element name="@anonymous17" type="text/plain">
    <style>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="font-size">8</basic-key>
      <basic-key name="font-bold">false</basic-key>
      <basic-key name="font-italic">true</basic-key>
      <basic-key name="paint">black</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">-50.0</basic-object>
        <basic-object name="y">20.0</basic-object>
      </compound-key>
      <basic-key name="alignment">left</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-30.0</basic-object>
      </compound-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="label">
      <basic-object name="nullValue">null</basic-object>
      <basic-object name="content">Icon:</basic-object>
    </template>
  </element>

  <element name="@anonymous18" type="text/plain">
    <style>
      <basic-key name="dynamic_height">false</basic-key>
      <basic-key name="font-size">8</basic-key>
      <basic-key name="font-bold">false</basic-key>
      <basic-key name="font-italic">true</basic-key>
      <basic-key name="paint">black</basic-key>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">-80.0</basic-object>
        <basic-object name="y">20.0</basic-object>
      </compound-key>
      <basic-key name="alignment">right</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-20.0</basic-object>
      </compound-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="label">
      <basic-object name="nullValue">null</basic-object>
      <basic-object name="content">Size:</basic-object>
    </template>
  </element>

</group-header>
```

Some points to note:

- the `pagebreak` attribute is set to `false`. If you change this to `true`, the reporting engine will start a new page before printing the group header;
- a `string-field` template is used to display the value in the `Category` column of the report's `TableModel`;

The group footer is highlighted in the figure below:



```
Play16                       186
Pause16                      192
Group Total:       18      5,038
Category:     navigation
Name:              Icon:    Size:
Home24                     1,190
Home16                       420
              Page 7
Page 7 of 9
```

Here is the XML used to define the group footer:

```
<group-footer name="anonymousBand@8361137">
  <style>
    <basic-key name="font-bold">true</basic-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">30.0</basic-object>
      <basic-object name="width" class="java.lang.Double">0.0</basic-object>
    </compound-key>
    <basic-key name="font-size">11</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
    <basic-key name="pagebreak-before">false</basic-key>
  </style>

  <default-style>
    <basic-key name="font-bold">true</basic-key>
    <basic-key name="font-size">11</basic-key>
    <basic-key name="font">SansSerif</basic-key>
    <basic-key name="font-italic">false</basic-key>
  </default-style>

  <element name="@anonymous19" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">0.0</basic-object>
        <basic-object name="y">5.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">100.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">left</basic-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>
    <template references="label">
      <basic-object name="nullValue">null</basic-object>
      <basic-object name="content">Group Total:</basic-object>
    </template>
  </element>

  <element name="@anonymous1a" type="text/plain">
    <style>
      <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
        <basic-object name="x">-50.0</basic-object>
        <basic-object name="y">5.0</basic-object>
      </compound-key>
      <basic-key name="dynamic_height">false</basic-key>
      <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
        <basic-object name="height" class="java.lang.Double">9.0</basic-object>
        <basic-object name="width" class="java.lang.Double">-30.0</basic-object>
      </compound-key>
      <basic-key name="paint">black</basic-key>
      <basic-key name="alignment">center</basic-key>
      <basic-key name="valignment">bottom</basic-key>
    </style>

    <template references="number-field">
      <basic-object name="nullValue">-</basic-object>
      <basic-object name="format">#0</basic-object>
      <basic-object name="field">GroupCount</basic-object>
    </template>
  </element>

  <element name="@anonymous1b" type="text/plain">
```

```
      <style>
        <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
          <basic-object name="x">-80.0</basic-object>
          <basic-object name="y">5.0</basic-object>
        </compound-key>
        <basic-key name="dynamic_height">false</basic-key>
        <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
          <basic-object name="height" class="java.lang.Double">9.0</basic-object>
          <basic-object name="width" class="java.lang.Double">-20.0</basic-object>
        </compound-key>
        <basic-key name="paint">black</basic-key>
        <basic-key name="alignment">right</basic-key>
        <basic-key name="valignment">bottom</basic-key>
      </style>
      <template references="number-field">
        <basic-object name="nullValue">-</basic-object>
        <basic-object name="format">#,##0</basic-object>
        <basic-object name="field">GroupSum</basic-object>
      </template>
    </element>
  </group-footer>
```

Some points to note:

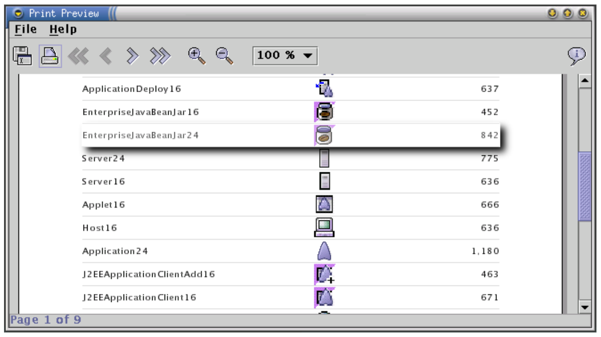
- the two elements that use `number-field` templates reference (by name) functions declared elsewhere in the report definition (see section 3.5.6);

3.5.12 The Item Band

The item band is displayed once per row of data in the report's `TableModel`. In the example, the item band contains four elements:

- a string field displaying the name of the icon;
- an image field displaying the icon itself;
- a number field displaying the size of the icon image (in bytes);
- a shape element that displays a line across the page.

One instance of the item band is highlighted in the figure below:



```
ApplicationDeploy16              637
EnterpriseJavaBeanJar16          452
EnterpriseJavaBeanJar24          842
Server24                         775
Server16                         636
Applet16                         666
Host16                           636
Application24                  1,180
J2EEApplicationClientAdd16       463
J2EEApplicationClient16          671
```

Here is the XML used to define the item band:

```
<!-- ============= -->
<!-- = ITEM BAND = -->
<!-- ============= -->
<itemband name="anonymousBand@581472">

  <!-- the band style -->
```

```
<style>
  <basic-key name="font-bold">false</basic-key>
  <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
    <basic-object name="height" class="java.lang.Double">26.0</basic-object>
    <basic-object name="width" class="java.lang.Double">0.0</basic-object>
  </compound-key>
  <basic-key name="font-size">10</basic-key>
  <basic-key name="font">SansSerif</basic-key>
  <basic-key name="font-italic">false</basic-key>
</style>

<!-- the default element style -->
<default-style>
  <basic-key name="font">SansSerif</basic-key>
  <basic-key name="font-size">10</basic-key>
  <basic-key name="font-bold">false</basic-key>
  <basic-key name="font-italic">false</basic-key>
</default-style>

<element name="@anonymous1c" type="shape/generic">
  <style>
    <basic-key name="stroke" class="java.awt.BasicStroke">0.1</basic-key>
    <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
      <basic-object name="x">0.0</basic-object>
      <basic-object name="y">25.0</basic-object>
    </compound-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">0.0</basic-object>
      <basic-object name="width" class="java.lang.Double">-100.0</basic-object>
    </compound-key>
    <basic-key name="keepAspectRatio">false</basic-key>
    <basic-key name="scale">true</basic-key>
    <basic-key name="paint">#dfdfdf</basic-key>
    <basic-key name="draw-shape">true</basic-key>
    <basic-key name="fill-shape">false</basic-key>
  </style>
  <datasource type="StaticDataSource">
    <compound-object name="value" class="java.awt.geom.Line2D$Float">
      <basic-object name="x2">100.0</basic-object>
      <basic-object name="x1">0.0</basic-object>
      <basic-object name="y2">25.0</basic-object>
      <basic-object name="y1">25.0</basic-object>
    </compound-object>
  </datasource>
</element>

<!-- element to display the name from the TableModel -->
<element name="@anonymous1d" type="text/plain">
  <style>
    <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
      <basic-object name="x">0.0</basic-object>
      <basic-object name="y">8.0</basic-object>
    </compound-key>
    <basic-key name="dynamic_height">false</basic-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">10.0</basic-object>
      <basic-object name="width" class="java.lang.Double">-50.0</basic-object>
    </compound-key>
    <basic-key name="paint">black</basic-key>
    <basic-key name="alignment">left</basic-key>
    <basic-key name="valignment">bottom</basic-key>
  </style>
  <template references="string-field">
    <basic-object name="nullValue">-</basic-object>
    <basic-object name="field">Name</basic-object>
  </template>
</element>

<!-- element to display icon from the TableModel -->
<element name="@anonymous1e" type="image/generic">
  <style>
    <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
      <basic-object name="x">-50.0</basic-object>
      <basic-object name="y">1.0</basic-object>
    </compound-key>
    <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
      <basic-object name="height" class="java.lang.Double">24.0</basic-object>
      <basic-object name="width" class="java.lang.Double">-30.0</basic-object>
    </compound-key>
    <basic-key name="keepAspectRatio">true</basic-key>
    <basic-key name="dynamic_height">false</basic-key>
    <basic-key name="scale">false</basic-key>
    <basic-key name="paint">white</basic-key>
  </style>
  <template references="image-field">
    <basic-object name="field">Icon</basic-object>
  </template>
</element>

<!-- element to display the file size from the TableModel -->
<element name="@anonymous1f" type="text/plain">
```

```
      <style>
        <compound-key name="absolute_pos" class="java.awt.geom.Point2D$Float">
          <basic-object name="x">-80.0</basic-object>
          <basic-object name="y">8.0</basic-object>
        </compound-key>
        <basic-key name="dynamic_height">false</basic-key>
        <compound-key name="min-size" class="com.jrefinery.report.targets.FloatDimension">
          <basic-object name="height" class="java.lang.Double">10.0</basic-object>
          <basic-object name="width" class="java.lang.Double">-20.0</basic-object>
        </compound-key>
        <basic-key name="paint">black</basic-key>
        <basic-key name="alignment">right</basic-key>
        <basic-key name="valignment">bottom</basic-key>
      </style>
      <template references="number-field">
        <basic-object name="nullValue">-</basic-object>
        <basic-object name="format">#,##0</basic-object>
        <basic-object name="field">Size</basic-object>
      </template>
    </element>
  </itemband>
```

Some points to note:

- unless overridden, text items in the band inherit the font settings specified in the *default-style* element for the band;
- the line element is scaled to cover the full width of the band;
- each of the remaining elements displays an item from the report's `TableModel`. The `field` attribute in the *template* element specifies the name of the column in the `TableModel` from which the data item is read.

3.6 Previewing the Report

Once the report template has been constructed, and the data is available, it is simple to create a print preview frame to display the report. The first step is to construct a `JFreeReport` instance from the XML report template file, and link the dataset to this instance. In the example, I use the following code:

```
URL in  = getClass().getResource("/com/jrefinery/report/demo/swing-icons.xml");
this.report = parseReport(in);
this.report.setData(this.data);
```

This assumes that the `swing-icons.xml` file is located in the same directory as the `SwingIconsDemo.class` file.

To display the print preview frame for this report:

```
PreviewFrame frame = new PreviewFrame(this.report);
frame.setLargeIconsEnabled(true);
frame.setToolbarFloatable(false);
frame.pack();
RefineryUtilities.positionFrameRandomly(frame);
frame.setVisible(true);
frame.requestFocus();
```

The print preview frame then provides all the standard options to the user, including paging back and forward through the report, zooming, printing, and export to PDF.

4 The “Simple” Report Definition Format

4.1 Overview

JFreeReport uses two XML-based text formats to store report definitions in text files:

- the “simple” format;
- the “extended” format.

The simple format does not support all the capabilities of the JFreeReport reporting engine, but it is relatively easy to work with. The extended format can fully describe a `JFreeReport` object, but it is more verbose than the simple format, and not so easy to work with.

In this section, the simple format is described. We assume that you have some knowledge of XML. If you need to learn more, a good reference is “XML in a Nutshell”, by Elliotte Rusty Harold and W. Scott Means (published by O’Reilly & Associates, Inc).

4.2 Report Definition Files

A report definition file is used to create a `JFreeReport` instance, which in turn controls the presentation of your data. Thus, you can control most aspects of your report’s appearance simply by editing the report definition file.

> *A word of warning: the report templates are still being developed. It is possible that changes will be made to the format in any new version of JFreeReport up until version 1.0.0 is released.*

4.3 The Document Type Definition

A *document type definition* (DTD) has been written for the simple file format—for convenience, a copy of the DTD has been included in Appendix A. You can view the very latest version of the DTD at:

```
http://jfreereport.sourceforge.net/report.dtd
```

Thomas Morgner is the author of the DTD.

4.4 The XML Declaration

The first line of the report template should contain the *XML declaration*. All XML documents should have an XML declaration.

A typical report template file will look like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!DOCTYPE report
  PUBLIC "-//JFreeReport//DTD report definition//EN//simple"
         "http://jfreereport.sourceforge.net/report.dtd">

<report name="My Report" pageformat="LETTER" orientation="portrait">

    <!-- report definition goes here  -->

</report>
```

You can choose to include or exclude the `DOCTYPE` element. To quote Thomas Morgner, JFreeReport Project Leader:

> “It is heavily recommended, that these doctypes are included in the document. They also ease up the editing of the report definitions, as the editor is able to validate the edited content. Most editors support code completion when a valid DTD is available.”

In most cases, your XML declaration will match the one in the sample above.

4.5 The Root Element

4.5.1 Overview

The *root element* in the report template file is the `<report>` element. The complete report template definition is enclosed between the opening `<report>` tag and the closing `</report>` tag.

4.5.2 Attributes

The `<report>` element has a number of attributes:

Attribute:	**Description:**
`name`	The report name.
`pageformat`	The page format (`LETTER` and `A4` are common settings—see the `PageFormatFactory` class, or the DTD, for the complete list of formats).
`orientation`	The orientation (`portrait` or `landscape`).
`leftmargin`	The left margin (in points).
`rightmargin`	The right margin (in points).
`topmargin`	The top margin (in points).
`bottommargin`	The bottom margin (in points).

The report `name` provides a mechanism for naming your report. This is used occasionally—the report name is available as a report property (described later), and it can be added as a document property when you save reports to certain formats (such as PDF).

4.5.3 Subelements

The report template is defined by a list of subelements that appear within the `<report>` element:

Element:	**Description:**
`<configuration>`	Configuration settings for the report (see the `ReportConfiguration` class).
`<functions>`	The report functions.
`<reportheader>`	A report header.
`<reportfooter>`	A report footer.
`<pageheader>`	A page header.
`<pagefooter>`	A page footer.
`<groups>`	The report groups (at least one must be defined).
`<items>`	The report item band (repeated once for each row of data).

These subelements are described in the sections that follow. Before we continue, let’s review the overall structure of the report template file:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<report name="My Report" pageformat="LETTER" orientation="portrait">

    <configuration>
        <!-- report configuration specified here (optional) -->
    </configuration>

    <functions>
        <!-- report functions declared here -->
    </functions>

    <reportheader>
        <!-- report header defined here -->
    </reportheader>

    <reportfooter>
        <!-- report footer defined here -->
    </reportfooter>

    <pageheader>
        <!-- page header defined here -->
    </pageheader>

    <pagefooter>
        <!-- page footer defined here -->
    </pageheader>

    <groups>
        <!-- groups defined here -->
    </groups>

    <items>
        <!-- item band defined here -->
    </items>

</report>
```

Each of these subelements (`<reportheader>`, `<reportfooter>` etc.) can, in turn, contain it’s own subelements. In this way, the report template is built up from a hierarchy of elements, starting with the root `<report>` element.

4.6 Report Configuration

4.6.1 Overview

The report configuration section allows you to control certain aspects of a report’s behaviour. Refer to the documentation for the `ReportConfiguration` class to see which properties of the report you can control.

4.6.2 Example

Here is an example of the report configuration element:

```
<configuration>
    <property name="com.jrefinery.report.preview.PreferredWidth">640</property>
    <property name="com.jrefinery.report.preview.PreferredHeight">480</property>
</configuration>
```

4.7 Report Functions

4.7.1 Overview

Report functions allow you to calculate values that can be displayed in your report. A number of standard functions are available, and JFreeReport has been designed so that you can add your own report functions.

4.7.2 Declaring Functions

To display the value of a function in a report, you need to first “declare” the function, so that the report generator knows that it needs to calculate the function value.

4.7.3 The Functions Element

Functions are declared within the `<functions>` element. You can declare any number of functions:

```
<functions>

    <!-- function declarations go here -->

</functions>
```

Each function declaration is represented by a `<function>` subelement.

4.7.4 The Function Element

The `<function>` element declares a particular function for use in a report. You need to assign a `name` to the function instance, and also specify the Java class that implements the function:

```
<function name="MyFunction" class="com.jrefinery.report.function.PageFunction">

    <!-- function properties (if any) go here -->

</function>
```

The function name is important because it is used by report items used to display the function value in a report band.

The function may require a set of properties. Function properties are defined using the `<properties>` subelement.

4.7.5 The Properties Element

The `<properties>` element defines a collection of properties for a function:

```
<properties>

    <!-- properties go here -->

</properties>
```

Some functions don't require any properties, for example the `PageCount` function. Those that do require properties to be defined, will expect to find all the required properties defined within this element.

4.7.6 The Property Element

The `<property>` element defines a single property:

```
<property name="field">value</properties>
```

The property value is the character data between the tags. The property `name` is used to identify the property. For example, the `ItemSum` function looks for a property with the name `field` and uses this property to determine which column of the `TableModel` to sum.

4.7.7 Some Examples of Function Declarations

In the preceding sections, a lot of new tags were introduced. To show how they all fit together, here are some sample function declarations:

```
<functions>

    <function name="PageNumber" class="com.jrefinery.report.function.PageFunction"/>

    <function name="Count1" class="com.jrefinery.report.function.ItemCountFunction">
        <properties>
            <property name="group">Color Group</property>
        </properties>
    </function>

    <function name="Count2" class="com.jrefinery.report.function.ItemCountFunction">
        <properties>
            <property name="group">Letter Group</property>
</properties>
    </function>

    <function name="Sum1" class="com.jrefinery.report.function.ItemSumFunction">
        <properties>
            <property name="field">Double</property>
            <property name="group">Color Group</property>
        </properties>
    </function>
    <function name="Sum2" class="com.jrefinery.report.function.ItemSumFunction">
        <properties>
            <property name="field">Double</property>
            <property name="group">Letter Group</property>
        </properties>
    </function>

</functions>
```

4.8 Report Bands

4.8.1 Overview

A *report band* is a horizontal section of a report, usually spanning the complete width of the report. Each band contains a collection of *report items* (see section 4.11), which are used to display the text and graphics that make up the report.

In JFreeReport, there are seven different types of report band:

Band:	**Description:**
Report Header	Printed once only at the start of the report.
Report Footer	Printed once only at the end of the report.
Page Header	Printed at the top of every page.
Page Footer	Printed at the bottom of every page.
Group Header	Printed at the start of every report group.
Group Footer	Printed at the end of every report group.
Items	Printed once for every row of data.

These bands are similar in that you can add any report elements to each band. What differs between the bands is when and how often they are printed.

4.8.2 Attributes

Each report band has a set of attributes. All the report bands share these attributes in common:

Attribute:	**Description:**
`height`	The *minimum* height of the band (in points).
`fontname`	The default font for the band.
`fontstyle`	The default font style for the band (deprecated).
`fontsize`	The default font size for the band.
`fsbold`	Use bold font?
`fsitalic`	Use italic font?
`fsunderline`	Use underlined font?
`fsstrikethr`	Use strikethrough font?

The `height` attribute controls the minimum amount of space used to print the band. It is defined in points (1/72 inch).

Important note: a band containing no elements and with `height` = 0 is considered empty and is not printed. A band containing no elements and `height` > 0 is NOT considered empty, the band is output (as white-space) at the specified height.

The font attributes provide default values for the report items contained within the band. For the font name, you can use any font that is supported on your system, but for portability it is recommended that you use the generic font names:

Generic Font Name:	**Description:**
`serif`	A serif font (e.g. Times New Roman).
`sansserif`	A sans-serif font (e.g. Arial or Helvetica).
`monospaced`	A monospaced font (e.g. Courier).

The `fontstyle` attribute is now deprecated, you should use the other font attributes to control the font style. If `fontstyle` is used, the value should be one of: `plain`, `bold`, `italic` or `bold+italic`.

4.8.3 The Report Header

The report header is printed once at the start of the report. It is defined using the `<reportheader>` tags as follows:

```
<reportheader height="48" fontname="serif" fontsize="18" fsbold="true">
    <!-- insert report elements here -->
</reportheader>
```

Inside these tags, you can add any report items (see section 4.11) you choose.

In addition to the height and font attributes, the report header has an `ownpage` attribute that controls whether or not the header is printed using a page of its own. The default value is `false`, you might want to set it to `true` if you want a title page for your report. For example:

```
<reportheader height="48" fontname="serif" fontsize="18" fsbold="true"
    ownpage="true">
    <!-- insert report elements here -->
</reportheader>
```

You can include as many report items as you want to within the `<reportheader>` element, just as you can for any other report band. The report items are described in section 4.11.

4.8.4 The Report Footer

The report footer is printed once at the end of the report. It is defined using the `<reportfooter>` tags as follows:

```
<reportfooter height="48" fontname="serif" fontsize="18" fsbold="true">
    <!-- insert report elements here -->
</reportfooter>
```

Inside these tags, you can add any report items (see section 4.11) you choose.

In addition to the height and font attributes, the report footer has an `ownpage` attribute that controls whether or not the footer is printed using a page of its own. The default value is `false`, you might want to set it to `true` if you want a closing page for your report. For example:

```
<reportfooter height="48" fontname="serif" fontsize="18" fsbold="true"
    ownpage="true">
    <!-- insert report elements here -->
</reportfooter>
```

You can include as many report items as you want to within the `<reportfooter>` element, just as you can for any other report band. The report items are described in section 4.11.

4.8.5 The Page Header

The page header is printed once at the top of every page. It is defined using the `<pageheader>` tags, as follows:

```
<pageheader height="24" fontname="serif" fontsize="12" fsbold="true">
    <!-- insert report elements here -->
</pageheader>
```

Inside these tags, you can add any report items (see section 4.11) you choose.

In addition to the height and font attributes, the page header has an `onfirstpage` attribute that controls whether or not the header is printed on the first page, and an `onlastpage` attribute that controls whether or not the header is printed on the last page. The default value for both attributes is `false`, since there is often a report header on the first page and a report footer on the last page. You can set either attribute (or both) to `true` like this:

```
<pageheader height="24" fontname="serif" fontsize="12"
    fsbold="true" onfirstpage="true" onlastpage="true">
    <!-- insert report elements here -->
</pageheader>
```

You can include as many report items as you want to within the `<pageheader>` element, just as you can for any other report band. The report items are described in section 4.11.

4.8.6 The Page Footer

The page footer is printed once at the bottom of each page. It is defined using the `<pagefooter>` tags as follows:

```
<pagefooter height="24" fontname="serif" fontsize="12" fsbold="true"
    onfirstpage="true">
    <!-- insert report elements here -->
</pagefooter>
```

Inside these tags, you can add any report items (see section 4.11) you choose.

In addition to the height and font attributes, the page footer has an `onfirstpage` attribute that controls whether or not the footer is printed on the first page, and an `onlastpage` attribute that controls whether or not the footer is printed on the last page. The default value for both attributes is `false`, since there is often a report header on the first page and a report footer on the last page.You can set either attribute (or both) to `true` like this:

```
<pagefooter height="24" fontname="serif" fontsize="12"
    fsbold="true" onfirstpage="true" onlastpage="true">
    <!-- insert report elements here -->
</pagefooter>
```

You can include as many report items as you want to within the `<pagefooter>` element, just as you can for any other report band. The report items are described in section 4.11.

4.8.7 Group Headers and Footers

Group headers and footers are very similar to page headers and footers except that, as the name suggests, they are printed before and after report groups. More information about report groups is provided in section 4.9.

4.9 Report Groups

4.9.1 Overview

JFreeReport can group data in a report, based on the values in one or more columns of the `TableModel`. Whenever the values change between one row of data and the next, a new group is started. Groups are useful because they subdivide your data, and allow you to apply report functions to subsets of the data.

In this section, we focus on how to define groups in the XML report definition. For more of an overview of how groups work, refer to section 6.

4.9.2 The Groups Element

The `<groups>` element is used to define the groups for a report, if there are any. The basic structure is shown here:

```
<groups>

    <group name="Group 1">
        <!-- group definition goes here -->
    </group>

    <group name="Group 2">
        <!-- group definition goes here -->
    </group>

</groups>
```

When you define multiple groups, you end up with nested subgroups in your report.

4.9.3 The Group Element

The `<group>` element defines the fields used to group data, plus the (optional) group header and footer bands. The structure is shown here:

```
<group name="Group 1">

    <groupheader height="18">
        <!-- report elements appear here -->
    </groupheader>

    <groupfooter height="18">
        <!-- report elements appear here -->
    </groupfooter>

    <fields>
        <!-- one or more field elements appear here -->
    </fields>

</group>
```

The group header (if defined) is printed once *every time a new instance of a group starts*. You can use the group header to provide labels for each group in your report.

Likewise, the group footer (if defined) is printed once at the end of every group instance. The group footer is a convenient place to display summary data for a group (item counts, totals and so forth).

4.9.4 The Group Header Element

The report items that should appear in the group header are defined within the `<GroupHeader>` element. The group header is a *report band*, and should be defined in a similar way to the report and page header bands.

4.9.5 The Group Footer Element

The report items that should appear in the group footer are defined within the `<GroupFooter>` element. The group footer is a *report band*, and should be defined in a similar way to the report and page footer bands.

4.9.6 The Fields Element

The `<fields>` element is used to specify the fields that define the group. Each field is defined using a `<field>` element, for example:

```
<fields>
    <field>Column A</field>
    <field>Column B</field>
    <field>Column C</field>
</fields>
```

The name inside the `<field>` tag should correspond to a column name from the report’s `TableModel`.

4.10 The Item Band

The item band is a very important report band, because it is printed once per row *for each row of data in the report’s dataset.* It is defined using the `<items>` element as follows:

```
<items height="12" fontname="serif" fontsize="10">
    <!-- insert report elements here -->
</pagefooter>
```

The item band defines no special attributes.

You can include as many report items as you want within the `<items>` element, just as you can for any other report band. The report items are described in section 4.11.

4.11 Report Items

4.11.1 Overview

A *report item* is used to display text or a graphical element within a report band. The following table lists the types of report items supported by JFreeReport:

Item:	**Description:**
Label	A static text item.
String Field	A report item that displays a `String` from the report's data row.
Number Field	A text item that displays a `Number` from the report's data row, with formatting.
Date Field	A text item that displays a `Date` from the report's data row, with formatting.
Image Field	A graphical item that displays an `Image` from the report's data row.
Image URL Field	A graphical item that displays an `Image` sourced from a URL obtained from the report's data row.
Line	A line item.
Rectangle	A rectangle item.
Image Reference	A report item that displays an `Image` sourced from a static URL.
Resource Label	??
Resource Field	??

Report items that access a “field” from the report's data row can obtain their data from:

- the current row of the report's `TableModel`;
- a report function or expression;
- a report property;

As mentioned previously, each report band can contain any number of report items.

4.11.2 Position Attributes

Most report items use the following attributes to specify the bounds for the item within its report band:

Attribute:	**Description:**
x	The horizontal position of the top left corner of the item (relative to its band).
y	The vertical position of the top left corner of the item (relative to its band).
width	The width of the item.
height	The height of the item.

All of these dimensions are most commonly specified in points (1/72 inch). However, you can also specify them as percentages, in which case the values are calculated relative to the size of the report band that the item belongs to.

4.11.3 Font Attributes

All report items that display text have the following font attributes:

Attribute:	**Description:**
fontname	The font name.
fontsize	The font size.
fontstyle	The font style (deprecated).
fsbold	Use a bold font?
fsitalic	Use an italic font?
fsunderline	Use an underlined font.
fsstrikethr	Use a strikethrough font.

All of the font attributes are optional—if they are omitted, the default font from the report band is used.

For the font name, you can use the name of any font installed on your system. However, for portability, it is recommended that you use one of the Java logical font names: `Serif`, `SansSerif` or `Monospaced`.

4.11.4 Labels

A *label* displays static text. Labels can be added to any report band.

A label is defined using a `<label>` element, with the label text being the character data between the tags. Label attributes are listed in the following table:

Attribute:	**Description:**
name	The name of the report item (optional).
x	The horizontal position of the top left corner of the item (relative to its band).
y	The vertical position of the top left corner of the item (relative to its band).
width	The width of the item.
height	The height of the item.
color	The foreground color of the label.
line-height	The line-height.
alignment	The horizontal alignment.
vertical-alignment	The vertical alignment.
dynamic	Expand the height of the report item to fit the content?

In addition, the standard font attributes (see section 4.11.3) are available.

Here is an example:

```
<label name="label1"
       x="0" y="0" width="144" height="20"
       alignment="left">Total:</label>
```

You should look through some of the sample report definition files for other examples.

4.11.5 String Fields

A *string field* is a report item that displays the value of a `String` object from a particular column in the the current row of the report's dataset.

A string field is defined using the `<string-field>` element. The usual font attributes are available (see section 4.11.3).

The `fieldname` attribute specifies the name of the column in the dataset from which this report item gets its value at report generation time. When defining a string field, you need to ensure that this matches the name returned by the `getColumnName(...)` method in your `TableModel`.

A typical string field item definition looks like this:

```
<string-field name="field2" fieldname="Name"
              x="0" y="0" width="72" height="20"
              alignment="left"></string-field>
```

You should look through some of the sample report templates for other examples.

4.11.6 Number Fields

A *number field* is a report item that displays the value of a `Number` object from a particular column in the the current row of the report's dataset.

A number field is defined using the `<number-field>` element. The usual font attributes are available (see section 4.11.3). In addition, you can specify a `format` for the number. Internally, the formatting is performed by a `DecimalFormat` object.

The `fieldname` attribute specifies the name of the column in the dataset from which this report item gets its value at report generation time. When defining a number field, you need to ensure that this matches the name returned by the `getColumnName(...)` method in your `TableModel`.

Here is an example:

```
<number-field name="int1" fieldname="Integer"
              x="300" y="0" width="76" height="8"
              alignment="right" format="#,##0"></number-field>
```

You should look through some of the sample report templates for other examples.

4.11.7 Date Fields

A *date field* is a report item that displays the value of a `Date` object from a particular column in the the current row of the report’s dataset.

A date field is defined using the `<date-field>` element. The usual text attributes are available (see section **??**). In addition, you can specify a `format` for the date. Internally, the formatting is performed by a `SimpleDateFormat` object.

The `fieldname` attribute the name of the column in the dataset from which this report item gets its value at report generation time. When defining a date field, you need to ensure that this matches the name returned by the `getColumnName(...)` method in your `TableModel`.

Here is an example:

```
<date-field name="date1" fieldname="Date"
            x="300" y="0" width="76" height="8"
            alignment="right" format="d-MMM-yyyy"></date-field>
```

You should look through some of the sample report templates for other examples.

4.11.8 Resource Labels

Not yet documented.

4.11.9 Resource Fields

Not yet documented.

4.11.10 Image Fields

An *image field* is a report item that displays an image, where the image is sourced from:

- a cell in the current row of the report’s `TableModel`;
- a function;
- a report property;

The following table lists the attributes for an image field:

Attribute:	**Description:**
name	The name of the report item (optional).
x	The horizontal position of the top left corner of the item (relative to its band).
y	The vertical position of the top left corner of the item (relative to its band).
width	The width of the item.
height	The height of the item.
fieldname	The image source.
dynamic	Expand the height of the report item to fit the content?
scale	Scale the image to fit the size of the report item?
keepAspectRatio	Preserve the image aspect ratio when scaling?

Here is an example:

```
<image-field x="0" y="0" width="100" height="100" fieldname="logo" />
```

4.11.11 Image URL Fields

An *image URL field* is a report item that displays an image that is located at a given URL, where the URL is sourced from:

- a cell in the current row of the report's `TableModel`;
- a function;
- a report property;

The following table lists the attributes for an image URL field:

Attribute:	**Description:**
name	The name of the report item (optional).
x	The horizontal position of the top left corner of the item (relative to its band).
y	The vertical position of the top left corner of the item (relative to its band).
width	The width of the item.
height	The height of the item.
fieldname	The image URL source.
dynamic	Expand the height of the report item to fit the content?
scale	Scale the image to fit the size of the report item?
keepAspectRatio	Preserve the image aspect ratio when scaling?

4.11.12 Image References

An *image reference* is a report item that displays an image from a static URL.

The following table lists the attributes for an image reference:

Attribute:	**Description:**
name	The name of the report item (optional).
x	The horizontal position of the top left corner of the item (relative to its band).
y	The vertical position of the top left corner of the item (relative to its band).
width	The width of the item.
height	The height of the item.
src	The image source.
dynamic	Expand the height of the report item to fit the content?
scale	Scale the image to fit the size of the report item?
keepAspectRatio	Preserve the image aspect ratio when scaling?

Here is an example (from the `report3.xml` file in the JFreeReport distribution):

```
<imageref name="logo" x="200" y="48" width="100" height="90"
        src="anim0002.wmf" dynamic="true"/>
```

4.11.13 Lines

A *line item* draws a line within a report band. Lines are defined using the `<line>` element. The following attributes can be defined:

Attribute:	**Description:**
name	The item name.
x1	The x-coordinate for the line's starting point.
y1	The y-coordinate for the line's starting point.
x2	The x-coordinate for the line's ending point.
y2	The y-coordinate for the line's ending point.
color	The line color.
weight	The line weight.

The coordinates are all relative to the report band that the line item belongs to. Y values increase as you move *down* the page.

One special case is handled—if the starting point *(x1, y1)* is the same as the ending point *(x2, y2)*, then a horizontal line is drawn across the entire width of the page, passing through the specified point (within the current band).

Here is an example:

```
<line name="line2" x1="0" y1="4" x2="0" y2="4" color="#CCCCCC" weight="0.5" />
```

4.11.14 Rectangles

A *rectangle item* draws a rectangle within a report band. Rectangles are defined using the `<rectangle>` element. The following attributes can be defined:

Attribute:	**Description:**
name	The item name.
x	The horizontal position of the top-left corner of the rectangle (relative to its band).
y	The vertical position of the top-left corner of the rectangle (relative to its band).
width	The width of the rectangle (in points).
height	The height of the rectangle (in points).
draw	Draw an outline around the rectangle?
fill	Fill the interior of the rectangle?
color	The color used for drawing and filling the rectangle.
weight	The line weight for the rectangle's outline (if any).

The *x*, *y*, *width* and *height* attributes are measured in points (1/72 inch). Here is an example that draws a rectangle at the top-left corner of a band:

```
<rectangle x="0" y="0" width="200" height="75"
           color="#CCFFCC" draw="true" fill="false"/>
```

The dimensions can also be specified as percentage values relative to the current size of the report band that the rectangle belongs to. For example, this rectangle will fill the lower right quadrant of its band:

```
<rectangle x="50%" y="50%" width="50%" height="50%"
           color="red" draw="true" fill="true"/>
```

5 The "Extended" Report Definition Format

5.1 Introduction

This section documents the "extended" report definition format.

5.2 The Report Definition

This is an XML-based format, with a complete report definition contained within a `<report-definition>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE report PUBLIC "-//JFreeReport//DTD report definition//EN//extended"
                         "http://jfreereport.sourceforge.net/extreport.dtd">

<report-definition name="Report 1">

  <!-- sub elements appear here -->

</report-definition>
```

Given a valid report definition file, you can use the `ReportGenerator` class to recreate a `JFreeReport` instance.

5.3 The Report Definition Element

Inside the `<report-definition>` element, a number of sub-elements are used to define the report. In skeleton form, the extended report definition format looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE report PUBLIC "-//JFreeReport//DTD report definition//EN//extended"
                         "http://jfreereport.sourceforge.net/extreport.dtd">

<report-definition name="Report 1">

  <parser-config>
    <!-- INSERT PARSER CONFIGURATION HERE -->
  </parser-config>

  <report-config>
    <!-- INSERT REPORT CONFIGURATION HERE -->
  </report-config>

  <styles>
    <!-- INSERT STYLES HERE -->
  </styles>

  <templates>
    <!-- INSERT TEMPLATES HERE -->
  </templates>

  <functions>
    <!-- INSERT FUNCTIONS HERE -->
  </functions>

  <report-description>
    <!-- INSERT REPORT HEADER HERE -->
    <!-- INSERT REPORT FOOTER HERE -->
    <!-- INSERT PAGE HEADER HERE -->
    <!-- INSERT PAGE FOOTER HERE -->
    <!-- INSERT GROUP DEFINITIONS HERE -->
    <!-- INSERT ITEM BAND HERE -->
  </report-description>

</report-definition>
```

These subelements are described in the following sections:

Element:	**Refer to:**
`<parser-config>`	Section 5.4
`<report-config>`	Section 5.5

5.4 The Parser Configuration

5.4.1 Overview

This section is used to configure the parser, by specifying the class names for the factory objects used during parsing.

5.4.2 Example

A typical parser configuration looks like this:

```
<!-- *********************** -->
<!-- * PARSER CONFIGURATION * -->
<!-- *********************** -->
<parser-config>
  <object-factory class="com.jrefinery.report.io.ext.factory.datasource.DataSourceCollector"/>
  <object-factory class="com.jrefinery.report.io.ext.factory.objects.DefaultClassFactory"/>
  <element-factory class="com.jrefinery.report.io.ext.factory.elements.DefaultElementFactory"/>
  <stylekey-factory class="com.jrefinery.report.io.ext.factory.stylekey.DefaultStyleKeyFactory"/>
  <stylekey-factory class="com.jrefinery.report.io.ext.factory.stylekey.PageableLayoutStyleKeyFactory"/>
  <template-factory class="com.jrefinery.report.io.ext.factory.templates.DefaultTemplateCollection"/>
  <datasource-factory class="com.jrefinery.report.io.ext.factory.datasource.DefaultDataSourceFactory"/>
</parser-config>
```

5.4.3 Notes

The `ParserConfigHandler` class is used by the parser when reading the parser configuration.

The `ParserConfigWriter` class is used to write the parser configuration.

5.5 The Report Configuration

A typical configuration looks like this:

```
<!-- ************************ -->
<!-- * REPORT CONFIGURATION * -->
<!-- ************************ -->
<report-config>
  <defaultpageformat orientation="portrait"
                     pageformat="LETTER"
                     topmargin="72"
                     bottommargin="72"
                     leftmargin="72"
                     rightmargin="72"/>
  <configuration>
    <property name="com.jrefinery.report.preview.PreferredHeight">480</property>
    <property name="com.jrefinery.report.preview.PreferredWidth">640</property>
  </configuration>
</report-config>
```

5.6 Styles

5.7 Templates

5.8 Functions

5.9 Data Definition

5.10 Report Definition

6 Groups

6.1 Introduction

Much of JFreeReport's descriptive power comes from its ability to *group rows of data* within a report. The reporting engine can count, sum, average and perform other calculations on the values in particular fields in a `TableModel`. By calculating and displaying these values for groups and sub-groups within a report, JFreeReport can assist you to evaluate and understand your data more easily.

As a report designer, to make the most of JFreeReport you need to understand how grouping works. This section illustrates the concepts by presenting a relatively simple example.

6.2 Sample Data

Consider the following table, which contains data that could easily be used to create a report using JFreeReport:

Color:	**Code:**	**Units:**
Red	A	7
Red	B	4
Red	B	9
Green	C	8
Green	C	2
Blue	A	1
Blue	A	7
Blue	B	4
Blue	C	5

Notice that there are nine rows in the table, and that the rows are sorted, first "by color", and then "by code".

In the following sections, several grouping options are illustrated using this sample data.

6.3 Basic Grouping

With the items in the table sorted by color, a fairly obvious division can be seen—the first three rows contain red items, the next two rows contain green items, and the last four rows contain blue items. This splits the table into three sections or "groups".

Here is group 1 (the red items):

Color:	**Code:**	**Units:**
Red	A	7
Red	B	4
Red	B	9

...and group 2 (the green items):

Color:	**Code:**	**Units:**
Green	C	8
Green	C	2

...and group 3 (the blue items):

Color:	**Code:**	**Units:**
Blue	A	1
Blue	A	7
Blue	B	4
Blue	C	5

These three groups are created by looking at the value in the `Color` column for each item in the table—you might refer to this as "grouping by color". The column name is all that is required to define the grouping.

JFreeReport allows you to define a group by specifying the column on which to sub-divide the data rows.

6.4 Grouping On More Than One Column

In the previous section, a single column was used to define a grouping. However, it is also possible to group data by specifying two or more columns. The groups are created in much the same way, except that items are grouped together when the values in ALL the columns specified for the group are the same.

For example, suppose we grouped the sample data "by color and code"—this would result in six groups:

Color:	**Code:**	**Units:**
Red	A	7

Color:	**Code:**	**Units:**
Red	B	4
Red	B	9

Color:	**Code:**	**Units:**
Green	C	8
Green	C	2

Color:	**Code:**	**Units:**
Blue	A	1
Blue	A	7

Color:	**Code:**	**Units:**
Blue	B	4

Color:	**Code:**	**Units:**
Blue	C	5

Notice that:

- the order of the data rows has not changed—they appear in the same order as in the original table.
- *within each group* both the color and code values do not change (by definition);

JFreeReport allows you to define a grouping by specifying the *columns* on which to form the groups.

6.5 Subgroups

We've seen that it is possible to define more than one grouping to a given set of data. A group can be defined in terms of one column, or multiple columns.

It is also possible to define a group that is a *subgroup* of another group (called the *parent* group). This is just a special case where the columns defined for the subgroup are a super-set of the columns defined for the parent group (that is, all the columns of the parent group are included in the subgroup).

Our previous example, the group defined for the "color" and "code" columns, is a subgroup of the "by color" group, because it includes all the columns of the parent group (just `Color`).

Subgroups are used to sub-divide groups. If you look closely at the groups in the previous section, you will see that the original "by color" grouping has been further sub-divided "by code".

The red group has been sub-divided into two groups, one for code "A" and one for code "B":

Color:	**Code:**	**Units:**
Red	A	7

Color:	**Code:**	**Units:**
Red	B	4
Red	B	9

The green group is not subdivided, since only one code ("C") occurs for this group:

Color:	**Code:**	**Units:**
Green	C	8
Green	C	2

The final group is subdivided into three sub-groups:

Color:	**Code:**	**Units:**
Blue	A	1
Blue	A	7

Color:	**Code:**	**Units:**
Blue	B	4

Color:	**Code:**	**Units:**
Blue	C	5

6.6 Row Order and Sorting

When you define groups and sub-groups for your report, it is important that the data is ordered in a way that is consistent with your group definitions. To see why this is the case, let's group our sample data "by code" without sorting it into the correct order first:

Color:	**Code:**	**Units:**
Red	A	7

Color:	**Code:**	**Units:**
Red	B	4
Red	B	9

Color:	**Code:**	**Units:**
Green	C	8
Green	C	2

Color:	**Code:**	**Units:**
Blue	A	1
Blue	A	7

Color:	**Code:**	**Units:**
Blue	B	4

Color:	**Code:**	**Units:**
Blue	C	5

Although we might expect all of the code "A" rows to appear in one group, this hasn't happened because the data is still sorted "by color".

If we re-order the data:

Color:	**Code:**	**Units:**
Red	A	7
Blue	A	1
Blue	A	7
Red	B	4
Red	B	9
Blue	B	4
Blue	C	5
Green	C	8
Green	C	2

...and then group, we have the "A" group:

Color:	**Code:**	**Units:**
Red	A	7
Blue	A	1
Blue	A	7

...and the "B" group:

Color:	**Code:**	**Units:**
Red	B	4
Red	B	9
Blue	B	4

...and the "C" group:

Color:	**Code:**	**Units:**
Blue	C	5
Green	C	8
Green	C	2

6.7 Why Doesn't JFreeReport Sort The Data?

A common question that JFreeReport developers ask is "Why doesn't JFreeReport sort the data?". This is a valid question, since most report generators do perform this function.

The reason that JFreeReport doesn't attempt to sort data is primarily because the `TableModel` interface sits between JFreeReport and your data. JFreeReport has no knowledge of the implementation that lies behind the `TableModel` interface.

This means that the only sorting strategy that would work for the general case is to copy the data from the `TableModel` into some local structure and sort it independently of the source data. This would be a wasteful strategy, so we currently pass responsibility for providing the data in sorted order on to the developer using the JFreeReport library.

7 Functions and Expressions

7.1 Introduction

Report functions and expressions are used to calculate values that can be displayed in a report. Expressions do not maintain any state information during report processing, so they are used mainly for calculating values within the current row. Functions, on the other hand, can record state information as the report processing proceeds, so can be used for calculating field totals, keeping page counts and so on.

In this section, examples are presented for many of the standard functions and expressions included with JFreeReport.

7.2 Standard Expressions

Two standard expressions are included with the current version of JFreeReport. These are located in the `com.jrefinery.report.function` package:

- `BSHExpression` - an expression that allows you to take advantage of the BeanShell scripting engine to write your own "scripted" expressions (see section 7.8);
- `TextFormatExpression` - an expression that allows you to format one or more items using Java's `MessageFormat` class (an example is given in section 7.5.3).

It is likely that additional standard expressions will be added in future versions of JFreeReport.

7.3 Standard Functions

A range of standard functions is included with the current version of JFreeReport. These are located in the `com.jrefinery.report.function` package:

- `ElementVisibilitySwitchFunction` - a function that alternates between `true` and `false` for each row of a report.
- `EventMonitorFunction` - this function is used for debugging purposes, it logs all event notifications it receives from the reporting engine.
- `GroupCountFunction` - counts the number of groups in a report (or subgroups in a group).
- `ItemAvgFunction` - calculates the average of one field in a report or group.
- `ItemColumnQuotientFunction` - calculates the quotient of two values in the same row of a report.

- `ItemCountFunction` - counts the number of items (rows) in a report or group (see section 7.6).
- `ItemHideFunction` - hides repeating values of a field in a report.
- `ItemMaxFunction` - tracks the maximum value of one field in a report or group.
- `ItemMinFunction` - tracks the minimum value of one field in a report or group.
- `ItemPercentageFunction` - calculates the percentage of the value in a field relative to the field total for the report or group.
- `ItemSumFunction` - calculates the sum of one field in a report or group (see section 7.7).
- `PageFunction` - returns the current page number (see section 7.5).

It is likely that additional standard functions will be added in future versions of JFreeReport.

7.4 Displaying the Report Date

7.4.1 Overview

This section describes how to display the current date in a report.

7.4.2 The Report Date Property

All reports define a standard property `report.date` which contains an instance of the `java.util.Date` class, created at the point that report processing begins.

7.4.3 Marking a Report Property

You need to "mark" a report property before it can be accessed by a report element. This tells the reporting engine that it should provide access, via the `DataRow`, to the report property.

Marking a report property just involves referencing the property name in a `<property-ref>` element inside the `<functions>` element of the report definition:

```
<!-- The report date. -->
<property-ref name="report.date"/>
```

7.4.4 Displaying the Report Date

To display the report date in a report element (which you can add to any report band), use a *date field*:

```
<date-field fieldname="report.date"
            format="d-MMM-yyyy"
            x="0" y="0" width="80" height="12"
            alignment="center"
            vertical-alignment="left"/>
```

Notice that:

- the `fieldname` attribute references (by name) the report property that we "marked" previously;
- the `format` attribute controls the formatting of the date (internally, this string is passed to a `SimpleDateFormat` instance to perform the formatting);

The `report.date` property is initialised at the moment JFreeReport begins processing a report.

7.5 Displaying Page Numbers

7.5.1 Overview

This section describes techniques for displaying page number information in reports.

7.5.2 Simple Page Numbers

To display the *current* page number in a report, you can use the `PageFunction` class. In your report template file, add the following function declaration:

```
<!-- The current page number. -->
<function name="MyPageFunction1"
          class="com.jrefinery.report.function.PageFunction"/>
```

Now, having declared this function with the name `MyPageFunction1` (you are free to choose another name if you wish), you can display the page number in any report band (the page header or footer is the most common location) using a *number field*:

```
<number-field fieldname="MyPageFunction1"
              format="Page 0"
              x="0" y="0" width="100%" height="100%"
              alignment="center"
              vertical-alignment="middle"/>
```

Notice how the `fieldname` attribute is used to reference the function we declared earlier—the function name (`MyPageFunction1`) is used for the reference. Formatting is applied via the `format` attribute, which in the example prepends the page number with the text “Page”. Internally, JFreeReport passes the format string to a `NumberFormat` instance, so you should refer to the Javadocs for `NumberFormat` for more information about the possible formatting options.

If you would like page numbering to start at some number other than “1”, you can add a parameter (property) to the function:

```
<!-- The current page number. -->
<function name="MyPageFunction1"
          class="com.jrefinery.report.function.PageFunction">
  <properties>
    </property name="start">42</property>
  </properties>
</function>
```

The example above will cause the page numbering to start at “42”.

7.5.3 Page N of M

It is possible to display both the current page number and the *total number of pages* in a report, by combining a `PageFunction` and a `PageTotalFunction` using a `TextFormatExpression`.

First, you need to include the `MyPageFunction1` function declaration, described in the previous section, in your report template file.

Second, you need to declare a `PageTotalFunction` to return the total number of pages for the report:

```
<!-- The page count. -->
<function name="MyPageCount"
          class="com.jrefinery.report.function.PageTotalFunction">
</function>
```

Finally, you need to declare a `TextFormatExpression` to combine the page number and page total into a string of the form “Page N of M”, ready for display in the report.

Here are the required declarations:

```
<!-- The current page number. -->
<function name="MyPageNumber"
          class="com.jrefinery.report.function.PageFunction"/>

<!-- The total page count. -->
<function name="MyPageCount"
          class="com.jrefinery.report.function.PageTotalFunction">
</function>

<!-- A 'Page <n> of <m>' string. -->
<expression name="MyPageNofM"
            class="com.jrefinery.report.function.TextFormatExpression">
  <properties>
    <property name="pattern">Page {0,number,integer} of {1,number,integer}</property>
    <property name="0">MyPageNumber</property>
    <property name="1">MyPageCount</property>
  </properties>
</expression>
```

Notice how the `MyPageNumber` and `MyPageCount` functions are referenced using properties in the `MyPageNofM` expression above. The property names (`0` and `1`) are important, they correspond to the position of the arguments in the text format expression.

With these functions declared, you can now display the page numbering information in any band in your report:

```
<string-field fieldname="MyPageNofM"
              x="0" y="3" width="100%" height="9"
              alignment="right"/>
```

Notice that the function result is a `String` this time, so it is displayed in a *string field* on the report. As before, the `fieldname` attribute is used to reference the function that supplies the page numbering information.

7.6 Counting Items

7.6.1 Overview

It is often useful to display an *item count* in a report. JFreeReport can display:

- a running count for a report or report group;
- a total count for a report or report group;

This section describes how to add item count information to your own reports.

7.6.2 Declaring an Item Count Function

To display an item count within a report, declare an `ItemCountFunction` in your report template. For example:

```
<function name="MyItemCountFunction1"
          class="com.jrefinery.report.function.ItemCountFunction"/>
```

This function counts each row of data (from the report's `TableModel`) as the report is being processed.

If you want the count to reset to zero at the start of a particular group, you can add a property that references the required group name:

```
<function name="MyItemCountFunction2"
          class="com.jrefinery.report.function.ItemCountFunction">
  <properties>
    <property name="group">MyGroup1</property>
  </properties>
</function>
```

You can declare multiple instances of the `ItemCountFunction`, so it is no problem to include item counts for one or more groups in your report as well as report totals.

7.6.3 Displaying the Item Count

To display the result of a item count function, you can add a *number field* to any band in your report. The number field should reference the function by name, for example:

```
<number-field fieldname="MyItemCountFunction1"
              format="0"
              x="300" y="7" width="168" height="10"
              alignment="right"/>
```

As each row of data in the report is processed, the count is incremented. If you display the function value in a report's *item band*, then you will see a *running count*.

Alternatively, if you are only interested in the *total count* for the report (or a group within the report), then you can display the item count in the report footer (or a group footer).

7.7 Summing Items

7.7.1 Overview

It is often useful to display an *item sum* in a report. JFreeReport can display:

- a running sum for a field in a report or report group;
- a total sum for a field in a report or report group;

This section describes how to add item sum information to your own reports.

7.7.2 Declaring an Item Sum Function

To display an item sum within a report, declare an `ItemSumFunction` in your report template. For example:

```
<function name="MyItemSumFunction1"
          class="com.jrefinery.report.function.ItemSumFunction">
  <properties>
    <property name="field">MyItem1</property>
  </properties>
</function>
```

This function sums the `MyItem1` field from each row of data (from the report's `TableModel`) as the report is being processed.

If you want the sum to reset to zero at the start of a particular group, you can add a property that references the required group name:

```
<function name="MyItemSumFunction1"
          class="com.jrefinery.report.function.ItemSumFunction">
  <properties>
    <property name="field">MyItem1</property>
    <property name="group">MyGroup1</property>
  </properties>
</function>
```

You can declare multiple instances of the `ItemSumFunction`, with different property settings, provided that each instance has a unique name.

7.7.3 Displaying the Item Sum

To display the result of a item sum function, you can add a *number field* to any band in your report. The number field should reference the function by name, for example:

```
<number-field fieldname="MyItemSumFunction1"
              format="#,##0"
              x="240" y="0" width="168" height="10"
              alignment="right"/>
```

As each row of data in the report is processed, the sum accumulates. If you display the function value in a report's *item band*, then you will see a *running total*.

Alternatively, if you are only interested in the *total sum* for the report (or a group within the report), then you can display the item sum in the report footer (or a group footer).

7.8 Expressions With BeanShell

7.8.1 Overview

The `BSHExpression` class allows you to develop custom expressions in JFreeReport without having to modify the JFreeReport source code. This section provides an example expression to illustrate the way that the `BSHExpression` class can be used to extend JFreeReport.

7.8.2 An Example

Suppose you have a `TableModel` where one or more columns contain numerical data that should be presented to end-users in text form. For example, imagine a column "Sex" that contains a "0" to indicate "male" and a "1" to indicate "female".[5]

Here is an expression declaration that can be used to convert such values into the strings `Male` and `Female`:

```
<!-- A sample BeanShell scripted expression. -->
<expression name="Gender"
            class="com.jrefinery.report.function.BSHExpression">
  <properties>
    <property name="expression">
      // return type Object is given...
      getValue() {
        Integer value = (Integer) dataRow.get("Sex");
        int v = value.intValue();
        if (v == 0) {
          return "Male";
        }
        else {
         return "Female";
        }
```

[5]This example is based on a question posted in the JFreeReport forum, and the solution posted by Thomas Morgner.

```
        }
      </property>
    </properties>
  </expression>
```

The `expression` property contains a small fragment of Java code that calculates a return value for the expression. This code fragment will be evaluated by the BeanShell scripting engine, and the result of the `getValue()` method (assumed to be an `Object` instance) is returned to JFreeReport as the expression value.

In the example, an integer value is obtained from the *Sex* column in the report's `TableModel` (you can access all items via the `dataRow.get(String)` method).

The expression is given the name `Gender` and can be referenced by any report element using this name. For example, in the item band you could add the following:

```
<string-field x="240" y="8" width="220" height="10" alignment="left"
fieldname="Gender" />
```

7.8.3 Performance

If you have expressions that you use frequently, or in more than one report, it is worthwhile considering implementing the expression as a Java class. BeanShell offers flexibility, but it is an interpreter so it will not evaluate expressions as quickly as a compiled Java class.

8 Miscellaneous Topics

8.1 Report Properties

8.1.1 Overview

A *report property* is an `Object` that is associated with a key (a `String`) and bound to a report. The `JFreeReport` class maintains a list of report properties, and allows you to add new properties as you require them. You can display the value of any report property in a report.

In this section, we present examples that show how to display custom report properties, including:

- a user name (or any other text);
- a company logo (or any other image);

To define a report property, you use the `setProperty(...)` method in the `JFreeReport` class.

8.1.2 Displaying the User's Name

To display the user's name in a report, you could define a custom report property that contains the user's name, then display that property in one of the report bands (for example, the page header).

The following code reads the current user name from the system properties, and adds it as a report property using the key `user.name`:

```
JFreeReport report = ...

// you can get the property value from anywhere you want...
String username = System.getProperty("user.name");
report.setProperty("user.name", username);

// mark the property so it appears in the DataRow...
report.setPropertyMarked("user.name", true);
```

The property is "marked" to make it available via the `DataRow`.

To display a report property in a report, you create an appropriate report element (for the property type) and add it to any report band. To display the `user.name` property defined in the previous section, you could use the following code:

```
TextElement t0 = ItemFactory.createStringElement(
    "T0",
    new Rectangle2D.Double(45.0, 0.0, 150.0, 20.0),
    Color.black,
    ElementAlignment.LEFT.getOldAlignment(),
    ElementAlignment.MIDDLE.getOldAlignment(),
    null, // font
    "-",  // null string
    "user.name"
);
report.getPageHeader().addElement(t0);
```

Notice how the property is referenced by name within the element definition.

8.1.3 Displaying a Company Logo

To display a company logo on your report, you can add the logo (or any other `java.awt.Image` as a report property, then display it in any band on your report using an image field element.

Here is some code that loads an image from a file, and uses it to set a report property:

```
// add an image as a report property...
URL imageURL = getClass().getResource("/com/jrefinery/report/demo/gorilla.jpg");
Image image = Toolkit.getDefaultToolkit().createImage(imageURL);
WaitingImageObserver obs = new WaitingImageObserver(image);
obs.waitImageLoaded();
this.report.setProperty("logo", image);
this.report.setPropertyMarked("logo", true);
```

The `WaitingImageObserver` is used to ensure that the image is fully loaded before the report property is set.

To display the property in the report header, you could use the following image field element:

```
<image-field x="0" y="0" width="100" height="100" fieldname="logo" />
```

8.2 Global Report Configuration

8.2.1 Overview

Global report configuration settings are used to control certain aspects of the way that JFreeReport behaves, including:

- the logging behaviour;
- the location of the DTD for the report template files;
- auto-initialisation of the `PDFOutputTarget`.

The configuration is stored in a `ReportConfiguration` object, which you can access using:

```
ReportConfiguration config = ReportConfiguration.getGlobalConfig();
```

8.2.2 Disabling Logging

You can disable logging by adding the following code near the beginning of your code:

```
ReportConfiguration config = ReportConfiguration.getGlobalConfig();
config.setDisableLogging(true);
```

It is important that you set this configuration property *before* the classloader loads the `Log` class. When the `Log` class is loaded, it reads the global report configuration and sets up the logging framework—after that point, it is not possible to disable logging without restarting the JVM.

8.2.3 Setting the Log Threshold

As an alternative to disabling logging altogether, you can set the threshold for messages that are logged. The four levels of log messages are:

- `Error` – error messages;
- `Warning` – warning messages;
- `Info` – information messages;
- `Debug` – debug messages;

To set the threshold for message output, add the following code near the beginning of your program:

```
ReportConfiguration config = ReportConfiguration.getGlobalConfig();
config.setLogLevel("Info");
```

This will filter out messages that are not at the same level or a lower level than the specified message level. In the example, error, warning and info messages would be generated, but debug messages would be filtered out.

8.3 Acrobat PDF Output

8.3.1 Output Direct to PDF

It is possible to output a report directly to PDF without showing a print preview frame or displaying the "save as PDF" dialog. This is particularly useful if you are running JFreeReport in a servlet environment.

Here is some code that writes a report directly to a PDF file—it is taken from the *StraightToPDF.java* demonstration application:

```
/**
 * Saves a report to PDF format.
 *
 * @param report  the report.
 * @param fileName target file name.
 *
 * @return true or false.
 */
public boolean savePDF(JFreeReport report, String fileName)
{
  OutputStream out = null;
  try
  {
    out = new BufferedOutputStream(new FileOutputStream(new File(fileName)));
    PageFormat pf = report.getDefaultPageFormat();
    PDFOutputTarget target = new PDFOutputTarget(out, pf, true);
    target.configure(report.getReportConfiguration());
    target.open();

    PageableReportProcessor proc = new PageableReportProcessor(report);
    proc.setOutputTarget(target);
    proc.processReport();

    target.close();
    return true;
  }
  catch (Exception e)
  {
    System.err.println("Writing PDF failed.");
    System.err.println(e.toString());
    return false;
  }
  finally
  {
    try
    {
      out.close();
    }
    catch (Exception e)
    {
      System.err.println("Saving PDF failed.");
      System.err.println(e.toString());
    }
  }
}
```

9 Package: com.jrefinery.report

9.1 Overview

This package contains the major classes and interfaces in the JFreeReport class library.

9.2 Band

9.2.1 Overview

The base class for all report bands. There are seven types of report band:

Band:	**Description:**
`ReportHeader`	An optional band that is displayed once at the start of the report.
`ReportFooter`	An optional band that is displayed once at the end of the report.
`PageHeader`	An optional band displayed at the top of every page.
`PageFooter`	An optional band displayed at the bottom of every page.
`GroupHeader`	A band that is displayed at the beginning of a report group.
`GroupFooter`	A band that is displayed at the end of a report group.
`ItemBand`	A band containing fields that are displayed once for each row of data in the table.

Each band contains a collection of *report elements* (subclasses of `Element`). The band itself is a report element.

9.2.2 Constructors

The default constructor is the only one provided.

9.2.3 Methods

To add an element to a band:

```
public void addElement(Element element);
```
Adds a report element to the band.

To get an existing element from the band:

```
public Element getElement(String name);
```
Returns the report element with the given name (or `null` if there is no element with that name).

To remove an element from the band:

```
public void removeElement(Element e);
```
Removes the specified element from the band.

To add a collection of elements to a band:

> `public void addElements(Collection elements);`
> Adds a collection of report elements to the band.

To get the default style-sheet for the element's in the band:

> `public ElementStyleSheet getBandDefaults();`
> Returns the default style-sheet for the elements in the band. Whenever an element is added to the band, this style-sheet will be added as a parent style-sheet for the element.

To get the content type for the band:

> `public String getContentType();`
> Returns the content type for the band (`X-container`).

9.2.4 Notes

The `toString()` method has been overridden to display debugging information.

See Also

`JFreeReport`, `Element`, `ElementStyleSheet`, `OutputTarget`.

9.3 DataRow

9.3.1 Overview

An interface that provides access to the items in the current row of the report. This includes:

- the fields (or columns) of the report's `TableModel`;
- any marked report properties;
- the values of the report's functions and expressions.

9.3.2 Methods

To get a value by column index:

> `public Object get(int col);`
> Returns the value of a specific column (referenced by index) in the current data row.

To get a value by column name:

> `public Object get(String col);`
> Returns the value of a specific column (referenced by name) in the current data row.

To get the name of a specific column in the data row:

> `public String getColumnName(int col);`
> Returns the name of the column (specified by index).

To get the index of a specific column in the data row:

> `public int findColumn(String name);`
> Returns the index of the column (specified by name).

To get the number of columns in the data row:

> `public int getColumnCount();`
> Returns the number of columns in the data row.

See Also

`DataRowDataSource`.

9.4 DataRowBackend

9.4.1 Overview

Maintains the state of the current data row, including the functions and expressions and report properties.

9.4.2 Methods

To get the report's `TableModel`:

> `public TableModel getTableModel();`
> Returns the report's `TableModel`

To get the current row index:

> `public int getCurrentRow();`
> Returns the current row index.

To get the functions and expressions:

> `public LevelledExpressionList getFunctions();`
> Returns the current list of functions and expressions.

9.5 DataRowConnector

9.5.1 Overview

This class connects the report element's data source chains to the underlying table model data, functions, expressions and report properties. Implements the `DataRow` interface.

9.6 Element

9.6.1 Overview

The abstract base class for all report elements. An element typically displays a small item of text, either a constant (label) or a value taken from the element's `DataSource`. An element can also be a shape (lines and rectangles for now, but other shapes may be added in the future).

The `Element` class implements the `DataTarget` interface.

Figure 1 illustrates the hierarchy of `Element` classes.

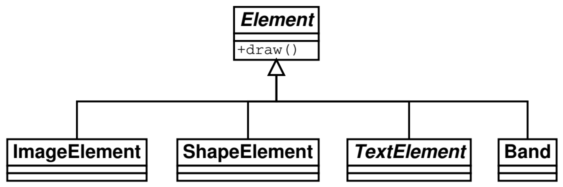


Figure 1: The `Element` classes

Elements are added to *report bands* (subclasses of the `Band` class). They are populated, as required, by the reporting engine during report generation.

9.6.2 Constructors

The default constructor is protected:

```
protected Element();
```
Creates a new element.

9.6.3 Attributes

All elements have a `name` attribute. This can be accessed via the `getName()` and `setName(String)` methods.

9.6.4 Methods

To find out the content type for an element:

```
public abstract String getContentType();
```
Returns the content type for the element.

To set the data source for an element:

```
public void setDataSource(DataSource ds);
```
Sets the data source for the element.

9.6.5 Notes

The element name should uniquely identify the element within a report. In the current version of JFreeReport, this isn't relied upon, but in the future it might be (for example, by a GUI report designer).

See Also

`Band`, `TextElement`, `DataTarget`.

9.7 ElementAlignment

9.7.1 Overview

An enumeration of the horizontal and vertical alignment settings for a report element (`Element`).

It is not possible to create new instances of this class, you can only access the predefined values:

Type:	**Description:**
`ElementAlignment.LEFT`	Left alignment.
`ElementAlignment.CENTER`	Center alignment (horizontal).
`ElementAlignment.RIGHT`	Right alignment.
`ElementAlignment.TOP`	Top alignment.
`ElementAlignment.MIDDLE`	Middle alignment (vertical).
`ElementAlignment.BOTTOM`	Bottom alignment (vertical).

See Also

`Element`, `AlignmentObjectDescription`.

9.8 Group

9.8.1 Overview

A group defines the fields (columns in the `TableModel`) on which the report data is grouped. Consecutive rows containing the same values in these fields are considered to belong to the same group.

A group header and a group footer can be assigned to the group. By default, an empty header and an empty footer are created.

9.8.2 Constructors

There is a default constructor:

```
public Group();
```
Creates a new group.

9.8.3 Methods

To get the group name:

```
public String getName();
```
Returns the name of the group.

To add a field to the group:

```
public void addField(String name);
```
Adds a field to the group. The field name should correspond to a column name in the report's `TableModel`.

To find out if the current row of data is the last in this group:

`public boolean lastItemInGroup(TableModel data, int row);`
Returns `true` if this row is the last in this group. Used by the reporting engine.

9.8.4 Notes

Report functions can reference groups by name.

See Also

`GroupHeader`, `GroupFooter`.

9.9 GroupFooter

9.9.1 Overview

A group footer is a report band that is printed at the end of a group. This class extends the `Band` class.

9.9.2 Constructors

There is a default constructor:

`public GroupFooter();`
Creates a new (empty) group footer.

9.9.3 Methods

This class adds no methods to those inherited from the `Band` class.

See Also

`Band`, `Group`.

9.10 GroupHeader

9.10.1 Overview

The group header is a report band that contains elements that should be printed at the start of a group. This class extends the `Band` class.

9.10.2 Constructors

There is a default constructor:

`public GroupHeader();`
Creates a new (empty) group header.

9.10.3 Methods

This class adds no methods to those inherited from the the `Band` class.

See Also

`Band`, `Group`.

9.11 GroupList

9.11.1 Overview

Maintains a list of `Group` objects in order. Used by the `JFreeReport` class.

9.12 ImageElement

9.12.1 Overview

A report element for displaying images. Supported image types include PNG, JPEG, WMF and GIF. This class extends `Element`.

9.12.2 Constructors

This class has only a default constructor.

9.12.3 Notes

This element expects to receive an `ImageReference` from its `DataSource`.

See Also

`Element`.

9.13 ImageReference

9.13.1 Overview

A reference to an image. This class records the URL for the image, the requested size, and also a `java.awt.Image` representing the image.

9.13.2 Constructors

To create a new image reference:

```
public ImageReference(URL url) throws IOException;
```

Reads an image from a URL.

To create a new image reference without an associated URL:

```
public ImageReference(Image img);
```

Creates an image reference for the supplied image.

9.13.3 Notes

The image types supported by this class are all those that can be loaded with the `createImage(...)` method in the `Toolkit` class, plus the WMF format supported by Pixie.

See Also

`ImageElement`.

9.14 ItemBand

9.14.1 Overview

A report band that is printed once for every row of data in the report's `TableModel`. This class extends the `Band` class.

9.14.2 Constructors

There is a default constructor:

> `public ItemBand();`
> Creates a new item band.

9.14.3 Methods

This class adds no methods to those inherited from the `Band` class.

9.15 ItemFactory

9.15.1 Overview

A utility class containing static methods for constructing report elements.

See Also

`Element`.

9.16 JFreeReport

9.16.1 Overview

This class is used to represent reports. Each report instance will maintain formatting and configuration information for the report, plus a reference to some data in the form of a `TableModel`.

Most often you will pass the `JFreeReport` object to a `PreviewFrame` for display, printing, or saving to file.

9.16.2 Usage

There are two methods for creating a report instance:

- from an XML report template (see section 4);
- using Java code;

If you have a report template file, you can parse it to create a `JFreeReport` object as follows:

```
URL in  = getClass().getResource("/path/to/report/MyReport.xml");
JFreeReport report = null;
ReportGenerator generator = ReportGenerator.getInstance();
try
{
  report = generator.parseReport(in);
}
catch (Exception e)
{
  ExceptionDialog.showExceptionDialog("Error on parsing",
                                      "Error while parsing " + in, e);
}
```

9.16.3 Report Properties

Every report maintains a list of report properties. A *report property* is an object (any `Object` instance) that is stored/retrieved using a key (a `String`).

To set a report property:

> `public void setProperty (String key, Object value);`
> Adds a property to the report using the specified key. If the supplied value is `null`, the property is removed from the collection. *Properties must be set before report processing begins, otherwise the property will not be visible to the reporting engine.*

To retrieve the value of a property:

> `public Object getProperty (String key);`
> Returns the property with the specified key.

An instance of the `ReportProperties` class is used to store the report properties. You can get a reference to this storage container using:

> `public ReportProperties getProperties();`
> Returns a reference to the report properties storage container.

When a report is processed, the report properties are cloned, and passed along the `ReportState` chain as processing proceeds.

To make a property available via the `DataRow` (used during report processing), it needs to be "marked":

> `public void setPropertyMarked (String key, boolean mark);`
> Marks a property for access via the data row.

9.16.4 Constructors

There is a default constructor which creates an empty report:

> `public JFreeReport();`
> Creates a new (empty) report. After creating the report, you can add formatting and configuration information.

There is an alternative "monolithic" constructor:

```
public JFreeReport(String name, ReportHeader reportHeader,
ReportFooter reportFooter, PageHeader pageHeader,
PageFooter pageFooter, ItemBand itemBand,
GroupList groups, Collection functions,
TableModel data, PageFormat defaultPageFormat);
```

Creates a new report.

If you parse reports from an XML report template file (the recommended approach) you will not need to use these constructors.

9.16.5 Methods

To set the data that will be used to generate the report:

```
public void setData(TableModel data);
```

Sets the data that will be used to generate the report. The column names in the `TableModel` are important, since they are referenced by individual elements in the report, and also by the group definitions.

Important note: the data will be processed in the order that it is represented in the `TableModel`, JFreeReport does NOT perform any sorting. You should presort your data, particularly if you are using report groups.

To access the report's configuration settings:

```
public ReportConfiguration getReportConfiguration();
```

Returns the report's configuration settings.

9.16.6 Notes

The `JFreeReport.INFO` object contains information about the JFreeReport project.

See Also

`JFreeReportConstants`.

9.17 JFreeReportConstants

9.17.1 Overview

An interface that defines some useful constants used by the `JFreeReport` class.

Constant:	**Description:**
`NAME_PROPERTY`	The report name property key ("report.name").
`REPORT_DATE_PROPERTY`	The report date property key ("report.date").
`REPORT_PAGEFORMAT_PROPERTY`	The report page format property key ("report.pageformat").
`REPORT_PREPARERUN_PROPERTY`	The report "prepare run" property key ("report.preparerun").
`REPORT_DEFINITION_SOURCE`	The report definition source property key ("report.definition.source").
`REPORT_DEFINITION_CONTENTBASE`	The report definition content base property key ("report.definition.contentbase").

9.17.2 Notes

This interface is implemented by the following classes:

- `JFreeReport`;
- `ReportState`;

9.18 JFreeReportInfo

9.18.1 Overview

This class holds information about the JFreeReport class library, including a list of the developers that have contributed to the project.

9.18.2 Notes

It is intended that there will be one instance of the class: `JFreeReport.INFO`.

Some of the information in this class is obtained from the localised resource bundle.

See Also

`JFreeReport`.

9.19 PageFooter

9.19.1 Overview

A report band that appears at the bottom of every page in the report. There are flags that allow the report footer to be suppressed on the first and/or last pages of the report.

9.19.2 Constructors

There is a default constructor:

```
public PageFooter();
```

Creates a new (empty) page footer.

9.19.3 Methods

You can set a flag that controls whether or not the footer is displayed on the first page:

```
public void setDisplayOnFirstPage(boolean flag);
```

Sets a flag that controls whether or not the footer is displayed on the first page of the report.

You may want to suppress the footer on the first page if there is a report header (particularly if the report header occupies all of the first page).

Similarly, you can set a flag that controls whether or not the footer is displayed on the last page:

> `public void setDisplayOnLastPage(boolean flag);`
> Sets a flag that controls whether or not the footer is displayed on the last page of the report.

You may want to suppress the footer on the last page if there is a report footer.

See Also

`Band`, `PageHeader`.

9.20 PageHeader

9.20.1 Overview

A report band that appears at the top of every page. There is a flag that allows the report header to be suppressed on the first page and/or last pages of the report.

9.20.2 Constructors

There is a default constructor:

> `public PageHeader();`
> Creates a new (empty) page header.

9.20.3 Methods

You can set a flag that controls whether or not the header is displayed on the first page:

> `public void setDisplayOnFirstPage(boolean flag);`
> Sets a flag that controls whether or not the header is displayed on the first page of the report.

You may want to suppress the header on the first page if there is a report header.

Similarly, you can set a flag that controls whether or not the header is displayed on the last page:

> `public void setDisplayOnLastPage(boolean flag);`
> Sets a flag that controls whether or not the header is displayed on the last page of the report.

You may want to suppress the header on the last page if there is a report footer (particularly if if occupies all of the last page of the report).

See Also

`Band`, `PageFooter`.

9.21 ReportFooter

9.21.1 Overview

A report band that is printed once only at the end of a report. Extends the `Band` class.

9.21.2 Constructors

There is a default constructor:

> `public ReportFooter();`
> Creates a new (empty) report footer.

9.21.3 Methods

You can set a flag that controls whether the report footer occupies the whole of the last page:

> `public boolean setOwnPage(boolean flag);`
> Sets a flag that controls whether the report footer occupies all of the last page.

This is useful if you want to design your report footer as a report summary page.

9.21.4 Notes

If there is a page footer, and it hasn't been suppressed for the last page of the report, it will be printed *after* the report footer.

See Also

`Band`.

9.22 ReportHeader

9.22.1 Overview

A report band that is printed once only at the start of a report. Extends the `Band` class.

9.22.2 Constructors

There is a default constructor:

> `public ReportHeader();`
> Creates a new (empty) report header.

9.22.3 Methods

You can set a flag that controls whether the report header occupies the whole of the first page:

> `public boolean setOwnPage(boolean flag);`
> Sets a flag that controls whether the report header occupies all of the first page.

This is useful if you want to design your report header as a report title page.

See Also

`Band`.

9.23 ReportInitialisationException

9.23.1 Overview

An exception that indicates an error has occurred while processing a report.

9.23.2 Constructors

In addition to the default constructor, the following is available:

> `public ReportProcessingException(String message);`
> Creates a new exception with a specific message.

See Also

`PageableReportProcessor`.

9.24 ReportInterruptedException

9.24.1 Overview

An exception that indicates that the report processing has been interrupted.

9.25 ReportProcessingException

9.25.1 Overview

An exception that indicates an error has occurred while processing a report.

9.25.2 Constructors

In addition to the default constructor, the following is available:

> `public ReportProcessingException(String message);`
> Creates a new exception with a specific message.

See Also

`PageableReportProcessor`.

9.26 ShapeElement

9.26.1 Overview

A report element that draws shapes.

See Also

`Element`.

9.27 TextElement

9.27.1 Overview

A report element that displays text. With the introduction of the `DataFilter` interface—and the classes that implement it—the `TextElement` class becomes very versatile. With appropriate filters, you can display labels, strings, dates and numbers, all with appropriate formatting.

This class extends the `Element` class.

9.27.2 Constructors

The only constructor is the default constructor. If you create a `TextElement` object yourself, be sure to set it's attributes using the accessor methods.

9.27.3 Methods

To retrieve the text that will be displayed by this element, use the `getValue()` method defined by the super-class.

If the value to be displayed by the field is `null`, a special string is substituted in the output. Use the `getNullString()` and `setNullString(String)` methods to control this.

9.27.4 Notes

Font and alignment settings are controlled by the element's style sheet.

The `toString()` method has been overridden to supply debugging information, if required.

See Also

`Element`, `Band`.

10 Package: c.j.r.action

10.1 Overview

This package contains abstract classes for common actions related to reports. Localised resources have been compiled for these standard actions (see section 24 for details).

10.2 AboutAction

10.2.1 Overview

An abstract action for displaying information about JFreeReport, or an application that uses JFreeReport. Subclasses will supply an `actionPerformed(...)` method implementation.

10.2.2 Constructor

The following constructor is available for the use of subclasses:

```
public AboutAction(ResourceBundle resources);
```

Creates a new action using the supplied resources.

10.2.3 Notes

Localised resources are used to initialise the action:

Resource:	Description:
`action.about.name`	The action name.
`action.about.description`	A short description of the action.
`action.about.mnemonic`	The action mnemonic.

The action name is typically used for the text on buttons or menu items created using this action. The description is often used as the tooltip text. In addition to these localised resources, this class loads the `About16.gif` and `About24.gif` icons.

Refer to Javadoc HTML files and source code for further details.

See Also

`JFreeReportResources`.

10.3 CloseAction

10.3.1 Overview

An abstract action for closing the JFreeReport print preview frame.

10.3.2 Constructor

The following constructor is available for the use of subclasses:

```
public CloseAction(ResourceBundle resources);
```
Creates a new action using the supplied resources.

10.3.3 Notes

Localised resources are used to initialise the action:

Resource:	Description:
`action.close.name`	The action name.
`action.close.description`	A short description of the action.
`action.close.mnemonic`	The action mnemonic.

The action name is typically used for the text on buttons or menu items created using this action. The description is often used as the tooltip text. There are no icons associated with this action.

Refer to Javadoc HTML files and source code for further details.

See Also

`JFreeReportResources`.

10.4 FirstPageAction

10.4.1 Overview

Not yet documented.

See Also

`JFreeReportResources`.

10.5 GotoPageAction

10.5.1 Overview

Not yet documented.

See Also

`JFreeReportResources`.

10.6 LastPageAction

10.6.1 Overview

Not yet documented.

See Also

`JFreeReportResources`.

10.7 NextPageAction

10.7.1 Overview

Not yet documented.

See Also

`JFreeReportResources`.

10.8 PageSetupAction

10.8.1 Overview

An abstract action for displaying a page setup dialog. Subclasses will supply an `actionPerformed(...)` method implementation.

10.8.2 Constructor

The following constructor is available for the use of subclasses:

```
public PageSetupAction(ResourceBundle resources);
```

Creates a new action using the supplied resources.

10.8.3 Notes

Localised resources are used to initialise the action:

Resource:	**Description:**
`action.page-setup.name`	The action name.
`action.page-setup.description`	A short description of the action.
`action.page-setup.mnemonic`	The action mnemonic.

The action name is typically used for the text on buttons or menu items created using this action. The description is often used as the tooltip text. In addition to these localised resources, this class loads the `PageSetup16.gif` and `PageSetup24.gif` icons.

Refer to Javadoc HTML files and source code for further details.

See Also

`JFreeReportResources`.

10.9 PreviousPageAction

10.9.1 Overview

Not yet documented.

See Also

`JFreeReportResources`.

10.10 PrintAction

10.10.1 Overview

An abstract action for printing a report. Subclasses will supply an `actionPerformed(...)` method implementation.

10.10.2 Constructor

The following constructor is available for the use of subclasses:

```
public PrintAction(ResourceBundle resources);
```

Creates a new action using the supplied resources.

10.10.3 Notes

Localised resources are used to initialise the action:

Resource:	**Description:**
`action.print.name`	The action name.
`action.print.description`	A short description of the action.
`action.print.mnemonic`	The action mnemonic.

The action name is typically used for the text on buttons or menu items created using this action. The description is often used as the tooltip text. In addition to these localised resources, this class loads the `Print16.gif` and `Print24.gif` icons.

Refer to Javadoc HTML files and source code for further details.

See Also

`JFreeReportResources`.

10.11 SaveAsAction

10.11.1 Overview

An abstract action for saving a report (for now, in PDF format). Subclasses will supply an `actionPerformed(...)` method implementation.

10.11.2 Constructor

The following constructor is available for the use of subclasses:

```
public SaveAsAction(ResourceBundle resources);
```

Creates a new action using the supplied resources.

10.11.3 Notes

Localised resources are used to initialise the action:

Resource:	Description:
`action.save-as.name`	The action name.
`action.save-as.description`	A short description of the action.
`action.save-as.mnemonic`	The action mnemonic.
`action.save-as.accelerator`	The action accelerator.

The action name is typically used for the text on buttons or menu items created using this action. The description is often used as the tooltip text. In addition to these localised resources, this class loads the `SaveAs16.gif` and `SaveAs24.gif` icons.

Refer to Javadoc HTML files and source code for further details.

See Also

`JFreeReportResources`.

10.12 ZoomInAction

10.12.1 Overview

Not yet documented.

See Also

`JFreeReportResources`.

10.13 ZoomOutAction

10.13.1 Overview

Not yet documented.

See Also

`JFreeReportResources`.

11 Package: c.j.r.event

11.1 Overview

This package contains classes that implement report events. The reporting engine will generate events during report processing. You can listen for these events via the `ReportListener` interface.

11.2 ReportEvent

11.2.1 Overview

This class records information about a report event.

11.2.2 Constructor

To construct a report event:

> `public ReportEvent(ReportState state);`
> Creates a new report event. A valid `state` must be supplied, `null` is not allowed.

11.2.3 Methods

To get the report state at the time of the event:

> `public ReportState getState();`
> Returns the report state at the time of the event.

To obtain a reference to the report that an event has been generated for:

> `public JFreeReport getReport();`
> Returns the report that an event has been generated for. This is a convenience method that extracts the required information from the report state.

See Also

`ReportListener`, `ReportState`.

11.3 ReportListener

11.3.1 Overview

This interface defines the methods that must be implemented by *report listeners*.

11.3.2 Methods

The following methods will be called by the reporting engine, during report generation, for all registered listeners:

> `public void reportStarted(ReportEvent event);`
> This method is called at the start of report processing.

`public void reportFinished(ReportEvent event);`
This method is called at the end of report processing.

`public void pageStarted(ReportEvent event);`
This method is called at the start of each page.

`public void pageFinished(ReportEvent event);`
This method is called at the end of each page.

`public void groupStarted(ReportEvent event);`
This method is called at the start of each group.

`public void groupFinished(ReportEvent event);`
This method is called at the end of each group.

`public void itemsStarted(ReportEvent event);`
This method is called at the start of each group of items.

`public void itemsFinished(ReportEvent event);`
This method is called at the end of each group of items.

`public void itemsAdvanced(ReportEvent event);`
This method is called when the cursor is advanced from one row to the next.

11.3.3 Notes

Among other things, the report listener mechanism is used to implement report functions.

See Also

`ReportEvent`, `Function`.

12 Package: c.j.r.filter

12.1 Overview

This package contains interfaces and classes that are used to implement data filters.

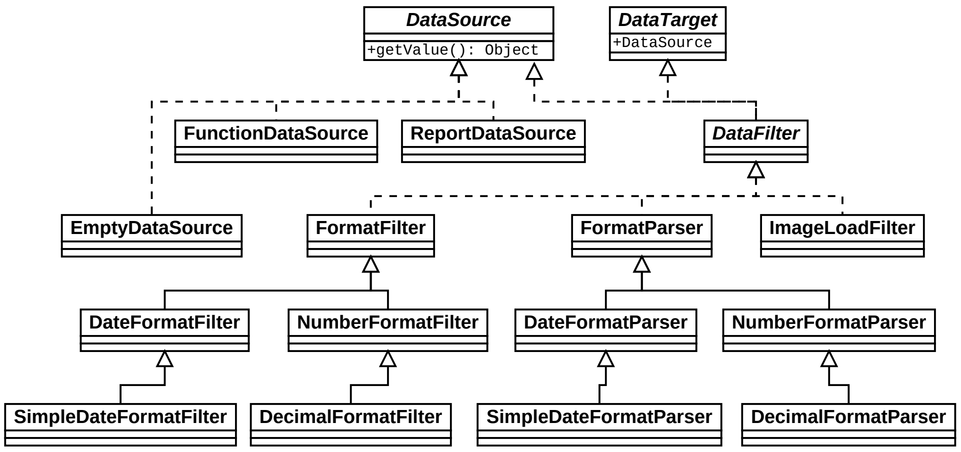


Figure 2: Classes

Filters are used to transform objects from other data sources.

12.2 DataFilter

12.2.1 Overview

The `DataFilter` interface is simply a combination of the `DataSource` and `DataTarget` interfaces.

Data filters are used to transform data values. By chaining together filters, complex transformations can be performed in a flexible way.

12.2.2 Notes

The following filters have been implemented:

- `StringFilter` – converts an `Object` to a `String`;

See Also

`DataSource`, `DataTarget`.

12.3 DataRowConnectable

12.3.1 Overview

An interface that specifies the connect and disconnect methods for a data source that can access values from the data row.

12.3.2 Notes

The `DataRowDataSource` class implements this interface.

See Also

`DataRowDataSource`.

12.4 DataRowDataSource

12.4.1 Overview

A `DataSource` that can access values from the current `DataRow`. This data source provides access to all the fields in the current row of the report's `TableModel`, plus the current values of all the functions and expressions defined for the report.

12.4.2 Constructor

The default constructor creates a data source with an empty column reference:

```
public DataRowDataSource();
```
Creates a data source with an empty column reference.

The other constructor allows you to specify the column reference (that is, the name of the field, expression or function):

```
public DataRowDataSource(String column);
```
Creates a data source that accesses the specified column in the current data row.

12.4.3 Methods

To set the column name for the data source:

```
public void setDataSourceColumnName(String name);
```
Sets the column name (`null` not permitted).

To get the connected data row (if any):

```
public DataRow getDataRow();
```
Returns the current data row.

See Also

`DataRow`.

12.5 DataSource

12.5.1 Overview

An interface that defines the behaviour of a *data source*. Data sources are used to access values for displaying in a report.

A data source passes its value to a `DataTarget`. The elements that appear on formatted reports (instances of the `Element` class) are `DataTarget` objects. The path between the `DataSource` and the `DataTarget` can be direct, or it can pass through a chain consisting of one or more `DataFilter` objects.

The most fundamental data source is the `DataRowDataSource`, because it provides access to the items in the current "data row" (the data row is an aggregation of the current row in the report's `TableModel`, and the current values for the functions, expressions and report properties.

There are other implementations of the `DataSource` interface, including:

- `StaticDataSource`;
- `EmptyDataSource`.

12.5.2 Methods

To obtain the current value of the data source:

```
public Object getValue();
```
Returns the current value for the data source.

All data sources are cloneable:

```
public Object clone();
```
Returns a clone of the data source.

See Also

`DataFilter`, `DataTarget`.

12.6 DataTarget

12.6.1 Overview

A *data target* is a consumer in the data processing chain. The target obtains data from a `DataSource`. All report elements (subclasses of the `Element` class) implement the `DataTarget` interface.

The path from a `DataSource` to a `DataTarget` can be direct, or you can chain together one or more `DataFilter` objects between the `DataSource` and the `DataTarget`. This provides a great deal of flexibility to manipulate a data item between its original source and its ultimate destination on a report.

12.6.2 Methods

To get the data source for this target:

```
public DataSource getDataSource();
```
Returns the data source.

To set the data source for this target:

```
public void setDataSource(DataSource ds);
```
Sets the data source.

12.6.3 Notes

The `DataFilter` interface extends the `DataTarget` interface, combining it with the `DataSource` interface. By combining both the producer and consumer roles, it is possible to create chains of `DataFilter` objects linking a `DataSource` object to a `DataTarget` object.

See Also

`DataSource`.

12.7 DateFormatFilter

12.7.1 Overview

A filter that converts a `Date` object to a formatted `String` object. This class extends the `FormatFilter` class.

12.7.2 Methods

To obtain a reference to the object responsible for the formatting:

```
public DateFormat getDateFormat();
```
Returns the formatter for this filter.

To set the formatter:

```
public void setDateFormat(DateFormat df);
```
Sets the formatter for this filter.

12.7.3 Notes

The `setFormatter(...)` method is overridden to ensure that only `DateFormat` instances are assigned as the formatter for this filter.

There is a `SimpleDateFormatFilter` subclass that works specifically with the `SimpleDateFormat` class.

See Also

`FormatFilter`, `SimpleDateFormatFilter`.

12.8 DateFormatParser

12.8.1 Overview

A filter that converts a `String` to a `Date`. Extends the `FormatParser` class.

12.8.2 Methods

To set the value returned when the data source value is `null`:

```
public void setNullValue(Object nullValue);
```
Sets the object to return in the case where the data source value is `null`.
Note that `nullValue` should be of type `Date`.

See Also

`FormatParser`.

12.9 DecimalFormatFilter

12.9.1 Overview

A filter that converts a `Number` object to a formatted `String` object, using an instance of `DecimalFormat` to perform the conversion. Extends the `NumberFormatFilter` class.

See Also

`NumberFormatFilter`.

12.10 DecimalFormatParser

12.10.1 Overview

A filter that converts a `String` to a `Number` object.

See Also

`NumberFormatParser`.

12.11 EmptyDataSource

12.11.1 Overview

A data source that always returns `null`.

12.11.2 Notes

An instance of this class is defined as the `NULL_DATASOURCE` constant in the `Element` class.

See Also

`DataSource`.

12.12 ExpressionDataSource

12.12.1 Overview

Not documented.

12.13 FormatFilter

12.13.1 Overview

A data filter that converts `Object` instances into `String` instances. Implements the `DataFilter` interface.

See Also

`DataFilter`.

12.14 FormatParser

12.14.1 Overview

A generic data filter used to convert a `String` to an `Object`. Implements the `DataFilter` interface.

12.14.2 Notes

Various subclasses have been implemented to handle specific `String` to `Object` conversions.

See Also

`DataFilter`, `DateFormatParser`.

12.15 FunctionDataSource

12.15.1 Overview

A data source that obtains its value from a report function.

12.15.2 Notes

The type of value returned depends on the function referenced. You can chain together one or more data filters to transform the return value if necessary.

See Also

`DataSource`.

12.16 ImageLoadFilter

12.16.1 Overview

A data filter that takes an `URL` as input, and outputs an `ImageReference` object.

See Also

`DataFilter`.

12.17 ImageRefFilter

12.17.1 Overview

A data filter that converts incoming `Image` objects (from another `DataSource`) into `ImageReference` objects.

This class implements the `DataFilter` interface.

12.17.2 Notes

This class will also accept an `ImageReference` from its datasource, passing it along the filter chain without modification.

12.18 NumberFormatFilter

12.18.1 Overview

A data filter that converts a `Number` object to a formatted `String` object. Extends the `FormatFilter` class.

See Also

`DataFilter`.

12.19 NumberFormatParser

12.19.1 Overview

A data filter that converts `String` objects to `Number` objects. Extends the `FormatParser` class.

See Also

`FormatParser`.

12.20 ReportDataSource

12.20.1 Overview

A data source that obtains its value from a column in the report's `TableModel`.

12.20.2 Methods

To specify the name of the column in the `TableModel` from which the data source will obtain its value:

> `public void setField(String field);`
> Sets the name of the column in the table model from which the data value is obtained.

12.20.3 Notes

This data source returns whatever `Object` it finds in the table model. You can chain together data filters to transform this object into other forms, if necessary.

See Also

`DataSource`.

12.21 ResourceFileFilter

12.21.1 Overview

A filter that looks up and returns a resource from a `ResourceBundle`.

See Also

12.22 SimpleDateFormatFilter

12.22.1 Overview

A data filter that converts `Date` objects into formatted `String` objects. Extends the `DateFormatFilter` class.

See Also

`DateFormatFilter`.

12.23 SimpleDateFormatParser

12.23.1 Overview

A data filter that is used to convert `String` objects into `Date` objects. Extends the `DateFormatParser` class.

12.23.2 Notes

If the value obtained from the datasource is not a `String`, it is converted using the `String.valueOf(...)` method.

See Also

`DateFormatParser`.

12.24 StaticDataSource

12.24.1 Overview

A *static data source* is a `DataSource` that returns a fixed value (the value can be any instance of `Object`).

12.24.2 Methods

To set the value for the static data source:

```
public void setValue(Object value);
```
Sets the value for this data source.

12.24.3 Notes

The `createLabelElement(...)` method in the `ItemFactory` class uses an instance of this class to construct a report item for displaying a label.

See Also

`DataSource`.

12.25 StringFilter

12.25.1 Overview

A filter (`DataFilter`) that converts the value (`Object`) obtained from its data source (`DataSource`) into a `String`.

The string filter also provides a *null string* attribute, text that is returned when the source value is `null`.

12.25.2 Methods

You can define a value that will be returned by this filter when the value obtained from the `DataSource` is `null`:

```
public void setNullValue(String nullValue);
```
Sets the value to be returned when the value from the data source is `null`.

12.26 URLFilter

12.26.1 Overview

A filter that converts the value obtained from its datasource to a `URL`.

This filter will accept several object types from its datasource: `URL`, `File` and `String`.

12.26.2 Methods

You can specify a *base URL* that is used to complete relative URLs:

> `public void setBaseURL(URL baseURL);`
> Sets the base URL used to complete relative URLs.

12.26.3 Notes

The `createImageURLField(...)` method in the `ItemFactory` class creates an instance of this filter when constructing a report item for displaying images.

See Also

`DataFilter`.

13 Package: c.j.r.filter.templates

13.1 Overview

A *template* is a `DataSource` that is constructed by chaining together a `DataSource` and one or more `DataFilter` instances. Templates are provided as a convenience, to make it easier to work with commonly used constructs.

13.2 AbstractTemplate

13.2.1 Overview

An abstract implementation of the `Template` interface. Concrete subclasses include:

- `DateFieldTemplate`
- `ImageFieldTemplate`
- `ImageURLElementTemplate`
- `ImageURLFieldTemplate`
- `LabelTemplate`
- `NumberFieldTemplate`
- `ResourceFieldTemplate`
- `ResourceLabelTemplate`
- `StringFieldTemplate`

See Also

`AbstractTemplateDescription`.

13.3 DateFieldTemplate

13.3.1 Overview

A *date field template* is used to access a `Date` value from the data row.

The template chains together the following objects:

- `DataRowDataSource` - to access the date from the data row;
- `SimpleDateFormatFilter` - to format the date as a `String`;
- `StringFilter` - used to add a “null” string, the text to display when the source data is `null`;

See Also

`DateFieldTemplateDescription`.

13.4 ImageFieldTemplate

13.4.1 Overview

An *image field template* is used to access an `Image` value from the data row.

The template chains together:

- `DataRowDataSource` - to access the `Image` from the data row;
- `ImageRefFilter` - wraps the raw `Image` in an `ImageReference`;

See Also

`ImageFieldTemplateDescription`.

13.5 ImageURLElementTemplate

13.5.1 Overview

An *image URL element template* is used to provide access to an image via a static URL.

The template chains together:

- `StaticDataSource` - to supply the `URL`;
- `URLFilter` - to convert the `Object` from the source into a `URL`.
- `ImageLoadFilter` - to load the image from the URL;

See Also

`ImageURLElementTemplateDescription`.

13.6 ImageURLFieldTemplate

13.6.1 Overview

An *image URL field template* is used to provide access to an image via a URL in the data row.

The template chains together:

- `DataRowDataSource` - to supply the `URL`;
- `URLFilter` - to convert the `Object` from the source into a `URL`.
- `ImageLoadFilter` - to load the image from the URL;

See Also
`ImageURLFieldTemplateDescription`.

13.7 LabelTemplate

13.7.1 Overview

A *label template* is used to display a static label.

The template chains together:

- `StaticDataSource` - to supply the static label;
- `StringFilter` - to convert the label (if necessary) into a `String`;

See Also
`LabelTemplateDescription`.

13.8 NumberFieldTemplate

13.8.1 Overview

A *number field template* is used to access a `Number` value from the data row.

The template chains together the following objects:

- `DataRowDataSource` - to access the `Number` from the data row;
- `DecimalFormatFilter` - to format the number as a `String`;
- `StringFilter` - used to add a "null" string, the text to display when the source data is `null`;

See Also
`NumberFieldTemplateDescription`.

13.9 ResourceFieldTemplate

13.9.1 Overview

A *resource field template* is used to lookup a value from a `ResourceBundle`, based on a key in the data row.

The template chains together:

- `DataRowDataSource` - to supply the resource key;
- `ResourceFileFilter` - to look up the resource value;
- `StringFilter` - to convert the resource value into a `String` (if required).

See Also

`ResourceFieldTemplateDescription`.

13.10 ResourceLabelTemplate

13.10.1 Overview

A *resource label template* is used to lookup a value from a `ResourceBundle`, based on a static key.

The template chains together:

- `StaticDataSource` - to supply the resource key;
- `ResourceFileFilter` - to look up the resource value;
- `StringFilter` - to convert the resource value into a `String` (if required).

See Also

`ResourceLabelTemplateDescription`.

13.11 StringFieldTemplate

13.11.1 Overview

A *string field template* is used to access a `String` value from the data row.

The template chains together the following objects:

- `DataRowDataSource` - to access the `String` from the data row;
- `StringFilter` - used to add a "null" string, the text to display when the source data is `null`;

See Also

`StringFieldTemplateDescription`.

13.12 Template

13.12.1 Overview

A *template* is a "ready-made" filtered `DataSource`.

Templates have been written for the most common types of data that you would display in a report.

13.12.2 Methods

The interface defines only some very basic methods for getting/setting the template name, and to create a new instance of a template (via cloning).

See Also

`TemplateDescription`.

14 Package: c.j.r.function

14.1 Overview

This package contains interfaces and classes that are used to implement report functions and expressions. A range of standard functions/expressions is supplied with JFreeReport, and you can also write your own.

14.1.1 Usage

Section 4.7 provides more information about using functions and expressions within reports.

14.2 AbstractExpression

14.2.1 Overview

An abstract base class that can be used to implement a new *report expression*. This class provides default implementations for the methods in the `Expression` interface.

Classes that extend this base class include:

- `BSHExpression`;
- `TextFormatExpression`.

14.2.2 Notes

See section 7 for information about the standard expressions available in JFreeReport.

See Also
`Expression`.

14.3 AbstractFunction

14.3.1 Overview

An abstract base class that can be used to implement a new *report function*. This class provides default implementations for the methods defined in the `Function` interface.

Classes that extend this base class include:

- `ItemCountFunction`;
- `ItemSumFunction`;
- `PageFunction`;
- `ReportPropertyFunction`;

14.3.2 Constructors

The constructor is protected, you never create an instance of this class directly:

> `protected AbstractFunction();`
> Creates a new function.

14.3.3 Methods

This class implements all the methods in the `Function` interface.

14.3.4 Notes

See section 7 for information about the standard functions available in JFreeReport.

See Also

`Function`.

14.4 BSHExpression

14.4.1 Overview

A *report expression* that incorporates the BeanShell scripting engine to allow you to write custom expressions within a report.

14.4.2 Usage

With this class, you can write a custom expression using regular Java code. This code will be interpreted by the BeanShell scripting engine, and the expression will return the result of the `getValue()` method (which you must include in your script).

The script to be evaluated is added to the `BSHExpression` instance as a property (named `expression`). For example, here is a very basic expression:

```
<expression name="MyExpression"
            class="com.jrefinery.report.function.BSHExpression">
  <properties>
    <property name="expression">
      getValue() { return "Some Result!"; }
    </property>
  </properties>
</expression>
```

The above example doesn't do anything useful (it always returns the same value), but it does illustrate the basic framework.

A more detailed example is included in section 7.8.

14.4.3 Notes

You can find out more about BeanShell at:

```
http://www.beanshell.org
```

At the time of writing, the version of BeanShell used in JFreeReport is 1.2b6.

See Also

`Expression`.

14.5 ElementVisibilitySwitchFunction

14.5.1 Overview

A function that alternates between `true` and `false` for each row in a report *and* sets the visibility of a report element from the report's *item band* accordingly.

This class extends the `AbstractFunction` class.

14.5.2 Constructors

There is a default constructor (intended mainly for use by the SAX handler):

> `public ElementVisibilitySwitchFunction();`
> Creates an unnamed function (the function is unusable until the name has been set).

14.5.3 Methods

To obtain the current value of the switch:

> `public Object getValue();`
> Returns the current value of the switch. The result can be cast to a `Boolean`.

14.5.4 Notes

One application of this function is to create alternating colored lines in a report.

See Also

`JFreeReport`.

14.6 EventMonitorFunction

14.6.1 Overview

Not yet documented.

14.7 Expression

14.7.1 Overview

An interface that defines the behaviour of a *report expression*. Report expressions are typically used to calculate values within the current row of a report. Unlike report functions (see the `Function` interface), expressions do not maintain any state information as the report processing proceeds from one row to the next.

14.7.2 Methods

Expressions are required to have a unique name, so that they can be referenced by report elements. The name should be unique among:

- the other expressions and functions in a report;
- the column names from the report's `TableModel`.

To get the name of an expression:

```
public String getName();
```
Returns the name of the expression.

To set the name of an expression:

```
public void setName(String name);
```
Sets the name of the expression.

To get the value of a property (these are used to define parameters for the expression):

```
public String getProperty(String name);
```
Returns the value of the property with the given name, or `null` if there is no such property.

Alternatively, this method returns a property value or some default value if the property is not found:

```
public String getProperty(String name, String defaultVal);
```
Returns the value of the property with the given name, or `defaultVal` if there is no such property.

To get the value of the expression:

```
public Object getValue();
```
Returns the value of the expression, which will usually be calculated "on-the-fly".

14.7.3 Notes

The `AbstractExpression` class provides a starting point for implementing new expressions.

See Also

`Function`.

14.8 ExpressionCollection

14.8.1 Overview

Not documented.

14.9 Function

14.9.1 Overview

An interface that defines the behaviour of a *report function* (see section 4.7). This interface extends the `Expression` and `ReportListener` interfaces.

Functions are named, a behaviour inherited from the `Expression` interface.

A function will often record "state" information, data that depends on the current stage of processing for the report (for example, the running total of one column in a report). For performance reasons, JFreeReport caches the "report state" at the beginning of every page, including cloning every function to record its state.

14.9.2 Methods

The `initialize()` method provides a function with an opportunity to perform any required setup, and to check that it has been correctly configured:

```
public void initialize() throws FunctionInitializeException;
```

Performs any required initialization and checks that the function has been correctly configured. Throws a `FunctionInitializeException` if there is any problem.

The current value of the function should be returned via the `getValue()` method:

```
public Object getValue();
```

Returns the current value of the function.

Functions can be cloned, a feature that is used to cache report state information at page boundaries:

```
public Object clone() throws CloneNotSupportedException;
```

Returns a clone of the function.

Functions can maintain a list of properties used to configure a particular function instance (the properties recognised by a report function will vary depending on what the function does):

```
public void setProperties(Properties p);
```

Sets the properties for a function.

14.9.3 Notes

The `AbstractFunction` class provides default implementations for the methods in this interface. It is usually a good idea to subclass `AbstractFunction` when you are creating new functions.

You need to refer to the documentation for individual report functions to see a list of the properties (if any) supported by the function.

Refer to section 4.7.4 for a description of the XML used to declare a function within a report template.

See Also

`AbstractFunction`, `ReportListener`.

14.10 FunctionInitializeException

14.10.1 Overview

An exception that signals a problem with the configuration of a report function.

See Also

`Function`.

14.11 FunctionProcessingException

14.11.1 Overview

Not yet documented.

14.12 GroupCountFunction

14.12.1 Overview

A report function that counts the number of instances of a named group in a report.

14.12.2 Methods

When the function receives notification of the *report started event*, it sets the group count to zero:

```
public void reportStarted(ReportEvent event);
```
A new report has been started, so the group count is set to zero.

When the function receives notification of the *group started event*, it checks to see if the function property `group` matches the current group, and if it does then it increments the group count:

```
public void groupStarted(ReportEvent event);
```
A new group has been started, increment the group counter.

14.12.3 Notes

This function is used in the unit testing of JFreeReport, to ensure that the reporting engine is grouping data correctly.

This class extends the `AbstractFunction` class.

See Also

`Function`.

14.13 ItemAvgFunction

14.13.1 Overview

A report function that calculates a running average for an item in a report. The name of the item to average is specified using the function's `field` property.

If the `group` property is set for the function, the running average is reset to zero every time the specified group starts. Otherwise, the running average accumulates for the entire report.

14.13.2 Notes

The result returned by this function is an instance of `Integer`.

See Also

`Function`.

14.14 ItemColumnQuotientFunction

14.14.1 Overview

A function that calculates the quotient of two fields in the same row of a report. For example, suppose you have two numerical columns "A" and "B" in your `TableModel`. This function allows you to calculate the value "A divided by B" (or "B divided by A").

14.14.2 Usage

To declare this type of function in the simple report definition format, you would use something like this:

```
<function name="MyFunction"
          class="com.jrefinery.report.function.ItemColumnQuotientFunction">
    <properties>
        <property name="dividend">A</property>
        <property name="divisor">B</property>
    </properties>
</function>
```

14.14.3 Notes

The `PercentageDemo` application included in the JFreeReport distribution (from version 0.8.2) demonstrates the use of this function.

14.15 ItemCountFunction

14.15.1 Overview

A function that counts the items (rows) in a report or report group.

14.15.2 Properties

This function recognises one property:

- `group` - the name of a report group. If this property is set, the item count is reset to zero at the beginning of each instance of the named group.

If the `group` property is not set, the item count accumulates for the entire report.

14.15.3 Constructors

The standard constructor:

```
public ItemCountFunction(String name);
```

Creates a named function.

The default constructor (intended mainly for use by the SAX handler):

```
public ItemCountFunction();
```

Creates an unnamed item count function. The function is unusable until it has been given a name.

14.15.4 Methods

At the point when the reporting engine notifies the function that the report is starting, the running count is set to zero:

```
public void reportStarted(ReportEvent event);
```

In response, this function resets the item count to zero.

At the point when the report engine notifies the function that a new report group is starting, the function checks the group name and, if it matches the value of the `group` property, the running count is reset to zero:

```
public void groupStarted(ReportEvent event);
```

If the group name matches this function's group, reset the item count to zero.

At the point when the reporting engine notifies the function that a row of data has been processed, the running count is incremented by one:

```
public void itemsAdvanced(ReportEvent event);
```

The item count is incremented by one.

14.15.5 Notes

The result returned by this function is an instance of `Integer`.

See section 7.6 for more information about using this function in your own reports.

See Also

`Function`.

14.16 ItemHideFunction

14.16.1 Overview

A function that hides a report item in a group if it contains the same value as the previous row.

The report item controlled by the function is specified with the `element` property. The value that is checked for repetition is specified with the `field` property.

14.16.2 Usage

In the XML report template file, you would declare an `ItemHideFunction` like this:

```
<function name="MyFunction"
          class="com.jrefinery.report.function.ItemHideFunction">
  <properties>
    <property name="element">MyElement</property>
    <property name="field">Field1</property>
  <properties>
</function>
```

The function name should be unique. You need to define two `<property>` elements:

- the first (with the name `element`) contains the name of the report item to be controlled by the function.
- the second (with the name `field`) specifies the field that determines the visibility of the report item.

14.16.3 Notes

The report item specified by this function must be in the report's *item band*.

See Also

`Function`.

14.17 ItemMaxFunction

14.17.1 Overview

A report function that returns the maximum value (so far) in the report (or group instance). The name of the column to calculate the maximum for is specified using the function's `field` property.

If the `group` property is set for the function, the maximum value is reset every time the specified group starts. Otherwise, the maximum is calculated for the entire report.

See Also

`Function`, `ItemMinFunction`.

14.18 ItemMinFunction

14.18.1 Overview

A report function that returns the minimum value (so far) in the report (or group instance). The name of the column to calculate the minimum for is specified using the function's `field` property.

If the `group` property is set for the function, the minimum value is reset every time the specified group starts. Otherwise, the minimum is calculated for the entire report.

See Also

`Function`, `ItemMaxFunction`.

14.19 ItemPercentageFunction

14.19.1 Overview

A *report function* that calculates the value of an item as a percentage of the group total for that item. This class extends the `AbstractFunction` class.

14.19.2 Usage

In the XML report template file, you might declare an `ItemPercentageFunction` like this:

```
<function name="MyFunction"
          class="com.jrefinery.report.function.ItemPercentageFunction">
  <properties>
    <property name="field">population</property>
    <property name="group">region</property>
  <properties>
</function>
```

The function name should be unique, and can be referenced by any report item used to display the value of the function. You need to define a `<property>` element (with the name `field`) that indicates the field of interest, and another `<property>` element (with the name `group`) that indicates the group.

See Also

`Function`.

14.20 ItemSumFunction

14.20.1 Overview

A report function that calculates a running total for an item in a report. The name of the item to sum is specified using the function's `field` property.

If the `group` property is set for the function, the running total is reset to zero every time the specified group starts. Otherwise, the running total accumulates for the entire report.

14.20.2 Constructors

The standard constructor:

> `public ItemSumFunction(String name);`
> Creates a named function. The `field` property must be specified before the function is used.

The default constructor:

> `public ItemSumFunction();`
> Creates an unnamed item sum function. The function is unusable until it has been given a name, and the `field` property has been set.

14.20.3 Methods

At the point when the reporting engine notifies the function that the report is starting, the running total is set to zero:

> `public void reportStarted(ReportEvent event);`
> In response, this function resets the sum to zero.

At the point when the report engine notifies the function that a new report group is starting, the function checks the group name and, if it matches the value of the `group` property, the running total is reset to zero:

> `public void groupStarted(ReportEvent event);`
> If the group name matches this function's group, reset the sum to zero.

14.20.4 Notes

The result returned by this function is an instance of `java.util.BigDecimal`.

See Also

`Function`.

14.21 LevelledExpressionList

14.21.1 Overview

A list of functions and expressions that are each assigned to a priority level.

14.22 PageFunction

14.22.1 Overview

A function that returns the current page number, either for the whole report (the default) or for a specified group within the report.

14.22.2 Parameters

The `PageFunction` class recognises two parameters:

Parameter:	**Description:**
`start`	The starting page number (defaults to “1”).
`group`	The name of the group that resets the page count (not set by default).

As with all functions, these parameters can be set using the `setProperty(...)` method inherited from the `AbstractFunction` class.

Section 7.5 describes how to make use of this function in your report template files.

14.22.3 Constructors

The standard constructor:

> `public PageFunction(String name);`
> Creates a named function that returns the current page number.

There is also a default constructor (intended mainly for use by the SAX handler):

> `public PageFunction();`
> Creates an unnamed function that returns the current page number. The function is unusable until the name has been set.

14.22.4 Methods

To obtain the current page number:

> `public Object getValue();`
> Returns the current page number. The result can be cast to an `Integer`.

14.22.5 Notes

This class extends `AbstractFunction`.

See Also

`Function`.

14.23 PageTotalFunction

14.23.1 Overview

A function that returns the current page count.

14.23.2 Constructors

The default constructor (intended mainly for use by the SAX handler that reads XML report template files):

```
public PageTotalFunction();
```

Creates an unnamed function that returns the current page count. The function is unusable until the name has been set.

14.23.3 Methods

To obtain the current page count:

```
public Object getValue();
```

Returns the current page count. The result can be cast to an `Integer`.

14.23.4 Notes

This class extends `PageFunction`.

See Also

`Function`.

14.24 PaintComponentFunction

14.24.1 Overview

A function...

14.25 PaintDynamicComponentFunction

14.25.1 Overview

A function...

14.26 ReportPropertyFunction

14.26.1 Overview

A function that returns the value of a *report property*. This class extends `AbstractFunction`.

> *THIS CLASS HAS BEEN DEPRECATED.*

All reports have the following standard properties:

Property Name:	**Description:**
`report.name`	The name of the report, as defined in the report template file.
`report.date`	A `Date` object, created at the time the report generation starts (see section 7.4).
`report.pagecount`	An `Integer` object, set to the total number of pages in the report (see section 7.5).

You can assign your own properties to a report using the `setProperty(...)` method in the `JFreeReport` class.

14.26.2 Usage

In the XML report template file, you would declare a `ReportPropertyFunction` like this:

```
<function name="MyFunction"
          class="com.jrefinery.report.function.ReportPropertyFunction">
  <properties>
    <property name="reportProperty">report.date</property>
  <properties>
</function>
```

The function name should be unique, and can be referenced by any report item used to display the value of the function. You need to define a single `<property>` element (with the name `reportProperty`) that contains the name of the report property to be returned by the function.

14.26.3 Constructors

To create a new `ReportPropertyFunction`:

> `public ReportPropertyFunction(String name, String propertyName);`
> Creates a named report property function that returns the value of a report property.

There is also a default constructor (intended mainly for use by the SAX handler):

> `public ReportPropertyFunction();`
> Creates an unnamed function with no property reference. The function is unusable until the name and property have been set.

14.26.4 Methods

To access the report property value:

> `public Object getValue();`
> Returns the value of the function (the report property).

14.26.5 Notes

The function value is read from the report's property collection when the *report started event* is fired.

See section 4.7 for an overview of the XML elements used to declare functions in a report template file.

See Also

`Function`, `JFreeReport`.

14.27 TextFormatExpression

14.27.1 Overview

A *report expression* that allows you to combined several fields (or function values), from one row of a report, into a single piece of formatted text. This expression gives you access to the features of the `java.text.MessageFormat` class within a report.

14.27.2 Usage

In the XML report template file, you might declare a `TextFormatExpression` like this:

```
<expression name="MyTextFormatExpression"
            class="com.jrefinery.report.function.TextFormatExpression">
  <properties>
    <property name="pattern"
      >Invoice for your order from {0, date, EEE, MMM d, yyyy}</property>
    <property name="0">MyReportPrintDate</property>
  <properties>
</expression>
```

The expression name should be unique, and can be referenced by any report item used to display the value of the expression. You need to define one `<property>` element (with the name `pattern`) that contains the `TextFormat` pattern to be applied by the expression. The remaining properties reference the source(s) for the values to be applied to the `TextFormat` expression.

14.27.3 Notes

This class extends the `AbstractExpression` class.

An example of the use of this expression is given in section 7.5.3.

See Also

`Expression`.

14.28 TotalGroupSumFunction

14.28.1 Overview

A *report function* that calculates the total value of one field in a report. You can specify whether the total is for the entire report, or just for a particular *report group*. This class extends the `AbstractFunction` class.

14.28.2 Notes

The `ItemPercentageFunction` class uses a `TotalGroupSumFunction` internally for calculating percentages.

See Also

`JFreeReport`.

14.29 TotalGroupSumQuotientFunction

14.29.1 Overview

A function...

15 Package: c.j.r.io

15.1 Overview

This base package contains classes responsible for reading report definitions from XML files.

15.2 Usage

The ReportGenerator class is used to parse a report definition and return a JFreeReport instance. A typical usage is:

```
JFreeReport report = null;
URL url = getClass().getResource("/com/jrefinery/report/demo/report1.xml");
ReportGenerator generator = ReportGenerator.getInstance();
try
{
  report = generator.parseReport(url);
}
catch (Exception e)
{
  ExceptionDialog.showExceptionDialog("Error on parsing",
                                      "Error while parsing " + url, e);
}
```

The XML file format is described in section 4.

15.3 InitialReportHandler

15.3.1 Overview

The *initial report handler* is the first ReportDefinitionHandler used by the Parser. Its job is to determine whether the XML report definition uses the "simple" or "extended" format.

15.3.2 Notes

This handler looks at the initial XML element tag, and handles the following cases:

- for a `<report-definition>` tag, an instance of ExtReportHandler is pushed onto the parser stack;
- for a `<report>` tag, an instance of ReportFactory is pushed onto the parser stack;
- for any other tag, a `SAXException` is thrown.

15.4 Parser

15.4.1 Overview

The *parser* is a SAX handler that manages the process of passing SAX events to various sub-handlers. Each sub-handler is responsible for parsing a particular XML element—due to the nested nature of XML elements, a *parser stack* is used to store references to the sub-handlers.

15.4.2 Notes

This class is used by the `ReportGenerator` class.

The initial sub-handler is an instance of `InitialReportHandler`, used to determine whether the XML stream is in the "simple" or "extended" format.

15.5 ParserEntityResolver

15.5.1 Overview

A *parser entity resolver* is used to point the parser to the locations of the DTDs for the "simple" and "extended" report formats.

15.6 ParserUtil

15.6.1 Overview

A collection of static utility methods used by the classes responsible for parsing report definitions.

15.7 ReportDefinitionException

15.7.1 Overview

An exception that can be thrown during the parsing of an XML report template file.

15.7.2 Notes

This exception is thrown by methods in `ReportGenerator` and `FontFactory`.

15.8 ReportDefinitionHandler

15.8.1 Overview

The *report definition handler* interface defines the methods that must be supported by the *sub-handlers* managed by the `Parser` class. These are a subset of the methods used for processing SAX events.

15.8.2 Notes

This interface is implemented by the following classes:

- `AbstractReportDefinitionHandler`;
- `BandFactory`;
- `BasicObjectHandler`;
- `BasicStyleKeyHandler`;
- `DataDefinitionHandler`;
- `ElementFactory`;
- `ElementHandler`;
- `ExpressionHandler`;
- `ExtReportHandler`;
- `FunctionFactory`;
- `FunctionsHandler`;
- `GroupFactory`;
- `GroupHandler`;
- `GroupsHandler`;
- `InitialReportHandler`;
- `ParserConfigHandler`;
- `PropertyHandler`;
- `ReportConfigHandler`;
- `ReportDescriptionHandler`;
- `ReportFactory`;
- `StylesHandler`;
- `StyleSheetHandler`;
- `TemplateHandler`;
- `TemplatesHandler`;

15.9 ReportGenerator

15.9.1 Overview

The *report generator* is a single shared object that is used for parsing report definitions.

15.9.2 Usage

To use this class, you should get hold of the single shared instance of the report generator, and pass it the URL of the report definition. For example:

```
JFreeReport report = null;
URL url = getClass().getResource("/com/jrefinery/report/demo/report1.xml");
ReportGenerator generator = ReportGenerator.getInstance();
try
{
  report = generator.parseReport(url);
}
catch (Exception e)
{
  ExceptionDialog.showExceptionDialog("Error on parsing",
                                      "Error while parsing " + url, e);
}
```

The XML file format is described in section 4.

15.9.3 Methods

To gain access to the single instance of this class:

> `public static ReportGenerator getInstance();`
> Returns a single shared instance of the `ReportGenerator` class.

To parse a report definition and create a report instance:

> `public JFreeReport parseReport(URL file)`
> `throws ReportDefinitionException, IOException;`
> Parses a report definition and returns the corresponding `JFreeReport` instance.

15.9.4 Notes

Internally, the report generator uses an instance of the `Parser` class to control the report parsing. The parser, in turn, maintains a stack of `ReportDefinition-Handler` objects that actually do the work of parsing elements in the report definition.

16 Package: c.j.r.io.ext

16.1 Overview

The base package for the classes used to implement a parser for the *extended report definition format*.

The main class in this package is the `ExtReportHandler` class, which will be installed on the parser stack by the `InitialReportHandler` if it finds that an "extended" report definition is being parsed. Once installed on the parser stack, the `ExtReportHandler` class delegates the majority of its work to other handlers in this package.

This package also features a `ReferenceDocGenerator` that will create reports that provide reference information for the extended report definition format.

16.2 Usage

Normally, you won't use the classes in this package directly. To read a report definition file, you should start with the `ReportGenerator` class in the `com.jrefinery.report.io` package.

You can write (or serialize) a `JFreeReport` object to a file using the `ReportWriter` class.

16.3 BandHandler

16.3.1 Overview

A *band handler...*

16.4 BasicObjectHandler

16.4.1 Overview

A *basic object handler* is a report definition handler that parses objects. This class implements the `ReportDefinitionHandler` interface.

16.4.2 Notes

The handler creates an `ObjectDescription` for the `Class` that it is expected to handle. Then it reads the character data from within the `<basic-object>` tags, and passes this to the `setParameter(...)` method in the `ObjectDescription` class.

16.5 BasicStyleKeyHandler

16.5.1 Overview

A *basic style key handler...*

16.6 CompoundObjectHandler

16.6.1 Overview

A *compound object handler...*

16.7 CompoundStyleKeyHandler

16.7.1 Overview

A *compound style key handler...*

16.8 DataDefinitionHandler

16.8.1 Overview

A *data definition handler...*

16.9 DataSourceHandler

16.9.1 Overview

A *data source handler...*

16.10 ElementHandler

16.10.1 Overview

An *element handler...*

16.11 ExpressionHandler

16.11.1 Overview

An *expression handler...*

16.12 ExtReportHandler

16.12.1 Overview

The *extended report handler* is used by the `Parser` to process the SAX events for the "extended" report definition format (see section **??**).

This handler examines the opening tags and delegates control to other handlers:

Tag:	**Handler:**
`<parser-config>`	`ParserConfigHandler`
`<report-config>`	`ReportConfigHandler`
`<styles>`	`StylesHandler`
`<templates>`	`TemplatesHandler`
`<functions>`	`FunctionsHandler`
`<data-definition>`	`DataDefinitionHandler`
`<report-definition>`	`ReportDescriptionHandler`

16.12.2 Notes

The `InitialReportHandler` class will push an instance of this handler onto the parser stack, if required.

16.13 FunctionsHandler

16.13.1 Overview

A *functions handler...*

16.14 GroupHandler

16.14.1 Overview

A *group handler...*

16.15 GroupsHandler

16.15.1 Overview

A *groups handler...*

16.16 ParserConfigHandler

16.16.1 Overview

A *report definition handler* that is responsible for parsing the contents of the `<parser-config>` element in the extended report definition format (see section 5.4).

16.16.2 Attributes

The following table lists the constants defined for the literal text used in the XML elements:

Attribute:	**Value:**
CLASS_ATTRIBUTE	`class`
DATADEFINITION_FACTORY_TAG	`datadefinition-factory`
DATASOURCE_FACTORY_TAG	`datasource-factory`
ELEMENT_FACTORY_TAG	`element-factory`
OBJECT_FACTORY_TAG	`object-factory`
STYLEKEY_FACTORY_TAG	`stylekey-factory`
TEMPLATE_FACTORY_TAG	`template-factory`

16.16.3 Notes

This handler is installed on the parser stack by the `ExtReportHandler`.

16.17 PropertyHandler

16.17.1 Overview

The *property handler* is responsible for parsing *(key, value)* pairs from the `<configuration>` section of the `<report-config>` element. Each pair is added to the report as a report property.

The parser will accept any key names for report properties. Those with particular significance include:

- com.jrefinery.report.preview.PreferredHeight;
- com.jrefinery.report.preview.PreferredWidth;

16.17.2 Notes

This handler is installed on the parser stack by the `ReportConfigHandler` class.

16.18 ReferenceDocGenerator

16.18.1 Overview

A *reference doc generator...*

16.19 ReportConfigHandler

16.19.1 Overview

A *report definition handler* that is responsible for parsing the contents of the `<report-config>` element in the extended report definition format (see section 5.5).

This handler is required to process the `<defaultpageformat>` tag and the `<configuration>` tag. Parsing of the latter is delegated to the `PropertyHandler` class. The handler also currently recognises—but otherwise ignores—an `<output-config>` tag.

16.19.2 Notes

This handler is installed on the parser stack by the `ExtReportHandler` class.

The `PageFormatFactory` class defines the valid values for the `pageformat` attribute.

16.20 ReportDescriptionHandler

16.20.1 Overview

A *report description handler* is responsible for parsing the bands within a report. A skeleton of a typical XML report definition file is:

```
<!-- ********************** -->
<!-- * REPORT DESCRIPTION * -->
<!-- ********************** -->
<report-description>

  <report-header name="myReportHeader">

    <!-- insert elements here -->

  </report-header>

  <report-footer name="myReportFooter">

    <!-- insert elements here -->

  </report-footer>

  <page-header name="myPageHeader">

    <!-- insert elements here -->

  </page-header>

  <page-footer name="myPageFooter">

    <!-- insert elements here -->

  </page-footer>

  <groups>

    <!-- define groups (and group headers/footers) here -->

  </groups>

  <itemband name="myItemBand">

    <!-- insert elements here -->

  </itemband>

</report-description>
```

This class is installed on the parser stack by the `ExtReportHandler` class.

16.21 StylesHandler

16.21.1 Overview

A *styles handler...*

16.22 StyleSheetHandler

16.22.1 Overview

A *style sheet handler...*

16.23 TemplateHandler

16.23.1 Overview

A *template handler...*

16.24 TemplatesHandler

16.24.1 Overview

A *templates handler...*

17 Package: c.j.r.io.ext.factory.datasource

17.1 Overview

Support classes for the extended parser. You won't normally use any of these classes directly.

17.2 AbstractDataSourceFactory

17.2.1 Overview

An abstract implementation of the `DataSourceFactory` interface. This base class is extended by the `DefaultDataSourceFactory` class.

17.2.2 Methods

To register a data source with the factory:

> `public void registerDataSources (String name, ObjectDescription o);`
> Registers a data source.

To retrieve a registered data source:

> `public ObjectDescription getDataSourceDescription(String name);`
> Returns a description for a registered data source.

17.3 DataSourceCollector

17.3.1 Overview

A *data source collector* is a `DataSourceFactory` implementation that is created from a collection of `DataSourceFactory` instances.

17.4 DataSourceFactory

17.4.1 Overview

A *data source factory* stores descriptions of all the data source types—these descriptions can be used to create new instances of a data source.

This interface extends `ClassFactory`.

17.4.2 Methods

To get a description for a registered data source:

> `public ObjectDescription getDataSourceDescription (String name);`
> Returns a description for the named data source.

To get the name of a data source:

> `public String getDataSourceName (ObjectDescription od);`
> Returns the name of the data source described by the supplied object description.

17.4.3 Notes

This interface is implemented by:

- `DataSourceCollector`
- `DefaultDataSourceFactory`

17.5 DefaultDataSourceFactory

17.5.1 Overview

A default implementation of the `DataSourceFactory` interface. This factory registers the following data sources by default:

Name:	**Class:**
DataRowDataSource	`DataRowDataSource`
DateFormatFilter	`DateFormatFilter`
DateFormatParser	`DateFormatParser`
DecimalFormatFilter	`DecimalFormatFilter`
DecimalFormatParser	`DecimalFormatParser`
EmptyDataSource	`EmptyDataSource`
FormatFilter	`FormatFilter`
FormatParser	`FormatParser`
ImageLoadFilter	`ImageLoadFilter`
ImageRefFilter	`ImageRefFilter`
NumberFormatFilter	`NumberFormatFilter`
NumberFormatParser	`NumberFormatParser`
SimpleDateFormatFilter	`SimpleDateFormatFilter`
SimpleDateFormatParser	`SimpleDateFormatParser`
StaticDataSource	`StaticDataSource`
StringFilter	`StringFilter`
URLFilter	`URLFilter`
ResourceFileFilter	`ResourceFileFilter`

Templates are also datasources:

Name:	**Class:**
DateFieldTemplate	`DateFieldTemplateDescription`
ImageFieldTemplate	`ImageFieldTemplateDescription`
ImageURLFieldTemplate	`ImageURLFieldTemplateDescription`
ImageURLElementTemplate	`ImageURLElementTemplateDescription`
LabelTemplate	`LabelTemplateDescription`
NumberFieldTemplate	`NumberFieldTemplateDescription`
StringFieldTemplate	`StringFieldTemplateDescription`
ResourceFieldTemplate	`ResourceFieldTemplateDescription`
ResourceLabelTemplate	`ResourceLabelTemplateDescription`

18 Package: c.j.r.io.ext.factory.elements

18.1 Overview

Support classes for the extended parser. You won't normally use these classes directly.

18.2 AbstractElementFactory

18.2.1 Overview

A base class for implementing the `ElementFactory` interface.

18.3 DefaultElementFactory

18.3.1 Overview

A default implementation of the `ElementFactory` interface. This factory automatically registers three element classes:

Class:	Content Type:
`TextElement`	`text/plain`
`ShapeElement`	`shape/generic`
`ImageElement`	`image/generic`

This class extends the `AbstractElementFactory` class.

18.4 ElementFactory

18.4.1 Overview

An *element factory* is used to supply new `Element` instances.

18.4.2 Methods

This interface defines a single method that returns an `Element`:

```
public Element getElementForType (String type);
```
Returns a new element of the requested type.

18.5 ElementFactoryCollector

18.5.1 Overview

An `ElementFactory` that is composed of multiple element factories.

19 Package: c.j.r.io.ext.factory.objects

19.1 Overview

A collection of object descriptions.

19.2 AbstractObjectDescription

19.2.1 Overview

An abstract base class that implements the `ObjectDescription` interface.

This base class:

- records the class name for the `Object` it describes;
- provides a storage mechanism for the parameters used to initialise new instances of the underlying object;

19.2.2 Methods

To set the name and class of a parameter for this object description:

```
public void setParameterDefinition (String name, Class obj);
```
Sets the name and class of a parameter for this object description.

To set the value of a parameter:

```
public void setParameter(String name, Object value);
```
Sets the value for the named parameter.

19.3 AlignmentObjectDescription

19.3.1 Overview

An *object description* that can be used to supply the predefined `ElementAlignment` instances.

19.3.2 Parameters

This object description used the following parameters:

Parameter:	**Class:**	**Description:**
value	`String`	The alignment type (one of "top", "bottom", "left", "right", "middle" or "center").

19.4 BandLayoutClassFactory

19.4.1 Overview

A *band layout class factory...*

19.5 BasicStrokeObjectDescription

19.5.1 Overview

An *object description* that can be used to create new instances of the `BasicStroke` class.

19.5.2 Parameters

This object description used the following parameters:

Parameter:	Class:	Description:
value	`String`	The width of the stroke.

19.6 BeanObjectDescription

19.6.1 Overview

A description of...

19.7 BooleanObjectDescription

19.7.1 Overview

An *object description* that can be used to supply instances of the `Boolean` class.

19.7.2 Parameters

This object description uses the following parameters:

Parameter:	Class:	Description:
value	`String`	The boolean value.

19.8 ByteObjectDescription

19.8.1 Overview

An *object description* that can be used to create new instances of the `Byte` class.

19.8.2 Parameters

This object description uses the following parameters:

Parameter:	Class:	Description:
value	`String`	The byte value.

19.9 CharacterObjectDescription

19.9.1 Overview

An *object description* that can be used to create new instances of the `Character` class.

19.9.2 Parameters

This object description uses the following parameters:

Parameter:	**Class:**	**Description:**
value	`String`	The character value.

19.10 ClassFactory

19.10.1 Overview

A *class factory...*

19.11 ClassFactoryCollector

19.11.1 Overview

A *class factory collector...*

19.12 ClassFactoryImpl

19.12.1 Overview

A *class factory* implementation...

19.13 ClassLoaderObjectDescription

19.13.1 Overview

An *object description* that can be used to create new instances of the `ClassLoader` class.

19.13.2 Parameters

This object description uses the following parameters:

Parameter:	**Class:**	**Description:**
class	`String`	The class name.

19.14 ColorObjectDescription

19.14.1 Overview

An *object description* that can be used to create new instances of the `Color` class.

19.14.2 Parameters

This object description uses the following parameters:

Parameter:	**Class:**	**Description:**
value	`String`	The hex or octal color value, or the color name.

19.15 DateObjectDescription

19.15.1 Overview

An *object description* that can be used to create new instances of the `Date` class.

19.15.2 Parameters

This object description uses the following parameters:

Parameter:	**Class:**	**Description:**
year	`String`	The year.
month	`String`	The month.
day	`String`	The day.

19.16 DecimalFormatObjectDescription

19.16.1 Overview

An *object description* that can be used to create new instances of the `DecimalFormat` class.

19.16.2 Parameters

This object description uses the following parameters:

19.17 DefaultClassFactory

19.17.1 Overview

A default class factory...

19.18 DoubleObjectDescription

19.18.1 Overview

A description of a `Double` object.

19.19 FloatObjectDescription

19.19.1 Overview

A description of a `Float` object.

19.20 FontDefinitionObjectDescription

19.20.1 Overview

A description of a `FontDefinition` object.

19.21 IntegerObjectDescription

19.21.1 Overview

A description of a `Integer` object.

19.22 Line2DObjectDescription

19.22.1 Overview

A description of a `Line2D` object.

19.23 LongObjectDescription

19.23.1 Overview

A description of a `Long` object.

19.24 ObjectDescription

19.24.1 Overview

The *object description* interface provides a generic description of an `Object`, with sufficient detail that fully initialised instances of the underlying object can be read from and written to XML files.

19.24.2 Notes

The following classes implement this interface:

- `AbstractObjectDescription`;
- `AlignmentObjectDescription`;
- `BasicStrokeObjectDescription`;
- `BeanObjectDescription`;
- `BooleanObjectDescription`;

- `ByteObjectDescription`;
- `CharacterObjectDescription`;
- `ClassLoaderObjectDescription`;
- `ColorObjectDescription`;
- `DateObjectDescription`;
- `DecimalFormatObjectDescription`;
- `DoubleObjectDescription`;
- `FloatObjectDescription`;
- `FontDefinitionObjectDescription`;
- `IntegerObjectDescription`;
- `Line2DObjectDescription`;
- `LongObjectDescription`;
- `Point2DObjectDescription`;
- `Rectangle2DObjectDescription`;
- `ResourceBundleObjectDescription`;
- `ShortObjectDescription`;
- `SimpleDateFormatObjectDescription`;
- `StringObjectDescription`;
- `URLObjectDescription`;

19.25 ObjectFactoryException

19.25.1 Overview

An exception...

19.26 ObjectReferenceGenerator

19.26.1 Overview

An *object reference generator*...

19.27 ObjectReferenceTableModel

19.27.1 Overview

An *object reference table model*...

19.28 Point2DObjectDescription

19.28.1 Overview

A description of a `Point2D` object.

19.29 Rectangle2DObjectDescription

19.29.1 Overview

A description of a `Rectangle2D` object.

19.30 ResourceBundleObjectDescription

19.30.1 Overview

A description of a `ResourceBundle` object.

19.31 ShortObjectDescription

19.31.1 Overview

A description of a `Short` object.

19.32 SimpleDateFormatObjectDescription

19.32.1 Overview

A description of a `SimpleDateFormat` object.

19.33 StringObjectDescription

19.33.1 Overview

A description of a `String` object.

19.34 URLClassFactory

19.34.1 Overview

A *URL class factory...*

19.35 URLObjectDescription

19.35.1 Overview

A description of a `URL` object...

20 Package: c.j.r.io.ext.factory.stylekey

20.1 Overview

Classes that provide support for the stylekey factory.

20.2 AbstractStyleKeyFactory

20.2.1 Overview

An *abstract style key factory...*

20.3 DefaultStyleKeyFactory

20.3.1 Overview

A *default style key factory...*

20.4 PageableLayoutStyleKeyFactory

20.4.1 Overview

A *pageable layout style key factory...*

20.5 StyleKeyFactory

20.5.1 Overview

A *style key factory...*

20.6 StyleKeyFactoryCollector

20.6.1 Overview

A *style key factory collector...*

20.7 StyleKeyReferenceGenerator

20.7.1 Overview

A *style key reference generator...*

20.8 StyleKeyReferenceTableModel

20.8.1 Overview

A *style key reference table model...*

21 Package: c.j.r.io.ext.factory.templates

21.1 Overview

Classes that provide support for the template factory.

21.2 AbstractTemplateDescription

21.2.1 Overview

An *abstract template description...*

21.3 DateFieldTemplateDescription

21.3.1 Overview

A *date field template description...*

21.4 DefaultTemplateCollection

21.4.1 Overview

A *default template collection...*

21.5 ImageFieldTemplateDescription

21.5.1 Overview

An *image field template description...*

21.6 ImageURLElementTemplateDescription

21.6.1 Overview

An *image URL element template description...*

21.7 ImageURLFieldTemplateDescription

21.7.1 Overview

An *image URL field template description...*

21.8 LabelTemplateDescription

21.8.1 Overview

A *label template description...*

21.9 NumberFieldTemplateDescription

21.9.1 Overview

A *number field template description...*

21.10 ResourceFieldTemplateDescription

21.10.1 Overview

A *resource field template description...*

21.11 ResourceLabelTemplateDescription

21.11.1 Overview

A *resource label template description...*

21.12 StringFieldTemplateDescription

21.12.1 Overview

A *string field template description...*

21.13 TemplateClassFactory

21.13.1 Overview

A *template class factory...*

21.14 TemplateCollection

21.14.1 Overview

A *template collection...*

21.15 TemplateCollector

21.15.1 Overview

A *template collector...*

21.16 TemplateDescription

21.16.1 Overview

A *template description...*

22 Package: c.j.r.io.ext.writer

22.1 Overview

This package contains support classes for serializing `JFreeReport` objects to a character stream using an XML-based format that we refer to as the *extended report definition format.*

The classes in this package allow you to write a report definition to a character stream. The `com.jrefinery.report.io.ext` package contains classes for reading a report definition back in from a character stream.

22.2 AbstractXMLDefinitionWriter

22.2.1 Overview

A base class for implementing writers for elements within the JFreeReport XML report format.

22.2.2 Methods

The `getSafeTags` list returns a structure that contains information about all the XML tags in the extended report definition format:

> `public static SafeTagList getSafeTags();`
> Returns a structure containing information about all the tags in the extended report definition format.

22.3 DataSourceWriter

22.3.1 Overview

A *data source writer...*

22.4 FunctionsWriter

22.4.1 Overview

This class is responsible for serializing a report's functions, expressions and "marked" properties.

22.4.2 Usage

The `ReportDefinitionWriter` class makes use of this class. You won't normally need to use it directly.

22.5 ObjectWriter

22.5.1 Overview

This class is responsible for serializing objects. To do this, it makes use of the `ObjectDescription` class corresponding to the `Object`.

22.6 ParserConfigWriter

22.6.1 Overview

This class is responsible for serializing the parser configuration to a character stream in XML format.

22.6.2 Usage

The `ReportDefinitionWriter` class makes use of this class. You won't normally need to use it directly.

22.7 ReportConfigWriter

22.7.1 Overview

This class is responsible for serializing the configuration settings for a report.

22.7.2 Usage

You won't normally use this class directly. It is used by the `ReportDefinitionWriter` class.

22.8 ReportConverter

22.8.1 Overview

A simple command line application for converting report definition files from the "simple" format to the "extended" format.

22.9 ReportConverterGUI

22.9.1 Overview

A simple GUI application that can be used to convert report definitions from the "simple" format to the "extended" format.

22.10 ReportDefinitionWriter

22.10.1 Overview

A *report definition writer* can write a `JFreeReport` object to a file using the extended report definition format. This class delegates a lot of work to other classes. After writing the header text, and the opening `<report-definition>` tag, this class hands control over to:

- a `ParserConfigWriter` to write the parser configuration;
- a `ReportConfigWriter` to write the report configuration;
- a `StylesWriter` to write the styles;
- a `TemplatesWriter` to write the templates;
- a `ReportDescriptionWriter` to write the report description (headers, footers, groups and elements);
- a `FunctionsWriter` to write the functions;

Then the report definition writer outputs the closing `</report-definition>` tag, and the report is complete.

22.10.2 Methods

To write the report to a character stream:

> `public void write (Writer w) throws ReportWriterException, IOException;`
> Writes a complete report definition (in the extended report definition format) to a character stream.

22.10.3 Notes

This class is used by the `ReportWriter` class.

22.11 ReportDescriptionWriter

22.11.1 Overview

This class is responsible for writing a large part of the extended report definition format. Everything inside the `<report-description>` element tags is handled by this class—this includes:

- the report header;
- the report footer;
- the page header;
- the page footer;
- the report groups (including headers and footers);
- the item band.

This class is utilised by the `ReportDefinitionWriter` class.

22.11.2 Methods

To write a report description to a character stream:

> `public void write(Writer writer) throws ReportWriterException, IOException`
> Writes a report description to a character stream. The description includes the report header, report footer, page header, page footer, groups (including headers and footers) and the item band.

22.12 ReportWriter

22.12.1 Overview

A *report writer* is used to serialize a `JFreeReport` object to a character stream. The report is written to the stream using the XML-based "extended" report definition format.

22.12.2 Usage

To serialize a `JFreeReport` instance to a file, you can use code similar to the following:

```
try {
    Writer stream = new OutputStreamWriter(new FileOutputStream(filename));
    ReportWriter writer = new ReportWriter(this.report);
    writer.addClassFactoryFactory(new URLClassFactory (new File(".").toURL()));
    writer.addClassFactoryFactory(new DefaultClassFactory());
    writer.addStyleKeyFactory(new DefaultStyleKeyFactory());
    writer.addStyleKeyFactory(new PageableLayoutStyleKeyFactory());
    writer.addTemplateCollection(new DefaultTemplateCollection());
    writer.addElementFactory(new DefaultElementFactory());
    writer.addDataSourceFactory(new DefaultDataSourceFactory());
    writer.write(stream);
    stream.close();
}
catch (Exception e) {
    e.printStackTrace();
}
```

22.12.3 Methods

To add a `DataSourceFactory`:

> `public void addDataSourceFactory (DataSourceFactory dsf);`
> Adds a `DataSourceFactory`.

22.12.4 Notes

The default encoding used by the writer is UTF-8.

22.13 ReportWriterException

22.13.1 Overview

A *report writer exception* is a stackable exception that can be thrown to indicate an error while writing a report in the extended report definition format.

22.14 SafeTagList

22.14.1 Overview

A utility class for storing information about the tags used in the extended report definition format. For some tags, it is safe to start a new line immediately after the tag, for others it is not.

22.14.2 Notes

The `getSafeTags()` method in the `AbstractXMLDefinitionWriter` class returns an initialised instance of this class.

22.15 StylesComparator

22.15.1 Overview

A comparator that imposes an order on two `ElementStyleSheet` objects. This comparator is used by the `StylesWriter` class.

22.16 StylesWriter

22.16.1 Overview

A *styles writer...*

22.17 StyleWriter

22.17.1 Overview

A *style writer...*

22.18 TemplatesWriter

22.18.1 Overview

This class is used by the `ReportDefinitionWriter` to serialize the templates used in a report.

In the current release, templates are not supported by the report writer, so this class simply writes out empty template tags.

23 Package: c.j.r.preview

23.1 Overview

This package contains classes relating to on-screen print preview of reports, plus dialogs for exporting reports to various file formats.

23.2 Usage

To provide an on-screen print preview for a report, you can use one of the review components:

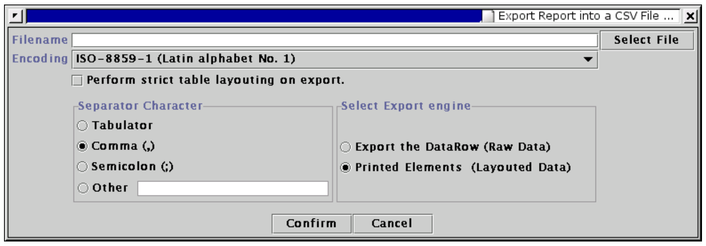
- `PreviewFrame`
- `PreviewInternalFrame`
- `PreviewDialog`

By default, the preview components will have menu options for all the supported export formats. If you don't want all the export formats, you can disable one or more of them by setting the `ReportConfiguration` for the report being previewed.

23.3 CSVExportDialog

23.3.1 Overview

An *export dialog* used to modify the settings for an export to CSV format:


Export Report into a CSV File ...

Filename

Encoding ISO-8859-1 (Latin alphabet No. 1)

☐ Perform strict table layouting on export.

Separator Character: Tabulator [✓ Comma (,)] Semicolon (;) Other

Select Export engine: Export the DataRow (Raw Data) [✓ Printed Elements (Layouted Data)]

This class implements the `ExportPlugin` interface.

23.3.2 Notes

If the user selects the "raw data" export, the `CSVProcessor` class is used to generate the output. For the "layouted data" option, the `CSVTableProcessor` class is used.

23.4 EncodingComboBoxModel

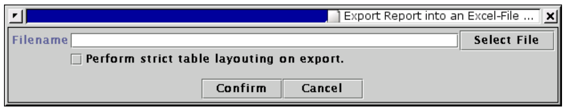
23.4.1 Overview

An *encoding combo box model*. This class stores the valid encodings for listing in a combo-box in the `CSVExportDialog`.

23.5 ExcelExportDialog

23.5.1 Overview

An *export dialog* used to modify the settings for an export to Excel format:


Export Report into an Excel-File ...

Filename

Select File

Perform strict table layouting on export.

Confirm

Cancel

If confirmed, the dialog will use the `ExcelProcessor` class to process the report and write it to Excel format.

23.5.2 Notes

This class implements the `ExportPlugin` interface.

23.6 ExportAction

23.6.1 Overview

An *export action* that is designed to work with classes that implement the `ExportPlugin` interface.

23.7 ExportPlugin

23.7.1 Overview

An *export plug in* is a class that is capable of:

- providing information to populate a menu item (to allow the user to select the plug-in);
- performing an export to a particular format;

The `ExportPluginFactory` class is used to create the plugins for a particular report, based on the plugins that have been enabled in the report configuration.

Export plug-ins are presented by the `PreviewProxyBase` class.

23.7.2 Notes

Export plugins are implemented as dialogs that can be used to customise, and then confirm or cancel, the export:

- `CSVExportDialog`
- `ExcelExportDialog`
- `HTMLExportDialog`
- `PDFSaveDialog`
- `PlainTextExportDialog`

23.8 ExportPluginFactory

23.8.1 Overview

The *export plugin factory* is used to create `ExportPlugin` instances.

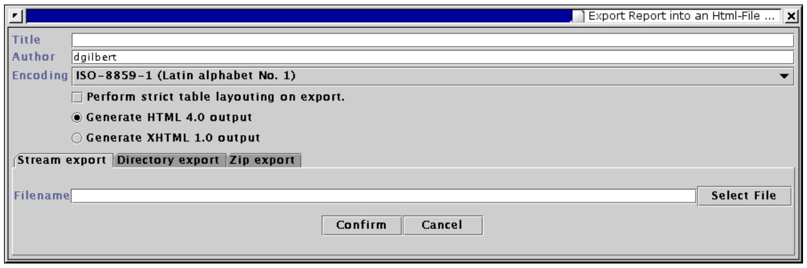
23.8.2 Notes

Recall that export plug-ins are implemented as dialogs, and that all dialogs require an owner. The report preview component (a frame, internal frame or dialog that implements the `PreviewProxy` interface) will be the owner of the dialog. One of the tasks performed by the `ExportPluginFactory` is to create plug-ins using the appropriate owner.

23.9 HtmlExportDialog

23.9.1 Overview

An *export dialog* used to modify the settings for an export to HTML format:


Export Report into an Html-File ...

Title
Author dgilbert
Encoding ISO-8859-1 (Latin alphabet No. 1)
☐ Perform strict table layouting on export.
● Generate HTML 4.0 output
○ Generate XHTML 1.0 output
Stream export | Directory export | Zip export
Filename
Select File
Confirm Cancel

23.9.2 Notes

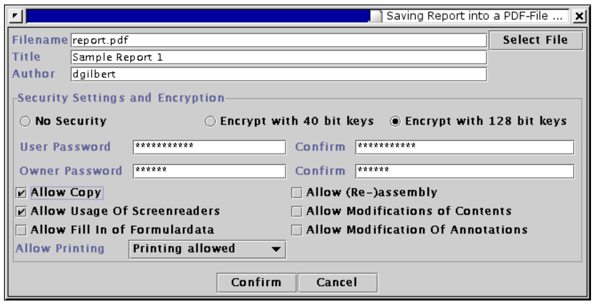
This class implements the `ExportPlugin` interface.

The `com.jrefinery.report.targets.table.html` package contains the supporting classes that actually perform the export that is initiated by this dialog.

23.10 PDFSaveDialog

23.10.1 Overview

A dialog used to specify the filename and properties for a report to be written to Acrobat PDF format.



Saving Report into a PDF-File ...

Filename: report.pdf
Title: Sample Report 1
Author: dgilbert

Security Settings and Encryption

No Security / Encrypt with 40 bit keys / [✓ Encrypt with 128 bit keys]

User Password: *********** Confirm: ***********
Owner Password: ****** Confirm: ******

- [✓] Allow Copy
- [✓] Allow Usage Of Screenreaders
- [ ] Allow Fill In of Formulardata
- [ ] Allow (Re-)assembly
- [ ] Allow Modifications of Contents
- [ ] Allow Modification Of Annotations

Allow Printing: Printing allowed

23.10.2 Usage

This dialog is automatically presented when the user selects the "Save as..." option from the print preview frame (see the PreviewFrame class).

23.10.3 Methods

To initialise the dialog's settings using the report configuration:

```
public void initFromConfiguration(ReportConfiguration config);
```

Initialises the dialog settings from the report configuration.

23.10.4 Notes

You can use this dialog to save a report in PDF format without displaying a preview frame or even the dialog itself.[6] Try the following:

```
PDFSaveDialog dialog = new PDFSaveDialog();
dialog.initFromConfiguration(report.getReportConfiguration());
dialog.setAuthor("Thomas Morgner");
dialog.setEncryptionValue(PDFOutputTarget.SECURITY_ENCRYPTION_128BIT);
dialog.setUserPassword("secret");
dialog.setOwnerPassword("topsecret");
dialog.writePDF(report, report.getDefaultPageFormat());
```

For an alternative approach to generating PDF output direct from JFreeReport, please refer to the *StraightToPDF.java* demonstration application included in the download.

See Also

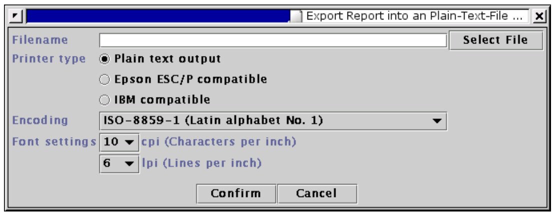
PreviewFrame.

[6]Thanks to Thomas Morgner for supplying this example via e-mail.

23.11 PlainTextExportDialog

23.11.1 Overview

An *export dialog* used to modify the settings for an export to plain text format:



Export Report into an Plain-Text-File ...

Filename

Printer type: [✓ Plain text output] Epson ESC/P compatible IBM compatible

Encoding: ISO-8859-1 (Latin alphabet No. 1)

Font settings: 10 cpi (Characters per inch)

6 lpi (Lines per inch)

23.11.2 Notes

This class implements the `ExportPlugin` interface.

23.12 PreviewDialog

23.12.1 Overview

An extension of the `JDialog` class, that is used to preview reports. This class uses the `PreviewProxyBase` class for its content pane.

23.12.2 Constructors

To construct a preview dialog:

```
public PreviewDialog(JFreeReport report) throws ReportProcessingException;
```
Creates a new report preview dialog.

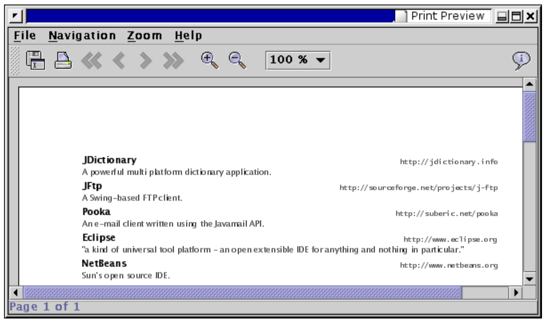
See Also

`PreviewFrame`.

23.13 PreviewFrame

23.13.1 Overview

A report preview frame (extends `JFrame`). This frame contains controls for paging forward and backward through the report, zooming in and out on a particular page, printing the report, and saving to PDF format.



JDictionary http://jdictionary.info
A powerful multi platform dictionary application.
JFtp http://sourceforge.net/projects/j-ftp
A Swing-based FTP client.
Pooka http://suberic.net/pooka
An e-mail client written using the Javamail API.
Eclipse http://www.eclipse.org
"a kind of universal tool platform – an open extensible IDE for anything and nothing in particular."
NetBeans http://www.netbeans.org
Sun's open source IDE.

This class implements the `PreviewProxy` interface.

23.13.2 Usage

Given a `JFreeReport` object, creating and displaying a preview frame is relatively straight-forward:

```
PreviewFrame frame = new PreviewFrame(myReport);
frame.pack();
frame.setVisible(true);
frame.requestFocus();
```

Since the `PreviewFrame` class extends `JFrame`, you handle the frame as you would any other Swing frame.

23.13.3 Constructors

To construct a preview frame:

> `public PreviewFrame(JFreeReport report) throws ReportProcessingException;`
> Creates a new report preview frame.

23.13.4 Methods

To control whether small (16x16) or large (24x24) icons are used in the preview frame's toolbar:

> `public void setLargeIconsEnabled(boolean b);`
> Sets a flag that controls whether or not large icons are used in the toolbar. *In fact, as currently implemented, this method has no effect since the toolbar buttons are setup when the preview frame is first constructed...this bug will be fixed in a future version of JFreeReport.*

23.13.5 Notes

Internally, the preview frame uses a `ReportPane` to display individual pages of the report.

The sample applications included with JFreeReport include many examples of this class in use.

See Also

`PreviewDialog`.

23.14 PreviewInternalFrame

23.14.1 Overview

An extension of the `JInternalFrame` class, that is used to preview reports. This class uses the `PreviewProxyBase` class for its content pane.

23.15 PreviewProxy

23.15.1 Overview

The *preview proxy* interface is implemented by the following preview components:

- `PreviewFrame`
- `PreviewInternalFrame`
- `PreviewDialog`

23.16 PreviewProxyBase

23.16.1 Overview

The *preview proxy base* is a `JComponent` that is used as the content pane for the following preview components:

- `PreviewFrame`
- `PreviewInternalFrame`
- `PreviewDialog`

The above components all implement the `PreviewProxy` interface.

23.17 ReportPane

23.17.1 Overview

A panel that can display one page of a report, with optional zooming (this class extends `JComponent`).

23.17.2 Constructors

There is just one constructor for this class:

```
public ReportPane(JFreeReport report);
```

Creates a new report pane.

23.17.3 Methods

The following method is called by Swing whenever the report pane needs repainting:

```
public void paintComponent(Graphics g);
```

Repaints the component, displaying the current page of the report.

23.17.4 Notes

The `PreviewFrame` and `PreviewDialog` classes make use of this class for displaying report pages.

24 Package: c.j.r.resources

24.1 Overview

This package contains localised resource bundles for JFreeReport.

Java defines a class naming scheme for resource bundles which allows the correct resource bundle to be loaded at runtime depending on the user's locale setting.

To date, the following resource bundles classes have been written:

Class:	**Description:**
`JFreeReportResources`	Base resources in English.
`JFreeReportResources_de`	Resources translated to German.

We are always looking for developers willing to contribute translations into other languages. Please contact us if you can help.

24.2 JFreeReportResources

24.2.1 Overview

This class contains JFreeReport resources that might require localisation.

24.2.2 Notes

There are some resources used by the `JFreeReportInfo` class:

Resource:	**Description:**
`project.name`	The name of the project.
`project.version`	The version number.
`project.info`	The URL for the project.
`project.copyright`	The copyright notice for JFreeReport.

Further resources are used by the standard actions:

Resource:	**Description:**
`action.save-as.name`	A name for the default save as action.
`action.save-as.description`	A description for the default save as action.
`action.save-as.mnemonic`	A mnemonic for the default save as action.
`action.save-as.accelerator`	An accelerator key for the default save as action.
`action.page-setup.name`	A name for the default page setup action.
`action.page-setup.description`	A description for the default page setup action
`action.page-setup.mnemonic`	A mnemonic for the default page setup action.
`action.print.name`	A name for the default print action.
`action.print.description`	A description for the default print action.
`action.print.mnemonic`	A mnemonic for the default print action.
`action.print.accelerator`	An accelerator key for the default print action.
`action.close.name`	A name for the default close action.
`action.close.description`	A description for the default close action.
`action.close.mnemonic`	A mnemonic for the default close action.
`action.about.name`	A name for the default about action.
`action.about.description`	A description for the default about action.
`action.about.mnemonic`	A mnemonic for the default about action.

Refer to Javadoc HTML files and source code for further details.

25 Package: c.j.r.states

25.1 Overview

This package contains classes representing the states in the *state transition diagram* (figure 3) used to track the processing of a report.

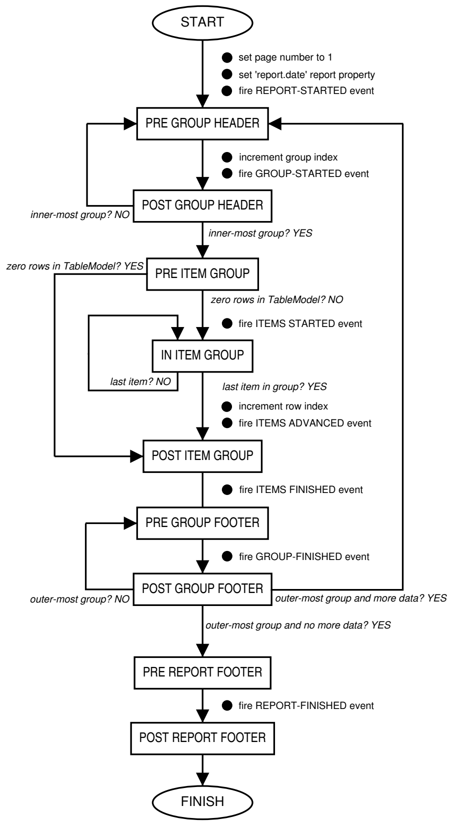


Figure 3: State transitions

If you want to know how JFreeReport processes reports internally, then you need to study and understand the state transition diagram. But for many developers, this is an implementation detail that can safely be ignored.

The classes representing the report states are:

- `StartState`;
- `PreGroupHeaderState`;

- PostGroupHeaderState;
- PreItemGroupState;
- InItemGroupState;
- PostItemGroupState;
- PreGroupFooterState;
- PostGroupFooterState;
- PreReportFooterState;
- PostReportFooterState;
- FinishState;

These classes are documented in the subsections that follow.

25.2 FinishState

25.2.1 Overview

The final state in the JFreeReport state transition diagram. Once this state is reached, the report processing is complete.

25.2.2 Constructors

To create a new "finish" state:

```
public Finish(ReportState previous);
```
Creates a new report state.

25.2.3 Methods

This class overrides the `isFinish()` method to return `true`, to indicate that this is the final state in the state transition diagram.

25.2.4 Notes

This class is "cloneable", a requirement for all `ReportState` subclasses.

25.3 InItemGroupState

25.3.1 Overview

The state for a report while processing of each row of data (from the `TableModel`) for the item group (this class extends the `ReportState` class).

25.3.2 The State Transition

Advancing to the next state involves:

- incrementing the *item index* (the row pointer for the report's data);
- firing an "items advanced" event;
- checking to see if there is more data to process in the current group—if yes, the next state is the `InItemGroupState` (this state), otherwise proceed to the `PostItemGroupState`.

This is illustrated in figure 4.

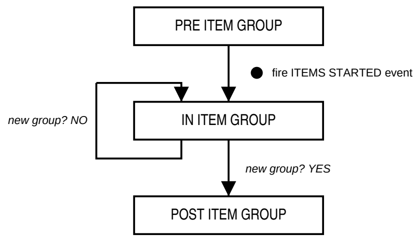


Figure 4: Transition from the "in-item-group" state

25.3.3 Constructors

To create a new "in-item-group" state:

```
public InItemGroup(ReportState previous);
```

Creates a new state.

25.3.4 Notes

This class is "cloneable", a requirement for all `ReportState` subclasses.

25.4 PostGroupFooterState

25.4.1 Overview

The state for a report immediately after the footer for a group is printed.

25.4.2 The State Transition

The transition from this state to the next is more complex than for the other states. Having just generated a group footer, there are several possibilities. If the current group is the *outer most* group (remember that JFreeReport allows nested groups), then:

- if there is more data, then a new group must be started so move to the `PreGroupHeaderState`;
- otherwise there is no more data, and all group footers have been generated, so move to the `PreReportFooterState`;

If the current group is an *inner* (or *nested*) group, then *if there is more data*:

- print the footer for the parent group;
- move to the header for the next instance of the current group;

Otherwise, there is no more data for this inner group:

- move to the `PreGroupFooterState` for the parent group (continuing to generate group footers until the outer most group is completed).

This is illustrated in figure 5.

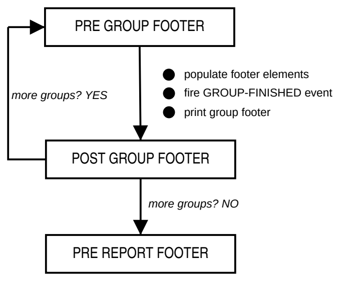


Figure 5: Transition from the "post-group-footer" state

25.4.3 Constructors

To create a new "post-group-footer" state:

```
public PostGroupFooterState(ReportState previous);
```
Creates a new report state.

25.4.4 Notes

This class is "cloneable" (a requirement for all `ReportState` subclasses).

25.5 PostGroupHeaderState

25.5.1 Overview

The state for a report immediately after the header for a group is processed (this class is an extension of the `ReportState` class).

25.5.2 The State Transition

Advancing to the next state involves determining whether or not the current group is the *item group* (the inner-most group in the report). If it is, then the next state is the `PreItemGroupState`, otherwise the next state is the `PreGroupHeaderState` because there are more group headers to process before starting on the data.

This is illustrated in figure 6.

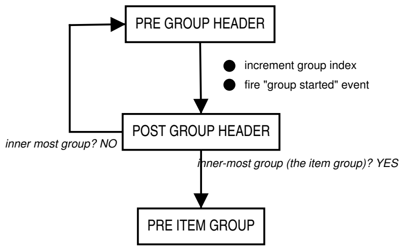


Figure 6: Transition from the "post-group-header" state

For the complete state transition diagram, refer to figure 3.

25.5.3 Constructors

To create a new "post-group-header" state:

> `public PostGroupHeaderState(ReportState previous);`
> Creates a new report state.

25.5.4 Methods

This class overrides the `isPrefetchState()` method to return `true`, indicating that any data items should be fetched from the *next* row not the current row (because the row index hasn't been incremented yet).

25.5.5 Notes

This class is "cloneable", a requirement for all `ReportState` subclasses.

25.6 PostItemGroupState

25.6.1 Overview

The state for a report immediately after one instance of the item group has been processed (this class is an extension of the `ReportState` class).

25.6.2 The State Transition

Advancing to the next state (PreGroupFooterState) involves firing an "items finished" event, as illustrated in figure 7.

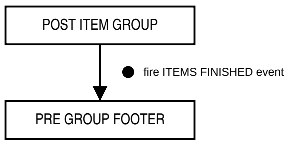


Figure 7: Transition from the "post-item-group" state

25.6.3 Constructors

To create a new "post-item-group" state:

```
public PostItemGroupState(ReportState previous);
```
Creates a new report state.

25.6.4 Notes

This class is "cloneable", a requirement for all ReportState subclasses.

25.7 PostReportFooterState

25.7.1 Overview

The state for a report immediately after the report footer is processed (this class is an extension of ReportState).

25.7.2 The State Transition

Advancing to the next state involves simply proceeding to the (FinishState).

25.7.3 Constructors

To create a new "post-report-footer" state:

```
public PostReportFooterState(ReportState previous);
```
Creates a new report state.

25.7.4 Notes

This class is "cloneable", a requirement for all ReportState subclasses.

25.8 PreGroupFooterState

25.8.1 Overview

The state for a report prior to printing a group footer (this class is an extension of `ReportState` class).

25.8.2 The State Transition

Advancing to the next state involves firing a “group finished” event and proceeding to the `PostGroupFooterState`, as illustrated in figure 8.

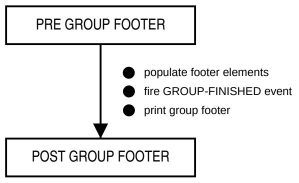


Figure 8: Transition from the “pre-group-footer” state

25.8.3 Constructors

To create a new “pre-group-footer” state:

```
public PreGroupFooterState(ReportState previous);
```
Creates a new report state.

25.8.4 Notes

This class is “cloneable”, a requirement for all `ReportState` subclasses.

25.9 PreGroupHeaderState

25.9.1 Overview

The report state immediately prior to printing a group header (this class is an extension of the `ReportState` class).

25.9.2

Advancing to the next state (the `PostGroupHeaderState`) involves:

- incrementing the group index;
- firing a “group-started” event;

This is illustrated in figure 9:
For the complete state transition diagram, refer to figure 3.

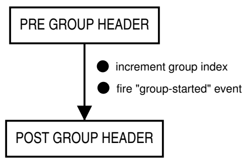


Figure 9: Transition from the "pre-group-header" state

25.9.3 Constructors

To create a new "pre-group-header" state:

> `public PreGroupHeaderState(ReportState previous);`
> Creates a new report state.

25.9.4 Methods

This class overrides the `isPrefetchState()` method to return `true`, indicating that any data items should be fetched from the *next* row not the current row (because the row index hasn't been incremented yet).

25.9.5 Notes

This class is "cloneable", a requirement for all `ReportState` subclasses.

25.10 PreItemGroupState

25.10.1 Overview

The state for a report immediately before a set of items in the inner-most group is processed (this class is an extension of the `ReportState` class).

25.10.2 The State Transition

Advancing to the next state involves:

- checking for the special case of a report with no data—in this case, jump directly to the `PostItemGroupState`;
- otherwise, generate an "items started" event and proceed to the `InItemGroupState`.

This is illustrated in figure 10.

25.10.3 Constructors

To create a new "pre-item-group" state:

> `public PreItemGroupState(ReportState previous);`
> Creates a new report state.

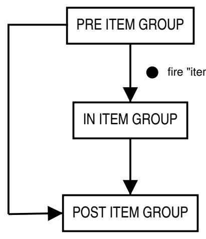


Figure 10: Transition from the "pre-item-group" state

25.10.4 Methods

This class overrides the `isPrefetchState()` method to return `true`, indicating that any data items should be fetched from the *next* row not the current row (because the row index hasn't been incremented yet).

25.10.5 Notes

This class is "cloneable", a requirement for all `ReportState` subclasses.

25.11 PreReportFooterState

25.11.1 Overview

The state for a report immediately before the report footer is to be processed (this class is an extension of `ReportState`).

25.11.2 The State Transition

Advancing to the next state involves firing a "report finished" event and proceeding to the (`PostReportFooterState`).

25.11.3 Constructors

To create a new "pre-report-footer" state:

```
public PreReportFooterState(ReportState previous);
```
Creates a new report state.

25.11.4 Notes

This class is "cloneable", a requirement for all `ReportState` subclasses.

25.12 ReportState

25.12.1 Overview

Represents the current state of a report during report processing. A range of subclasses is used represent particular states, and collectively these states are used to create the JFreeReport *state transition diagram* (see section 25.1).

25.12.2 Restarting Reports Using the Report State

A key function provided by the report state classes is the ability to "restart" report processing at some intermediate point in a report.

For example, suppose JFreeReport generates a 60 page report, and the user requests a preview of page 55—do we need to reprocess the first 54 pages of the report in order to display page 55? In theory, yes, because the report may contain values that depend on data presented earlier in the report (for example, a running total for a field).

But regenerating all the earlier pages would be too time consuming, so the solution is to add sufficient information to the report state that, given a `ReportState` object, the reporting engine can restart processing the report from any page in the report.

The report state retains information about:

- the page number;
- the group index;
- the current item (row index for the report's data);
- the report functions and expressions (including their current values);

25.12.3 Constructors

This class is a base class that is never instantiated directly. However, the constructors are available to subclasses:

> `protected ReportState(JFreeReport report);`
> Creates a new report state for the specified report.
>
> `protected ReportState(ReportState previous);`
> Creates a new report state with the same information as the previous state.

25.12.4 Methods

The state maintains a record of the current page number. You can use the `getCurrentPage()` and `setCurrentPage(int)` methods to access the page number.

The report state tracks the current *group index*. The `PreGroupHeaderState` class will call the `enterGroup()` method to increment the index:

> `public void enterGroup();`
> Enters a new group. This simply increments the current group index.

Similarly, the `PostGroupFooterState` class will call the `leaveGroup()` method to decrement the group index:

> `public void leaveGroup();`
> Leaves the current group. This simply decrements the current group index.

To get the current data item (an index into the report's `TableModel`):

> `public int getCurrentDataItem();`
> Returns the current data item. Initially, this method returns `-1`.

A related method returns the current "display" item, this being either the index of the current row or, for some states, the index of the next row. This is used by some header states to provide access to the data in the first row of the group that will be processed next:

> `public int getCurrentDisplayItem();`
> Returns the "display" data item. This is either the index of the current data item, or the index of the next data item, depending on the value returned by the `isPrefetchState()` method.

The `isPrefetch()` method controls whether the current display item is equal to the current data item plus one (`true`) or simply equal to the current data item (`false`).

25.12.5 Notes

This class is "cloneable", a feature that is used by the `ReportPane` class to cache state information at every page boundary (to enable pages to be regenerated quickly without processing all the data again).

25.13 StartState

25.13.1 Overview

The initial state for report generation (this class is an extension of `ReportState`).

25.13.2 The State Transition

Before advancing to the next state (the `PreGroupHeaderState`), the following tasks are performed:

- the `report.date` property is set to the current date/time;
- a "report-started" event is fired;

This is illustrated in figure 11.

For the complete state transition diagram, refer to figure 3.

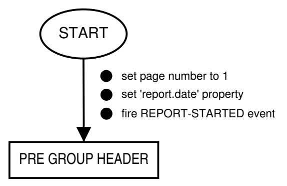


Figure 11: Transition from the “start” state

25.13.3 Constructors

To create a new “start” state:

> `public StartState(JFreeReport report);`
> Creates a new starting state for the specified report. The report (which is effectively a layout definition) is cloned before processing proceeds—this isolates the report processing from changes to the original report definition.

25.13.4 Methods

This class overrides the `isStart()` method to return `true`, indicating that this is the first state in the state transition diagram, although this is not used anywhere.

The `isPrefetchState()` method is overridden to return `true`.

25.13.5 Notes

There are four classes that create new instances of this class:

- `CSVProcessor`;
- `PageableReportProcessor`;
- `TableProcessor`;
- `XMLProcessor`;

In normal usage, you won’t need to create a new instance of this class yourself.

See Also

`ReportState`, `JFreeReport`.

26 Package: c.j.r.tablemodel

26.1 Overview

This package contains classes that assist with implementing the `TableModel` interface.

26.2 ResultSetTableModelFactory

26.2.1 Overview

A utility class for creating a `TableModel` instance from a JDBC `ResultSet` instance.

26.2.2 Methods

To get a reference to the single instance of this utility class:

```
public static ResultSetTableModelFactory getInstance();
```
Returns the single instance of this class.

To create a `TableModel` based on a JDBC `ResultSet`:

```
public CloseableTableModel createTableModel(ResultSet rs);
```
Returns a `TableModel` based on a result set. If the `ResultSet` is scrollable, this method will return a `ScrollableResultSetTableModel`. Otherwise, the data is copied to a `DefaultTableModel`.

26.3 ScrollableResultSetTableModel

26.3.1 Overview

A `TableModel` implementation that is backed by a scrollable `ResultSet`. This class implements the `CloseableTableModel` interface.

See Also

`ResultSetTableModelFactory`.

26.4 SubSetTableModel

26.4.1 Overview

A wrapper class that implements the `TableModel` interface, returning a subset of the rows of the contained `TableModel`.

26.5 TableModelInfo

26.5.1 Overview

A utility class that prints out information about a `TableModel`.

26.5.2 Methods

To print out information about a `TableModel`:

```
public static void printTableModel (TableModel mod);
```

Writes information about a `TableModel` to standard output.

27 Package: c.j.r.targets

27.1 Overview

This is the base package for the *output targets* supported by JFreeReport.

JFreeReport currently supports output to:

- the screen and printer (via the `G2OutputTarget` class);
- Acrobat PDF format (via the `PDFOutputTarget`);

Other output targets will be implemented in the future.

27.2 FloatDimension

27.2.1 Overview

A subclass of `Dimension2D` that stores its *width* and *height* attributes using `float` values.

27.2.2 Notes

This class is used by the `ElementStyleSheet` and `BandStyleSheet` classes.

28 Package: c.j.r.targets.csv

28.1 Overview

Classes that support JFreeReport output to "comma separated values" format (CSV).

28.2 CSVProcessor

28.2.1 Overview

A *report processor* that generates output in CSV format.

28.2.2 Constructor

To create a new processor:

```
public CSVProcessor(JFreeReport report, String separator,
boolean writeDataRowNames);
```

Creates a new report processor. The `report` is cloned by the constructor, and all processing is performed using the clone.

28.2.3 Notes

This class is used by the `CSVExportDialog` class.

28.3 CSVQuoter

28.3.1 Overview

To be documented...

28.4 CSVWriter

28.4.1 Overview

To be documented...

29 Package: c.j.r.targets.pageable

29.1 Overview

This package contains the “pageable” output targets.

29.2 LogicalPage

29.2.1 Overview

Represents a *logical page*.

29.2.2 Notes

For now, there is a 1-to-1 relationship between the logical page and the physical page. But a future release of JFreeReport will include the ability to distribute the content of one logical page across multiple physical pages.

See Also

`PhysicalPage`.

29.3 OutputTarget

29.3.1 Overview

An interface that defines the behaviour of a *report output target*. There are currently two targets supported by JFreeReport:

- `G2OutputTarget` - handles output to a `Graphics2D` device (includes the screen and printer);
- `PDFOutputTarget` : handles output to Acrobat PDF files.

Other targets may be implemented in the future.

29.3.2 Methods

The interface defines a range of methods relied upon by the reporting engine to generate output. The `AbstractOutputTarget` class provides default implementations for many, though obviously not all, of these methods.

It is possible to assign properties to an output target. The properties that are recognised by a particular output target are specific to each target implementation. As an example, the `PDFOutputTarget` recognises `title` and `author` properties (among others), and adds these to the document information saved in the PDF file.

To get a property that has been set for the output target:

```
public Object getProperty(String property);
```

Returns the property with the specified key.

To set a property for the output target:

> `public void setProperty(String property, Object value);`
> Sets a property for the output target.

When the reporting engine begins sending a report to an output target, it will "open" the report:

> `public void open() throws OutputTargetException;`
> Opens the output target. This provides an opportunity for the target implementation to perform any setup tasks (may do nothing).

When the reporting engine finishes sending a report to an output target, it will "close" the report:

> `public void close() throws OutputTargetException;`
> Closes the output target. This provides an opportunity for the target implementation to perform any cleanup tasks (may do nothing).

To get the font:

> `public Font getFont();`
> Returns the current font setting for the output target.

To set the font:

> `public void setFont(Font font);`
> Sets the font.

To get the paint:

> `public Paint getPaint();`
> Returns the current paint setting for the output target.

To set the paint:

> `public void setPaint(Paint paint);`
> Sets the paint.

To get the stroke:

> `public Stroke getStroke();`
> Returns the current stroke setting for the output target.

To set the stroke:

> `public void setStroke(Stroke stroke);`
> Sets the stroke.

To begin a page:

> `public void beginPage(PhysicalPage page) throws OutputTargetException;`
> Signals that a new page is starting.

To end a page:

> `public void endPage() throws OutputTargetException;`
> Signals that a page has ended.

To draw a `String` at the current cursor position:

> `public void drawString(String text);`
> Draws a string on the output target.

To draw a `Shape`:

> `public void drawShape(Shape shape);`
> Draws a shape on the output target.

To fill a `Shape`:

> `public void fillShape(Shape shape);`
> Fills a shape on the output target.

To draw an image:

> `public void drawImage(ImageReference image);`
> Draws an image on the output target. By supplying an `ImageReference`, the target has the opportunity to embed raw data if that will improve the output quality.

29.3.3 Notes

At this time there are two implementations of this interface: `G2OutputTarget` and `PDFOutputTarget`.

29.4 OutputTargetException

29.4.1 Overview

An exception that can be thrown by some of the methods in the `OutputTarget` interface.

See Also

`OutputTarget`.

29.5 PageableReportProcessor

29.5.1 Overview

A report processor.

29.5.2 Constructor

To create a report processor:

> `public PageableReportProcessor(JFreeReport report);`
> Creates a new report processor. The constructor clones the report definition for internal use.

29.5.3 Methods

To set the output target for the report processor:

> `public void setOutputTarget(OutputTarget target);`
> Sets the output target for the report processing.

To process a report (a "two-step" procedure):

> `public void processReport() throws ReportProcessingException;`
> Processes a complete report in two steps. The first pass is performed by calling the `repaginate()` method—this generates no output, but calculates values that may be required for the actual report output in the second step (e.g. total page count, field sums required for percentage calculations, and so on). The second pass actually generates the output for the report by repeatedly calling the `processPage()` method.

To "paginate" the report (that is, process the entire report and retain state information for the beginning of every page in the report):

> `public ReportStateList repaginate() throws ReportProcessingException;`
> Processes the entire report and returns a list containing the `ReportState` for the beginning of each page in the report.

To process a single page:

> `public ReportState processPage(ReportState initial, OutputTarget target);`
> Processes a single page in the report, starting with the initial state for that page. The return value is the initial state for the next page.

29.6 ReportStateList

29.6.1 Overview

A list that is used to store the `ReportState` for the beginning of each page in a report (see the `repaginate()` method in the `PageableReportProcessor` class).

An important feature of this list is that it uses "weak" references for reports that have large numbers of pages.

29.7 SizeCalculator

29.7.1 Overview

An interface for obtaining information about the dimensions of a `String`. Every `OutputTarget` can create a class that implements this interface, via the `createTextSizeCalculator()` method.

29.7.2 Methods

To calculate the width of a string:

> `public float getStringWidth(String text, int lineStartPos, int endPos);`
> Returns the width of a (sub) string.

To calculate the height of one line of text (taking into account the font's ascent, descent and leading):

```
public float getLineHeight();
```
Returns the height of a single line of text.

29.7.3 Notes

Classes that implement this interface will maintain the state information (font settings and so on) required to calculate the `String` dimensions.

See Also

OutputTarget.

29.8 Spool

29.8.1 Overview

A *spool* is a sequence of operations (instances of `PhysicalOperation`) that can be applied to an `OutputTarget`.

30 Package: c.j.r.targets.pageable.bandlayout

30.1 Overview

This package contains interfaces and classes for laying out reports in bands. For now, JFreeReport only supports the static layout that was possible in previous versions, but in the future other layout managers will be implemented.

30.2 BandLayoutManager

30.2.1 Overview

An interface that defines the methods that must be supported by a *band layout manager*.

30.2.2 Methods

To calculate the *preferred* layout size for a band:

> `public Dimension2D preferredLayoutSize(Band b);`
> Calculates the preferred layout size for a band.

To calculate the *minimum* size for a band:

> `public Dimension2D minimumLayoutSize(Band b);`
> Calculates the minimum layout size for a band.

30.3 BandLayoutManagerUtil

30.3.1 Overview

A collection of static utility methods intended for use by classes that implement the `BandLayoutManager` interface.

30.3.2 Methods

To get the layout manager for a report element:

> `public static BandLayoutManager getLayoutManager(Element e, OutputTarget ot);`
> Returns the layout manager for a report element.

To get the bounds of a report element:

> `public static Rectangle2D getBounds(Element e, Rectangle2D bounds);`
> Returns the bounds of an element. If you pass in a non-null `bounds` object, it will be updated and returned as the result (this prevents unnecessary object creation, since you can recycle objects).

To set the bounds of a report element:

> `public static void setBounds(Element e, Rectangle2D bounds);`
> Sets the bounds for a report element.

30.4 LayoutManagerCache

30.4.1 Overview

A storage container for caching information about the minimum and preferred sizes of the elements in a band.

30.4.2 Constructor

Use the default constructor to create a new cache.

30.4.3 Methods

To set the current band:

> `public void setCurrentBand(Band b);`
> Sets the current band.

To cache the minimum size for an element:

> `public void putMinSize (Element e, Dimension2D min);`
> Stores the minimum size of an element in the cache (if the element is already represented in the cache, the minimum size is updated).

To cache the preferred size for an element:

> `public void putPrefSize (Element e, Dimension2D preferred);`
> Stores the minimum size of an element in the cache (if the element is already represented in the cache, the minimum size is updated).

To clear the cache:

> `public void flushCache();`
> Clears the cache.

30.4.4 Notes

Internally, the cache uses the class `ElementCacheCarrier` to record the attributes for each element.

30.5 StaticLayoutManager

30.5.1 Overview

An implementation of the `BandLayoutManager` interface.

30.5.2 Constructor

Use the default constructor to create a new layout manager.

30.5.3 Notes

Internally, the layout manager uses a `LayoutManagerCache` to cache information about the elements in a band.

30.6 Content

30.6.1 Overview

An interface that defines an item of *report content*. The item could be text, a shape or image, or a container for other report content items.

For report content items that contain other items, this interface provides a method for accessing the *sub-content items*. First, you can determine how many sub items there are using the `getContentPartCount()` method. Then you can access individual sub items using the `getContentPart(int)` method.

30.6.2 Methods

To find the content type:

> `public ContentType getContentType();`
> Returns the content type (`TEXT`, `IMAGE`, `SHAPE` or `CONTAINER`).

To get the bounds for the content:

> `public Rectangle2D getBounds();`
> Returns the bounds for the content.

To get the minimum content size:

> `public Rectangle2D getMinimumContentSize();`
> Returns the minimum size for the content.

To get the content that fits within a particular boundary:

> `public Content getContentForBounds(Rectangle2D bounds);`
> Returns the content that fits within the given boundary.

To get the number of sub-content items:

> `public int getContentPartCount();`
> Returns the number of sub-content items. Only subclasses of `ContentContainer` will return non-zero values.

To get a sub-content item:

> `public Content getContentPart(int part);`
> Returns a sub-content item.

30.6.3 Notes

Classes that implement this interface include:

- `TextLine`;
- `ShapeContent`;
- `ImageContent`;
- `ContentContainer`;

31 Package: c.j.r.targets.pageable.operations

31.1 Overview

Operations and alignment classes.

31.2 BottomAlignment

31.2.1 Overview

A utility class that can align a `Rectangle2D` to the bottom of its current reference bounds.

31.2.2 Constructor

To create a new alignment object:

> `public BottomAlignment(Rectangle2D bounds);`
> Creates a new alignment object with the specified *reference bounds*.

31.2.3 Methods

To align a rectangle to the bottom of the reference bounds:

> `public Rectangle2D align(Rectangle2D r);`
> Returns a new rectangle that is the result of aligning `r` with the current reference bounds.

31.2.4 Notes

Note that `r` is clipped to the reference bounds *before* any alignment is performed.

See Also

`BoundsAlignment`.

31.3 BoundsAlignment

31.3.1 Overview

The base class for a collection of utility classes that can align a `Rectangle2D` instance with *reference bounds* maintained by this class. Subclasses include:

- `TopAlignment`;
- `MiddleAlignment`;
- `BottomAlignment`;

31.3.2 Methods

To align a rectangle with this object's reference bounds:

> `public abstract Rectangle2D align(Rectangle2D r);`
> Aligns a rectangle with this object's reference bounds. Different subclasses implement different alignment types.

31.4 CenterAlignment

31.4.1 Overview

Not yet documented.

31.5 HorizontalBoundsAlignment

31.5.1 Overview

Not yet documented.

31.6 ImageOperationModule

31.6.1 Overview

Not yet documented.

31.7 LeftAlignment

31.7.1 Overview

Not yet documented.

31.8 MiddleAlignment

31.8.1 Overview

A utility class that can align a `Rectangle2D` to the (vertical) middle of its current reference bounds.

31.8.2 Constructor

To create a new alignment object:

> `public MiddleAlignment(Rectangle2D bounds);`
> Creates a new alignment object with the specified *reference bounds*.

31.8.3 Methods

To align a rectangle to the middle of the reference bounds:

> `public Rectangle2D align(Rectangle2D r);`
> Returns a new rectangle that is the result of aligning `r` with the current reference bounds.

The alignment is performed in the vertical direction, not the horizontal direction.

31.8.4 Notes

Note that `r` is clipped to the reference bounds *before* any alignment is performed.

See Also

`BoundsAlignment`.

31.9 OperationFactory

31.9.1 Overview

Not yet documented.

31.10 OperationModule

31.10.1 Overview

The base class for an *operation module*.

31.10.2 Methods

To find out if this is a “generic” module:

> `public boolean isGeneric();`
> Returns `true` if this is a generic module, and `false` otherwise.

To find out if this module can handle a particular content type:

> `public boolean canHandleContentType(String contentType);`
> Returns `true` if this module can handle the specified content type, and `false` otherwise.

See Also

`ImageOperationModule`, `ShapeOperationModule`, `TextOperationModule`.

31.11 PhysicalOperation

31.11.1 Overview

The base class for operations that can be added to a `PhysicalPage` and ultimately applied to an `OutputTarget`.

31.11.2 The Operations

The following operations are defined:

- `AddComment` - adds a comment to the debug log;
- `PrintImageOperation` - prints an image within the current bounds for an output target;
- `PrintShapeOperation` - prints a shape within the current bounds for an output target;
- `PrintFilledShapeOperation` - prints a filled shape within the current bounds for an output target;
- `PrintTextOperation` - prints text within the current bounds for an output target;
- `SetBoundsOperation` - sets the bounds for an output target;
- `SetFontOperation` - sets the font for an output target;
- `SetPaintOperation` - sets the paint for an output target;
- `SetStrokeOperation` - sets the stroke for an output target;

31.11.3 Methods

All subclasses are required to implement this method:

```
public void performOperation(OutputTarget ot) throws OutputTargetException;
```
Performs the operation on the output target.

See Also

`OperationFactory`, `OperationModule`.

31.12 RightAlignment

31.12.1 Overview

Not yet documented.

31.13 ShapeOperationModule

31.13.1 Overview

An *operation module* that works with shape content.

31.13.2 Methods

To create a list of operations that will render report content within particular bounds:

> `public List createOperations(Element e, Content value, Rectangle2D bounds);`
> Returns a list of operations that draws the contents of a shape element within particular bounds.

To create a representation of the content of a report element:

> `public Content createContentForElement(Element e, Rectangle2D bounds, OutputTarget ot);`
> Creates content for an element.

31.14 TextOperationModule

31.14.1 Overview

Not yet documented.

31.15 TopAlignment

31.15.1 Overview

A utility class that can align a `Rectangle2D` to the top of its current reference bounds.

31.15.2 Constructor

To create a new alignment object:

> `public TopAlignment(Rectangle2D bounds);`
> Creates a new alignment object with the specified *reference bounds*.

31.15.3 Methods

To align a rectangle to the top of the reference bounds:

> `public Rectangle2D align(Rectangle2D r);`
> Returns a new rectangle that is the result of aligning `r` with the current reference bounds.

31.15.4 Notes

Note that `r` is clipped to the reference bounds *before* any alignment is performed.

See Also

`BoundsAlignment`.

31.16 VerticalBoundsAlignment

31.16.1 Overview

Not yet documented.

32 Package: c.j.r.targets.pageable.output

32.1 Overview

This package contains interfaces and classes for writing reports to particular *output targets*.

JFreeReport currently supports output to:

- the screen and printer (via the `G2OutputTarget` class);
- Acrobat PDF format (via the `PDFOutputTarget`);

Other output targets may be implemented in the future.

32.2 AbstractOutputTarget

32.2.1 Overview

A base class for implementing new output targets. Support is provided for attaching properties to an output target—these are used to configure the behaviour of particular output targets.

32.2.2 Constructors

To create an output target:

> `public AbstractOutputTarget(PageFormat format);`
> Creates a new output target where the logical page size and the physical page size are the same.

To create an output target where the logical page size is different to the physical page size:[7]

> `public AbstractOutputTarget(PageFormat logical, PageFormat physical);`
> Creates a new output target where the logical page size and the physical page size can be different.

To create an output target:

> `public AbstractOutputTarget(LogicalPage logicalPage);`
> Creates a new output target with the specified logical page.

32.2.3 Methods

To get the value of a property that is attached to the output target:

> `public Object getProperty(String property);`
> Returns the value of the property with the given key, or `null` if there is no such property.

[7] Implementation of this feature is incomplete.

Alternatively, you can specify a default value to be returned when the property is not defined:

> `public Object getProperty(String property, Object default);`
> Returns the value of the property with the given key, or `default` if there is no such property.

To set a property:

> `public void setProperty(String property, Object value);`
> Attaches a property to the output target. If `value` is `null`, the existing property is removed.

To get the logical page defined for the output target:

> `public LogicalPage getLogicalPage();`
> Returns the logical page for the output target.

To create a default layout manager for the output target:

> `public BandLayoutManager getDefaultLayoutManager();`
> Creates and returns a default layout manager for the output target.

32.2.4 Notes

Subclasses include: `G2OutputTarget` and `PDFOutputTarget`.

See Also

`OutputTarget`.

32.3 DummyOutputTarget

32.3.1 Overview

A wrapper for an `OutputTarget` that suppresses any real output. This allows a report to be processed for layout purposes before it is actually displayed.

32.4 EpsonPrinterCommandSet

32.4.1 Overview

Implements the command set for Epson ESC/P compatible printers. This class is a subclass of `PrinterCommandSet`.

32.5 G2OutputTarget

32.5.1 Overview

An output target that allows reports to be sent to the screen, to a printer, or to any other device that implements `Graphics2D` support. This class extends `AbstractOutputTarget`, and implements the `OutputTarget` interface.

32.5.2 Constructors

The constructor requires a `Graphics2D` instance and a `PageFormat` object:

```
public G2OutputTarget(Graphics2D g2, PageFormat pageFormat);
```
Creates a new output target using a `Graphics2D` instance.

32.5.3 Methods

This class implements all the methods defined in the `OutputTarget` interface.

32.5.4 Notes

This class is used by the `ReportPane` class to display reports on the screen.

See Also

`OutputTarget`, `PDFOutputTarget`.

32.6 IBMPrinterCommandSet

32.6.1 Overview

Implements the command set for IBM compatible printers.

32.7 PDFOutputTarget

32.7.1 Overview

An output target that generates a stream in Acrobat PDF format.

32.7.2 Constructors

To construct a `PDFOutputTarget`:

```
public PDFOutputTarget(OutputStream out, PageFormat pageFormat,
boolean embedFonts);
```
Creates a new `PDFOutputTarget`.

32.7.3 Methods

This class implements all the methods in the `OutputTarget` interface.

32.7.4 Notes

This class utilises the iText library written by Bruno Lowagie, Paulo Soares and others.[8] You can find out more about iText at:

```
http://www.lowagie.com/iText
```

Note that the URL is case-sensitive!

[8] iText is free software under the terms of the GNU Lesser General Public Licence.

See Also

`OutputTarget`, `G2OutputTarget`.

32.8 PlainTextOutputTarget

32.8.1 Overview

An `OutputTarget` that generates plain text. Non-text report elements are ignored.

32.9 PlainTextPage

32.9.1 Overview

A plain text page is used to buffer the contents of one page, and write the buffered data when the page is closed.

32.10 PrinterCommandSet

32.10.1 Overview

The printer command set for plain text output.

32.10.2 Notes

Subclasses include `EpsonPrinterCommandSet` and `IBMPrinterCommandSet`.

33 Package: c.j.r.targets.pageable.pagelayout

33.1 Overview

Page layout classes.

33.2 ElementChooserAgent

33.2.1 Overview

Not yet documented.

33.3 EventType

33.3.1 Overview

Not yet documented.

33.4 FlowPageLayouter

33.4.1 Overview

Not yet documented.

33.5 LayoutAgent

33.5.1 Overview

Not yet documented.

33.6 LayoutAgentProgress

33.6.1 Overview

Not yet documented.

33.7 LayoutTask

33.7.1 Overview

Not yet documented.

33.8 PageLayouter

33.8.1 Overview

Not yet documented.

33.9 SimplePageLayoutCursor

33.9.1 Overview

Not yet documented.

33.10 SimplePageLayouter

33.10.1 Overview

Not yet documented.

34 Package: c.j.r.targets.pageable.physicals

34.1 Overview

Support classes for mapping logical pages to physical pages.

34.2 LogicalPageImpl

34.2.1 Overview

A simple implementation of the `LogicalPage` interface.

34.2.2 Notes

The current implementation does not distribute content over multiple physical pages.

34.3 PhysicalPage

34.3.1 Overview

Represents a *physical page*.

34.3.2 Methods

To add an operation to the page:

```
public void addOperation(PhysicalOperation op);
```

Adds an operation to the page.

35 Package: c.j.r.targets.style

35.1 Overview

This package contains interfaces and classes for the *style-sheets* that control the appearance of report elements.

35.2 BandDefaultStyleSheet

35.2.1 Overview

This class defines the default style sheet for all bands. To access the single instance of the default style sheet:

```
public static final BandDefaultStyleSheet getBandDefaultStyle();
```

Returns the default style sheet shared by all bands.

35.2.2 Notes

The default values provided by the style-sheet are:

Key:	**Default Value:**
`MINIMUM_SIZE`	`new FloatDimension(0, 0);`
`MAXIMUM_SIZE`	`new FloatDimension(Short.MAX_VALUE, Short.MAX_VALUE);`
`BOUNDS`	`new Rectangle.Float();`
`PAGE_BREAK_BEFORE`	`Boolean.FALSE;`
`PAGE_BREAK_AFTER`	`Boolean.FALSE;`
`DISPLAY_ON_FIRSTPAGE`	`Boolean.TRUE;`
`DISPLAY_ON_LASTPAGE`	`Boolean.TRUE;`
`ABSOLUTE_DIM`	`new FloatDimension(-100.0, -100.0);`
`ABSOLUTE_POS`	`new FloatDimension(0, 0);`

See Also

`Band`, `BandStyleSheet`.

35.3 BandStyleSheet

35.3.1 Overview

A style-sheet that controls the settings for a band as well as the default appearance for the elements within the band. This class extends the `ElementStyleSheet` class.

35.3.2 Notes

The keys defined for the band style sheet include:

Key:	Description:
PAGE_BREAK_BEFORE	A flag that controls whether or not to force a page break *before* generating the header.
PAGE_BREAK_AFTER	A flag that controls whether or not to force a page break *after* generating the header.
DISPLAY_ON_FIRSTPAGE	A flag that controls whether or not the band is displayed on the *first* page of the report (intended for the page header and/or page footer).
DISPLAY_ON_LASTPAGE	A flag that controls whether or not the band is displayed on the *last* page of the report (intended for the page header and footer)
REPEAT_HEADER	A flag that controls whether or not the band is repeated at the beginning of each new page (intended for group headers).

See Also

Band, BandDefaultStyleSheet.

35.4 ElementDefaultStyleSheet

35.4.1 Overview

This class defines the default style sheet for all elements. To access the single instance of the default style sheet:

```
public static final ElementStyleSheet getDefaultStyle();
```
Returns the default style sheet shared by all elements.

35.4.2 Notes

The default values provided by the style-sheet are:

Key:	Default Value:
FONT	new Font("Serif", Font.PLAIN, 10);
PAINT	Color.black;
MINIMUMSIZE	new FloatDimension(0, 0);
MAXIMUMSIZE	new FloatDimension(Short.MAX_VALUE, Short.MAX_VALUE);
BOUNDS	new Rectangle2D.Float();
ALIGNMENT	ElementAlignment.LEFT;
VALIGNMENT	ElementAlignment.BOTTOM;
VISIBLE	Boolean.TRUE;

See Also

Element, ElementStyleSheet.

35.5 ElementStyleSheet

35.5.1 Overview

An *element style-sheet* contains zero, one or many attributes that affect the appearance of report elements:

- every report element has an associated style-sheet—see the `getStyle()` method in the `Element` class;
- for each attribute in the style-sheet, there is a predefined key that can be used to access that attribute—see below for details;
- a style-sheet maintains a list of parent style-sheets. If an attribute is not defined in a style-sheet, the code refers to the parent style-sheets to see if the attribute is defined there.

35.5.2 The Keys

The keys defined for an element style-sheet are:

Key:	**Default Value:**
`FONT`	The font name.
`FONTSIZE`	The font size.
`BOLD`	Bold font attribute.
`ITALIC`	Italic font attribute.
`UNDERLINE`	Underline font attribute.
`STRIKETHROUGH`	Strikethrough font attribute.
`PAINT`	The foreground paint.
`MINIMUMSIZE`	The element's minimum size.
`MAXIMUMSIZE`	The element's maximum size.
`PREFERREDSIZE`	The element's preferred size.
`BOUNDS`	The element's bounds.
`ALIGNMENT`	The horizontal alignment.
`VALIGNMENT`	The vertical alignment.
`VISIBLE`	A flag that controls whether or not the element is visible.
`SCALE`	A flag that controls whether or not images are scaled to fit the current bounds.
`KEEP_ASPECT_RATIO`	A flag that controls whether scaled images retain their original aspect ratio.

Each key is an instance of the `StyleKey` class.

35.5.3 Constructor

To create a new element style-sheet:

```
public ElementStyleSheet (String name);
```
Creates a new style-sheet with the given name (`null` not permitted).

35.5.4 Methods

To access a style attribute:

> `public Object getStyleProperty(StyleKey key);`
> Returns the value of the style attribute with the given key. If the key is not found (in this style sheet or any of the parent style sheets) the method returns `null`.

Another method for accessing a style attribute allows you to specify a default value for the case where the attribute is not found:

> `public Object getStyleProperty(StyleKey key, Object defaultValue);`
> Returns the value of the style attribute with the given key. If the key is not found (in this style sheet or any of the parent style sheets) the method returns `defaultValue`.

To set a style attribute:

> `public void setStyleProperty (StyleKey key, Object value);`
> Sets an attribute value within the style sheet. If the `value` is `null`, the attribute is removed from the style-sheet.

To add a parent style-sheet:

> `public void addParent(ElementStyleSheet parent);`
> Adds a parent style-sheet. You can add multiple parent style-sheets if you want to—when accessing a style attribute, they will be searched in the order that they were added.

To remove a parent style-sheet:

> `public void removeParent(ElementStyleSheet parent);`
> Removes the parent style-sheet.

To get a list of the parent style-sheets:

> `public List getParents ();`
> Returns a list of the parent style-sheets. The list may be empty.

35.5.5 Notes

See Also

`StyleKey`, `ElementDefaultStyleSheet`.

35.6 StyleKey

35.6.1 Overview

A *style key* is a key used to access a *style attribute* in a *style-sheet*. JFreeReport uses two style-sheet types:

- `ElementStyleSheet` – to control the appearance of individual elements;
- `BandStyleSheet` – to control the appearance of a report band.

The keys used by JFreeReport are defined as constants in the above style-sheet classes.

35.7 StyleSheet

35.7.1 Overview

To be documented.

36 Package: c.j.r.targets.support

36.1 Overview

This package contains support classes for the JFreeReport class library.

36.2 ReportProcessorUtil

36.2.1 Overview

A utility class that contains static methods for exporting reports to various formats. These are convenience methods that cover the typical usage—no customisation is possible. If you require greater control over the export, you should use the appropriate *report processor* class directly.

36.2.2 Methods

To export a report to Acrobat PDF format:

> `public static boolean createPDF(JFreeReport report, String fileName);`
> Exports the report to the specified file in Acrobat PDF format (using the iText class library).

To export a report in Microsoft Excel format:

> `public static void createXLS (JFreeReport report, String filename) throws IOException, FunctionInitializeException, ReportProcessingException;`
> Exports the report to the specified file in Microsoft Excel format (using the Apache POI class library).

To export a report in plain text format:

> `public static void createPlainText (JFreeReport report, String filename) throws IOException, ReportProcessingException, FunctionInitializeException, OutputTargetException;`
> Exports the report to the specified file in plain text format.

To export a report in “comma separated value” (CSV) format:

> `public static void createCSV (JFreeReport report, String filename) throws ReportProcessingException, FunctionInitializeException, IOException;`
> Exports the report to the specified file in CSV format.

To export a report in HTML format:

> `public static void createStreamHTML (JFreeReport report, String filename) throws IOException, FunctionInitializeException, ReportProcessingException;`
> Exports the report to the specified file in HTML format.

To export a report in HTML format, using one HTML file per page:

> `public static void createDirectoryHTML (JFreeReport report,`
> `String filename) throws IOException,`
> `ReportProcessingException, FunctionInitializeException;`
> Exports the report to the specified directory in HTML format (one file per page).

To export a report to HTML format, using one HTML file per page, and storing all files in a ZIP archive:

> `public static void createZIPHTML (JFreeReport report,`
> `String filename) throws IOException,`
> `ReportProcessingException, FunctionInitializeException;`
> Exports the report to the specified ZIP archive in HTML format (one HTML file per page).

To export a report in RTF format:

> `public static void createRTF (JFreeReport report, String filename)`
> `throws IOException,`
> `ReportProcessingException, FunctionInitializeException;`
> Exports the report to the specified file in RTF format.

36.2.3 Notes

For some output formats, an alternative to using this class is to use the corresponding *report processor* directly:

- `ExcelProcessor`
- `CSVTableProcessor`
- `HtmlProcessor`
- `RTFProcessor`

37 Package: c.j.r.targets.support.itext

37.1 Overview

Support classes for the iText library.

37.2 BaseFontFactory

37.2.1 Overview

A utility class that is used to find and register all *TrueType* fonts for possible embedding in the PDF file.

37.3 BaseFontRecord

37.3.1 Overview

A PDF font record.

37.4 BaseFontRecordKey

37.4.1 Overview

A PDF font record key.

37.5 BaseFontSupport

37.5.1 Overview

A utility class for iText font support.

38 Package: c.j.r.targets.table

38.1 Overview

The base package for table-based output.

38.2 AbstractTableCellDataFactory

38.2.1 Overview

An abstract base class for subclasses that implement the `TableCellDataFactory` interface. Known subclasses include:

- `ExcelCellDataFactory`
- `HtmlCellDataFactory`
- `RTFCellDataFactory`

38.2.2 Methods

This method returns a cell for the band area:

> `public TableCellData createBandCell(Element e, Rectangle2D rect);`
> Returns a cell (`TableBandArea`) for the band area.

This method creates background cells from shape elements in a report:

> `public TableCellBackground createBackground (Element e, Shape shape, Rectangle2D bounds);`
> Converts lines (horizontal and vertical only) and rectangles into background cells.

38.3 TableBandArea

38.3.1 Overview

A simple extension of the `TableCellBackground` class.

38.4 TableCellBackground

38.4.1 Overview

This class encapsulates the attributes that control the background formatting for a cell, including the borders and background color. It extends the `TableCellData` class.

38.5 TableCellData

38.5.1 Overview

The base class for table cell data. This class defines the cell's outer bounds, used by the `TableGrid` class to calculate grid boundaries.

38.5.2 Notes

The `TableCellBackground` class is a subclass of this class.

38.6 TableCellDataFactory

38.6.1 Overview

An interface for creating table cell data from report elements. This interface is implemented by the `AbstractTableCellDataFactory` class.

38.6.2 Methods

This interface defines a single method that creates a `TableCellData` instance from an `Element`:

```
public TableCellData createCellData (Element e, Rectangle2D rect);
```
Converts a report element into table data.

38.7 TableGrid

38.7.1 Overview

A class that works out grid boundaries as `TableCellData` items are added.

38.7.2 Methods

To add a new item to the grid:

```
public void addData (TableCellData pos);
```
Adds an item to the grid. The upper left corner of the item is used to create new horizontal and vertical boundaries in the grid. If the *strict* flag is set, then the lower right corner of the item is also used to create new horizontal and vertical boundaries in the grid.

Once all items have been added to the grid, a `TableGridLayout` can be created:

```
public TableGridLayout performLayout();
```
Creates a table grid layout.

38.8 TableGridLayout

38.8.1 Overview

To be documented...

38.9 TableGridPosition

38.9.1 Overview

A carrier for a `TableCellData` instance that maintains row and column indices, plus row and column spans.

38.9.2 Notes

This class is used by the `TableGridLayout` class.

38.10 TableProcessor

38.10.1 Overview

A base class for implementing a *report processor* that generates output in a tabular form. Subclasses include:

- `CSVTableProcessor`
- `ExcelProcessor`
- `HTMLProcessor`
- `RTFProcessor`

All reports exported to tabular formats have an option to use *strict* layouting. In this mode, extra cells are added to the table to emulate the layout of the printed report. Of course, this results in a more complex table being created.

38.10.2 Constructor

To create a new processor:

```
public TableProcessor(JFreeReport report);
```

Creates a new report processor. The `report` is cloned by the constructor, and all processing is performed using the clone. The *strict* flag is initialised from the setting in the report's configuration.

38.10.3 Methods

The following method must be overridden by subclasses:

```
protected abstract TableProducer createProducer (boolean dummy);
```

Creates a producer for the table.

38.11 TableProducer

38.11.1 Overview

The base class for implementing table producers. The following subclasses have been implemented:

- `CSVTableProducer`
- `ExcelProducer`
- `HtmlProducer`
- `RTFProducer`

38.12 TableWriter

38.12.1 Overview

A *table writer* creates the content for the exported report.

Extends the `AbstractFunction` class.

38.13 TableWriterCursor

38.13.1 Overview

A cursor for tracking the current position when writing table output (used by the `TableWriter` class).

39 Package: c.j.r.targets.table.csv

39.1 Overview

This package contains classes that provide support for exporting reports to CSV format.

39.2 Usage

39.2.1 Via Print Preview

By default, the on-screen print preview facilities will provide an option to export to CSV format. If the user selects this option, an `CSVExportDialog` is used to initiate (or cancel) the export. That class will take care of the report processing, using classes from this package.

39.2.2 Using the CSV Processor Directly

You can send a report straight to an CSV file (without displaying a preview of the report first), by using the `CSVProcessor` class:

```
CSVProcessor pr = new CSVProcessor(report);
pr.setStrictLayout(false);
OutputStream fout = new BufferedOutputStream(new FileOutputStream(filename));
pr.setOutputStream(fout);
pr.processReport();
fout.close();
```

See the `StraightToEverything.java` demo in the distribution for an example.

39.3 CSVCellData

39.3.1 Overview

A cell containing data (a `String`) for CSV output. This class extends the `TableCellData` class.

39.4 CSVCellDataFactory

39.4.1 Overview

A class that handles the conversion of report elements into `TableCellData` instances. In this case, only elements containing `String` values are converted—they are returned as `CSVCellData` instances.

39.5 CSVTableProcessor

39.5.1 Overview

An extension of the `TableProcessor` class for generating CSV output.

39.5.2 Notes

For data oriented output, see `CSVProcessor`.

39.6 CSVTableProducer

39.6.1 Overview

An extension of the `TableProducer` class, used to create CSV output.

40 Package: c.j.r.targets.table.excel

40.1 Overview

This package contains classes that support the export of reports to Excel format. JFreeReport uses the Jakarta POI library (version 1.5.1) to write files in Excel format.

40.2 Usage

40.2.1 Via Print Preview

By default, the on-screen print preview facilities will provide an option to export to Excel format. If the user selects this option, an `ExcelExportDialog` is used to initiate (or cancel) the export. That class will take care of the report processing, using classes from this package.

40.2.2 Using the Excel Processor Directly

You can send a report straight to an Excel file (without displaying a preview of the report first), by using the `ExcelProcessor` class:

```
ExcelProcessor pr = new ExcelProcessor(report);
pr.setStrictLayout(false);
OutputStream fout = new BufferedOutputStream(new FileOutputStream(filename));
pr.setOutputStream(fout);
pr.processReport();
fout.close();
```

Alternatively, you can use the `createXLS(...)` method in the `ReportProcessorUtil` class.

40.3 DateExcelCellData

40.3.1 Overview

A cell containing a date. This class is a subclass of `ExcelCellData`.

40.4 DefaultExcelCellData

40.4.1 Overview

A cell containing a `String`. This class is a subclass of `ExcelCellData`.

40.5 ExcelCellData

40.5.1 Overview

An abstract base class for all Excel cells. Extends the `TableCellData` class.

40.5.2 Notes

Style information is recorded in an `ExcelDataCellStyle` instance.

40.6 ExcelCellDataFactory

40.6.1 Overview

This class is used to create `ExcelCellData` instances from report elements.

40.6.2 Notes

Internally, an `ExcelCellStyleFactory` is used to convert element styles, with the ability to reuse existing styles.

40.7 ExcelCellStyleFactory

40.7.1 Overview

A class that creates `ExcelCellDataStyle` instances for report elements. The class also retains information about the styles already created, to enable reuse...this is because Excel has a limitation on the number of styles that can be used.

40.8 ExcelDataCellStyle

40.8.1 Overview

This class encapsulates information about a cell style.

40.9 ExcelFontFactory

40.9.1 Overview

This class keeps track of the fonts that have been used in the Excel file so far.

40.10 ExcelProcessor

40.10.1 Overview

A report processor that outputs a report in Excel format. This class extends the `TableProcessor` class.

40.10.2 Methods

To create a new table producer:

```
public TableProducer createProducer(boolean dummy);
```

Creates a new producer. If `dummy` is `true`, the producer will be initialised with a dummy output stream (no real output is generated).

40.10.3 Notes

The Jakarta POI library is used to write files in Excel format.

40.11 ExcelProducer

40.11.1 Overview

The class that is responsible for producing an Excel spreadsheet for the exported report. This class is a subclass of `TableProducer`.

40.12 ExcelToolLibrary

40.12.1 Overview

A class containing utility methods for working with Excel.

40.12.2 Methods

To find the index of the nearest Excel color:

> `public static short getNearestColor(Color awtColor);`
> Returns the Excel index of the color that is the closest match to `awtColor`.

40.13 HSSFFontWrapper

40.13.1 Overview

This class is used to carry Excel style font information.

40.14 NumericExcelCellData

40.14.1 Overview

A cell containing a number with an associated format string. This class is a subclass of `ExcelCellData`.

41 Package: c.j.r.targets.table.html

41.1 Overview

This package contains classes that provide support for HTML and XHTML output.

41.2 Usage

41.2.1 Via Print Preview

By default, the on screen print preview facilities will provide an option to export reports to HTML format. If the user selects this option, an `HtmlExportDialog` is used to initiate (or cancel) the export. That class will take care of report processing, using classes from this package.

41.2.2 Using the HTML Processor Directly

You can export a report directly to HTML using the `HtmlProcessor` class. For example, to send the output to a single file:

```
HtmlProcessor pr = new HtmlProcessor(report);
pr.setStrictLayout(false);
OutputStream fout = new BufferedOutputStream(new FileOutputStream(filename));
pr.setFilesystem(new StreamHtmlFilesystem (fout));
pr.processReport();
fout.close();
```

To send the output to a directory containing one HTML file per report page:

```
HtmlProcessor pr = new HtmlProcessor(report);
pr.setFilesystem(new DirectoryHtmlFilesystem (new File(filename)));
pr.processReport();
```

And finally, to create a ZIP file containing the HTML files:

```
HtmlProcessor pr = new HtmlProcessor(report);
OutputStream fout = new BufferedOutputStream(new FileOutputStream(filename));
pr.setFilesystem(new ZIPHtmlFilesystem (fout, "data"));
pr.processReport();
fout.close();
```

41.3 DirectoryHtmlFilesytem

41.3.1 Overview

An implementation of the `HtmlFilesystem` interface that writes output to multiple files within a single directory.

41.4 EmptyContentHtmlReferenceData

41.4.1 Overview

Represents an empty HTML reference. Extends `HtmlReferenceData`.

41.5 HRefReferenceData

41.5.1 Overview

An *href* reference.

41.6 HtmlCellData

41.6.1 Overview

The base class for all HTML cell content.

41.6.2 Notes

Subclasses include:

- `HtmlImageCellData`

The `HtmlCellDataFactory` class converts report items into instances of (subclasses of) this class.

41.7 HtmlCellDataFactory

41.7.1 Overview

This class is responsible for translating report items into HTML cell content. Extends `AbstractTableCellDataFactory`.

41.8 HtmlCellStyle

41.8.1 Overview

This class encapsulates style information for an HTML cell.

41.8.2 Constructor

To create a new instance:

```
public HtmlCellStyle(FontDefinition font, Color fontColor,
ElementAlignment verticalAlignment, ElementAlignment horizontalAlignment);
```
Creates a new cell style instance.

41.9 HtmlFilesytem

41.9.1 Overview

An interface for the output targets supported by the `HtmlProducer` class. These include:

- `DirectoryHtmlFilesystem`

- `StreamHtmlFilesystem`
- `ZIPHtmlFilesystem`

41.9.2 Methods

The *root stream* is the output stream used to write the main HTML file. If content is generated in other files, it will be referenced from the main file.

> `public OutputStream getRootStream () throws IOException;`
> Returns the root stream, used to write the main HTML file.

To create an image reference:

> `public HtmlReferenceData createImageReference(ImageReference reference) throws IOException;`
> Creates an image reference.

To create a CSS reference:

> `public HtmlReferenceData createCSSReference (String styleSheet) throws IOException;`
> Creates a CSS reference.

To close the “filesystem”:

> `public void close() throws IOException;`
> Closes the filesystem.

41.10 HtmlImageCellData

41.10.1 Overview

An HTML cell containing an image. Extends `HtmlCellData`.

41.11 HtmlProcessor

41.11.1 Overview

This class manages report processing for the purpose of generating HTML output.

For usage information, refer to section 41.2.2.

41.12 HtmlProducer

41.12.1 Overview

Extends `TableProducer`.

41.13 HtmlReferenceData

41.13.1 Overview

The base class for HTML references to external items such as images and stylesheets. Subclasses include:

- `ImageReferenceData`
- `InternalCSSReferenceData`

41.14 HtmlStyleCollection

41.14.1 Overview

A utility class for creating and storing `HtmlCellStyle` objects, reusing existing instances where possible.

41.15 HtmlTextCellData

41.15.1 Overview

An HTML cell containing text. Extends `HtmlCellData`.

41.16 ImageReferenceData

41.16.1 Overview

This class generates an HTML reference to an image file. Extends `HtmlReferenceData`.

41.17 InternalCSSReferenceData

41.17.1 Overview

Extends `HtmlReferenceData`.

41.18 StreamHtmlFilesystem

41.18.1 Overview

An implementation of the `HtmlFilesystem` interface that writes all output to a single stream (file).

41.19 ZipHtmlFilesytem

41.19.1 Overview

An implementation of the `HtmlFilesystem` interface that writes all output to a zip file.

42 Package: c.j.r.targets.table.rtf

42.1 Overview

This package contains classes that provide support for exporting reports to "rich text format" (RTF) files.

42.2 Usage

To export a report to RTF format, you can use code like this:

```
RTFProcessor pr = new RTFProcessor(report);
pr.setStrictLayout(false);
OutputStream fout = new BufferedOutputStream(new FileOutputStream(filename));
pr.setOutputStream(fout);
pr.processReport();
fout.close();
```

42.3 RTFCellData

42.3.1 Overview

The base class for representing RTF cell data (extends `TableCellData`). The cell carries style information represented by an instance of `RTFCellStyle`.

42.3.2 Notes

Subclasses include:

- `RTFImageCellData`
- `RTFTextCellData`

42.4 RTFCellDataFactory

42.4.1 Overview

This class is responsible for converting report items into instances of `RTFCellData`.

42.5 RTFCellStyle

42.5.1 Overview

Contains cell style information.

42.5.2 Constructor

To create a new style instance:

```
public RTFCellStyle(FontDefinition font, Color fontColor,
ElementAlignment verticalAlignment, ElementAlignment horizontalAlignment);
```

Creates a new style instance.

42.6 RTFImageCellData

42.6.1 Overview

A cell containing an image.

42.7 RTFProcessor

42.7.1 Overview

A *report processor* that coordinates the process of exporting a report to RTF format.

42.8 RTFProducer

42.8.1 Overview

Extends `TableProducer`.

42.9 RTFTextCellData

42.9.1 Overview

A text cell (extends `RTFCellData`).

43 Package: c.j.r.targets.xml

43.1 Overview

This package contains classes that support JFreeReport output to XML format.

43.2 Usage

To export a report in XML format, you can use code similar to the following (taken from the `StraightToXML.java` demo application, included in the JFreeReport distribution):

```
public boolean saveXML(JFreeReport report, String fileName)
{
  Writer out = null;
  try
  {
    out = new BufferedWriter(new FileWriter(new File(fileName)));

    XMLProcessor xprc = new XMLProcessor(report);
    xprc.setWriter(out);
    xprc.processReport();
    return true;
  }
  catch (Exception e)
  {
    System.err.println("Writing XML failed.");
    System.err.println(e.toString());
    return false;
  }
  finally
  {
    try
    {
      if (out != null)
      {
        out.close();
      }
    }
    catch (Exception e)
    {
      System.err.println("Saving XML failed.");
      System.err.println(e.toString());
    }
  }
}
```

43.3 XMLProcessor

43.3.1 Overview

A *report processor* that generates output in XML format.

43.3.2 Constructors

To create a new processor:

`public XMLProcessor(JFreeReport report);`
Creates a new report processor. The `report` is cloned by the constructor, and all processing is performed using the clone.

43.3.3 Methods

To set the character stream writer for the report processor:

> `public void setWriter(Writer writer);`
> Sets the character stream writer for the report processor.

To process the report:

> `public void processReport () throws ReportProcessingException;`
> Processes the report.

43.3.4 Notes

The `StraightToXML.java` demonstration application provides an example of how to use this class.

43.4 XMLWriter

43.4.1 Overview

This class generates the content for the XML export. Extends `AbstractFunction`.

43.4.2 Notes

This class is used by the `XMLProcessor`—it creates an instance, attaches it to the report, then begins processing.

44 Package: c.j.r.util

44.1 Overview

This package contains utility classes used in the JFreeReport class library. Classes and interfaces are included for:

- downward compatibility with JDK 1.2.2;
- a simple logging framework;
- generating standard `Paper` and `PageFormat` instances;
- useful data structures;
- user interface effects;

44.2 AbstractActionDowngrade

44.2.1 Overview

This class extends the `javax.swing.AbstractAction` class by implementing the `ActionDowngrade` interface.

44.2.2 Notes

This class allows JFreeReport to compile using JDK 1.2.2. It is used throughout the `com.jrefinery.report.action` package.

44.3 ActionButton

44.3.1 Overview

A button (extends `javax.swing.JButton`) that works with actions.

44.3.2 Notes

This duplicates functionality introduced in JDK 1.3, allowing JFreeReport to compile using JDK 1.2.2.

See Also

`ActionMenuItem`.

44.4 ActionDowngrade

44.4.1 Overview

An interface that extends `javax.swing.action` by defining two constants that were added to the `Action` interface in JDK 1.3. This interface is implemented by the `AbstractActionDowngrade` class.

44.4.2 Notes

This interface is used to allow JFreeReport to compile using JDK 1.2.2.

See Also

`AbstractActionDowngrade`.

44.5 ActionMenuItem

44.5.1 Overview

A menu item (extends `javax.swing.JMenuItem`) that works with actions.

44.5.2 Notes

This duplicates functionality introduced in JDK 1.3, allowing JFreeReport to compile using JDK 1.2.2.

This class is used by the `PreviewFrame` class.

See Also

`ActionButton`.

44.6 ActionRadioButton

44.6.1 Overview

A radio button (extension of `JRadioButton`) that works with actions.

44.6.2 Notes

This class is provided for JDK 1.2.2 compatibility.

44.7 CharacterEntityParser

44.7.1 Overview

A utility class for replacing encoding and decoding standard HTML and XML character entities.

44.8 CloseableTableModel

44.8.1 Overview

An interface that extends the `TableModel` interface by adding a `close()` method.

44.8.2 Methods

This interface adds just one method to the `TableModel` interface:

```
public void close();
```
Closes the table model.

44.8.3 Notes

Classes that implement this interface include:

- `ResultSetTableModelFactory`
- `ScrollableResultSetTableModel`.

44.9 DefaultReportConfiguration

44.9.1 Overview

A simple extension of the `ReportConfiguration` class that defines some default configuration settings. This class is, in fact, not used anywhere.

44.10 ExceptionDialog

44.10.1 Overview

A dialog (extends `javax.swing.JDialog`) that is used to display exception messages and stack traces.

44.10.2 Notes

This dialog is used by the `PreviewFrame` class.

44.11 FileSystemFilter

44.11.1 Overview

A generic file name filter implementation (extends `FileFilter` and implements `FilenameFilter`).

44.12 FloatingButtonEnabler

44.12.1 Overview

This class can be used to provide a "mouse-over" effect on buttons—a button border is only displayed when the mouse pointer is over the button.

44.13 HashNMap

44.13.1 Overview

A container that allows you to store more than one object per key value..

44.13.2 Notes

This class is not used in JFreeReport at present.

44.14 HTMLCharacterEntities

44.14.1 Overview

A collection of HTML character entities, stored as a properties collection (this class extends `Properties`).

44.14.2 Notes

This class is used by the `HTMLCharacterEntityParser` class.

44.15 ImageComparator

44.15.1 Overview

A class that can test two images for equality.

44.16 IOUtils

44.16.1 Overview

A collection of I/O related utility methods.

44.17 KeyedQueue

44.17.1 Overview

A hash-table like container that has a maximum number of elements. If the maximum number of elements is exceeded, an element is removed on a first-in, first-out basis.

44.17.2 Notes

This class is used by the `ImageLoadFilter` class.

44.18 LengthLimitingDocument

44.18.1 Overview

An extension of the `PlainDocument` class that limits the total length of the document.

44.18.2 Notes

This class is used in the `CSVExportDialog` to ensure that the user can enter only one character for the separator value.

44.19 LevelList

44.19.1 Overview

A list that associates a level (instance of `Integer`) with each item in the list.

44.20 Log

44.20.1 Overview

A static class that implements (with the help of a couple of other classes) a simple logging framework used during the development of JFreeReport. This framework allows messages to be logged on zero, one or more logging targets.

Four logging levels are defined:

Level:	**Description:**
ERROR	Used to log an error message.
WARN	Used to log a warning message.
INFO	Used to log an info message.
DEBUG	Used to log a debug message.

44.20.2 Methods

To add a logging target:

```
public static void addTarget(LogTarget target);
```
Adds a new logging target.

To log a message:

```
public static void log(int level, String message);
```
Logs a message. All log targets will receive the message.

44.20.3 Notes

You can switch off logging using the global report configuration (see section 8.2).

44.21 LogTarget

44.21.1 Overview

An interface that defines the methods that must be supported by a *logging target*. A logging target can be registered with the Log class, and will receive logging messages.

44.21.2 Notes

The `SystemOutLogTarget` is the only implementation of this interface to date.

44.22 NoCloseOutputStream

44.22.1 Overview

An extension of `FilterOutputStream` that does not close its parent stream when the `close` method is called.

44.23 NullOutputStream

44.23.1 Overview

A null output stream, used to ensure that no actual output is produced for reports during the "prepare run".

44.24 ObjectStreamResolveException

44.24.1 Overview

An exception that is thrown when an object from the XML input stream cannot be resolved.

44.25 PageFormatFactory

44.25.1 Overview

A class that can create `PageFormat` and `Paper` instances.

44.25.2 Standard Paper Sizes

This class includes definitions for the standard paper sizes defined by Adobe:

```
http://partners.adobe.com/asn/developer/pdfs/tn/5003.PPD_Spec_v4.3.pdf
```

The dimensions for the standard paper sizes are:

```
public static final int[] PAPER10X11 = { 720, 792 };
public static final int[] PAPER10X13 = { 720, 936 };
public static final int[] PAPER10X14 = { 720, 1008 };
public static final int[] PAPER12X11 = { 864, 792 };
public static final int[] PAPER15X11 = { 1080, 792 };
public static final int[] PAPER7X9 =   { 504, 648 };
public static final int[] PAPER8X10 = { 576, 720 };
public static final int[] PAPER9X11 = { 648, 792 };
public static final int[] PAPER9X12 = { 648, 864 };
public static final int[] A0 = { 2384, 3370 };
public static final int[] A1 = { 1684, 2384 };
public static final int[] A2 = { 1191, 1684 };
public static final int[] A3 = { 842, 1191 };
public static final int[] A3_TRANSVERSE = { 842, 1191 };
public static final int[] A3_EXTRA = { 913, 1262 };
public static final int[] A3_EXTRATRANSVERSE = { 913, 1262 };
public static final int[] A3_ROTATED = { 1191, 842 };
public static final int[] A4 = { 595, 842 };
public static final int[] A4_TRANSVERSE = { 595, 842 };
public static final int[] A4_EXTRA = { 667, 914 };
public static final int[] A4_PLUS = { 595, 936 };
public static final int[] A4_ROTATED = { 842, 595 };
public static final int[] A4_SMALL = { 595, 842 };
public static final int[] A5 = { 420, 595 };
public static final int[] A5_TRANSVERSE = { 420, 595 };
public static final int[] A5_EXTRA = { 492, 668 };
public static final int[] A5_ROTATED = { 595, 420 };
public static final int[] A6 = { 297, 420 };
public static final int[] A6_ROTATED = { 420, 297 };
public static final int[] A7 = { 210, 297 };
public static final int[] A8 = { 148, 210 };
public static final int[] A9 = { 105, 148 };
public static final int[] A10 = { 73, 105 };
public static final int[] ANSIC = { 1224, 1584 };
public static final int[] ANSID = { 1584, 2448 };
public static final int[] ANSIE = { 2448, 3168 };
public static final int[] ARCHA = { 648, 864 };
public static final int[] ARCHB = { 864, 1296 };
public static final int[] ARCHC = { 1296, 1728 };
public static final int[] ARCHD = { 1728, 2592 };
public static final int[] ARCHE = { 2592, 3456 };
public static final int[] B0 = { 2920, 4127 };
public static final int[] B1 = { 2064, 2920 };
public static final int[] B2 = { 1460, 2064 };
public static final int[] B3 = { 1032, 1460 };
public static final int[] B4 = { 729, 1032 };
public static final int[] B4_ROTATED = { 1032, 729 };
public static final int[] B5 = { 516, 729 };
public static final int[] B5_TRANSVERSE = { 516, 729 };
public static final int[] B5_ROTATED = { 729, 516 };
public static final int[] B6 = { 363, 516 };
public static final int[] B6_ROTATED = { 516, 363 };
public static final int[] B7 = { 258, 363 };
public static final int[] B8 = { 181, 258 };
public static final int[] B9 = { 127, 181 };
public static final int[] B10 = { 91, 127 };
public static final int[] C4 = { 649, 918 };
public static final int[] C5 = { 459, 649 };
public static final int[] C6 = { 323, 459 };
public static final int[] COMM10 = { 297, 684 };
public static final int[] DL = { 312, 624 };
public static final int[] DOUBLEPOSTCARD = { 567, 419 };
// should be 419.5, but I ignore that..
public static final int[] DOUBLEPOSTCARD_ROTATED = { 419, 567 };
public static final int[] ENV9 = { 279, 639 };
public static final int[] ENV10 = { 297, 684 };
public static final int[] ENV11 = { 324, 747 };
public static final int[] ENV12 = { 342, 792 };
public static final int[] ENV14 = { 360, 828 };
public static final int[] ENVC0 = { 2599, 3676 };
public static final int[] ENVC1 = { 1837, 2599 };
public static final int[] ENVC2 = { 1298, 1837 };
public static final int[] ENVC3 = { 918, 1296 };
public static final int[] ENVC4 = { 649, 918 };
public static final int[] ENVC5 = { 459, 649 };
public static final int[] ENVC6 = { 323, 459 };
public static final int[] ENVC65 = { 324, 648 };
public static final int[] ENVC7 = { 230, 323 };
public static final int[] ENVCHOU3 = { 340, 666 };
public static final int[] ENVCHOU3_ROTATED = { 666, 340 };
public static final int[] ENVCHOU4 = { 255, 581 };
public static final int[] ENVCHOU4_ROTATED = { 581, 255 };
public static final int[] ENVDL = { 312, 624 };
public static final int[] ENVINVITE = { 624, 624 };
public static final int[] ENVISOB4 = { 708, 1001 };
public static final int[] ENVISOB5 = { 499, 709 };
public static final int[] ENVISOB6 = { 499, 354 };
public static final int[] ENVITALIAN = { 312, 652 };
public static final int[] ENVKAKU2 = { 680, 941 };
public static final int[] ENVKAKU2_ROTATED = { 941, 680 };
public static final int[] ENVKAKU3 = { 612, 785 };
public static final int[] ENVKAKU3_ROTATED = { 785, 612 };
public static final int[] ENVMONARCH = { 279, 540 };
public static final int[] ENVPERSONAL = { 261, 468 };
public static final int[] ENVPRC1 = { 289, 468 };
public static final int[] ENVPRC1_ROTATED = { 468, 289 };
public static final int[] ENVPRC2 = { 289, 499 };
public static final int[] ENVPRC2_ROTATED = { 499, 289 };
public static final int[] ENVPRC3 = { 354, 499 };
public static final int[] ENVPRC3_ROTATED = { 499, 354 };
public static final int[] ENVPRC4 = { 312, 590 };
public static final int[] ENVPRC4_ROTATED = { 590, 312 };
public static final int[] ENVPRC5 = { 312, 624 };
public static final int[] ENVPRC5_ROTATED = { 624, 312 };
public static final int[] ENVPRC6 = { 340, 652 };
public static final int[] ENVPRC6_ROTATED = { 652, 340 };
public static final int[] ENVPRC7 = { 454, 652 };
public static final int[] ENVPRC7_ROTATED = { 652, 454 };
public static final int[] ENVPRC8 = { 340, 876 };
public static final int[] ENVPRC8_ROTATED = { 876, 340 };
public static final int[] ENVPRC9 = { 649, 918 };
public static final int[] ENVPRC9_ROTATED = { 918, 649 };
public static final int[] ENVPRC10 = { 918, 1298 };
public static final int[] ENVPRC10_ROTATED = { 1298, 918 };
public static final int[] ENVYOU4 = { 298, 666 };
public static final int[] ENVYOU4_ROTATED = { 666, 298 };
public static final int[] EXECUTIVE = { 522, 756 };
public static final int[] FANFOLDUS = { 1071, 792 };
public static final int[] FANFOLDGERMAN = { 612, 864 };
public static final int[] FANFOLDGERMANLEGAL = { 612, 936 };
public static final int[] FOLIO = { 595, 935 };
public static final int[] ISOB0 = { 2835, 4008 };
public static final int[] ISOB1 = { 2004, 2835 };
public static final int[] ISOB2 = { 1417, 2004 };
public static final int[] ISOB3 = { 1001, 1417 };
public static final int[] ISOB4 = { 709, 1001 };
public static final int[] ISOB5 = { 499, 709 };
public static final int[] ISOB5_EXTRA = { 570, 782 };
public static final int[] ISOB6 = { 354, 499 };
public static final int[] ISOB7 = { 249, 354 };
public static final int[] ISOB8 = { 176, 249 };
public static final int[] ISOB9 = { 125, 176 };
public static final int[] ISOB10 = { 88, 125 };
public static final int[] LEDGER = { 1224, 792 };
public static final int[] LEGAL = { 612, 1008 };
public static final int[] LEGAL_EXTRA = { 684, 1080 };
public static final int[] LETTER = { 612, 792 };
public static final int[] LETTER_TRANSVERSE = { 612, 792 };
public static final int[] LETTER_EXTRA = { 684, 864 };
public static final int[] LETTER_EXTRATRANSVERSE = { 684, 864 };
public static final int[] LETTER_PLUS = { 612, 914 };
public static final int[] LETTER_ROTATED = { 792, 612 };
public static final int[] LETTER_SMALL = { 612, 792 };
public static final int[] MONARCH = ENVMONARCH;
public static final int[] NOTE = { 612, 792 };
public static final int[] POSTCARD = { 284, 419 };
public static final int[] POSTCARD_ROTATED = { 419, 284 };
public static final int[] PRC16K = { 414, 610 };
public static final int[] PRC16K_ROTATED = { 610, 414 };
public static final int[] PRC32K = { 275, 428 };
public static final int[] PRC32K_ROTATED = { 428, 275 };
public static final int[] PRC32K_BIG = { 275, 428 };
public static final int[] PRC32K_BIGROTATED = { 428, 275 };
public static final int[] QUARTO = { 610, 780 };
public static final int[] STATEMENT = { 396, 612 };
public static final int[] SUPERA = { 643, 1009 };
public static final int[] SUPERB = { 864, 1380 };
public static final int[] TABLOID = { 792, 1224 };
public static final int[] TABLOIDEXTRA = { 864, 1296 };
```

44.25.3 Methods

To create a `Paper` instance using one of the predefined standard paper sizes:

> `public Paper createPaper(String name);`
> Creates a `Paper` object. For the `name`, you can use any of the constants listed in the previous section—introspection is used to obtain the required dimensions. The *imageable area* for the paper is, by default, set to the whole of the page (no border).

To create a `PageFormat` instance:

> `public PageFormat createPageFormat(Paper paper, int orientation);`
> Creates a `PageFormat` object, with the specified paper size and orientation.

44.25.4 Notes

This class is used by the `ReportFactory` class. Under normal circumstances, you won't need to use this class directly.

44.26 PropertiesIterator

44.26.1 Overview

This iterator provides access to a subset of the properties in a `java.util.Properties` collection. You can specify an optional prefix, and the iterator will search for property names of the form `<prefix>0`, `<prefix>1`, `<prefix>2`, ... , `<prefix>n`.

44.26.2 Notes

This class is used in the implementation of the `TextFormatExpression` class.

44.27 PropertyFileReportConfiguration

44.27.1 Overview

An extension of the `ReportConfiguration` class that adds a method for reading properties from a property file.

44.28 ReportConfiguration

44.28.1 Overview

A class used to represent both the *global report configuration* (a single default configuration that is "inherited" by all reports) and the *local report configuration* assigned to each individual report.

44.28.2 Usage - Global Configuration

To get access to the *global report configuration*:

```
ReportConfiguration global = ReportConfiguration.getGlobalConfig();
```

44.28.3 Usage - Report Configuration

You can obtain the configuration for a report using:

```
ReportConfiguration config = myReport.getReportConfiguration();
```

Using this reference, you can modify the report configuration. For example, to disable the "export to Excel" facility for the report:

```
config.setExportToExcelEnabled(false);
```

44.28.4 Methods

To set the log level:

> `public void setLogLevel(String level);`
> Sets the log level (which is read by the `Log` class when it is first loaded). Valid values are:
>
> - `Error` – log error messages only;
> - `Warn` – log warning and error messages;
> - `Info` – log info, warning and error messages;
> - `Debug` – log all messages.

To enable/disable export to CSV format:

> `public void setEnableExportCSV(boolean enabled);`
> Sets a flag that determines whether or not the "export to CSV" option is made available in the print preview facility for this report.

To enable/disable export to Excel format:

> `public void setEnableExportExcel(boolean enabled);`
> Sets a flag that determines whether or not the "export to Excel" option is made available in the print preview facility for this report.

See Also

`JFreeReport`.

44.29 ReportProperties

44.29.1 Overview

A container for storing *report properties*, used by the `JFreeReport` class.

This class provides a mechanism for "marking" particular properties. Access to report properties, via the `DataRow`, is provided for marked properties only.

44.29.2 Methods

To add a property:

> `public void put(String key, Object value);`
> Adds a property to the collection.

To retrieve a property:

> `public Object get(String key);`
> Returns a property from the collection (or `null` if there is no such key.

To "mark" or "unmark" a property:

> `public void setMarked(String property, boolean marked);`
> Marks or unmarks a property in the collection. There is no requirement that the property marked is actually defined in the collection.

44.29.3 Notes

Internally, properties are stored in a `HashTable`. This wrapper class ensures that all property keys are `String` objects.

This class is `Cloneable`, to support the cloneability of the `JFreeReport` class.

44.30 ReportPropertiesList

44.30.1 Overview

A wrapper for the `ReportProperties` container that provides access to the "marked" properties only.

44.31 StackableException

44.31.1 Overview

An extension of the `Exception` class that allows exceptions to be "stacked". This is a useful debugging aid.

44.32 StackableRuntimeException

44.32.1 Overview

An extension of the `RuntimeException` class that allows exceptions to be "stacked". This is a useful debugging aid.

44.33 StringUtil

44.33.1 Overview

A collection of utility functions for working with `String` objects.

44.34 SystemOutLogTarget

44.34.1 Overview

A *logging target* that sends all logging messages to `System.out`.

44.34.2 Methods

This class provides implementations for all the methods in the `LogTarget` interface.

44.35 SystemPropertyConfiguration

44.35.1 Overview

A *report configuration* class that provides *read only* access to all the standard Java system properties, whether they are related to JFreeReport or not.

This class is a subclass of `ReportConfiguration`.

44.35.2 Methods

The method for setting properties is overridden:

> `public void setConfigProperty(String key, String value);`
> Throws an `OperationNotSupportedException` because this is a *read only* collection.

44.35.3 Notes

An instance of this class is installed, by default, as the parent of the *global report configuration* object. This means that you can set default configuration settings in the system properties, if you want to.

44.36 WaitingImageObserver

44.36.1 Overview

An *image observer* that waits for an image to finish loading before continuing. This class implements the `ImageObserver` interface.

44.36.2 Usage

The following example demonstrates the use of this class by loading an image from a URL and adding it to a report as a property:

```
// add an image as a report property...
URL imageURL = getClass().getResource("/com/jrefinery/report/demo/gorilla.jpg");
Image image = Toolkit.getDefaultToolkit().createImage(imageURL);
WaitingImageObserver obs = new WaitingImageObserver(image);
obs.waitImageLoaded();
this.report.setProperty("logo", image);
this.report.setPropertyMarked("logo", true);
```

44.37 WeakReferenceList

44.37.1 Overview

An abstract base class for implementing a list using weak references.

44.37.2 Notes

The `ReportStateList` uses a subclass of this class to store report states.

44.38 Worker

44.38.1 Overview

A *worker* thread carries out a task and then sleeps until either new work is assigned or the worker is killed.

44.38.2 Notes

This class is used by the report preview components for repaginating reports.

A The Report Definition DTDs

A.1 Introduction

The JFreeReport report definition DTDs are reproduced here for convenience. For the latest versions, please refer to:

`http://jfreereport.sourceforge.net/`

The original author of both DTDs is Thomas Morgner.

A.2 The Simple Report Format DTD

The simple report format:

```
<!--
  Report DTD for JFreeReport version 0.8.1.

ChangeLog:

2002-05-01: Taq: Added multiline-field and multiline-function element
  2002-05-23: Changed the font style for extended attributes. The old attribute is also valid
              and gets silently mapped to the new style by the parser.
  2002-06-09: Added the rectangle shape element
  2002-06-30: ImageField, ImageFunction
  2002-07-10: ImageURLField, ImageURLFunction
  2002-07-17: Updated missing attributes in Image* tags
  2002-09-16: Removed BASELINE attribute from font definition, fixed Orientation-attribute declaration,
              Bands can have a default font declaration
  2002-10-16: All textelements have now a "dynamic" attribute which defaults to false
  2002-12-01: Removed deprecated General and multiline-field and *-function tags
              GroupHeader has "repeat" attribute
  2002-12-02: Band-heights are no longer required. If they are set, they act as minimum-height.
  2002-12-08: Added support for ReportConfiguration over XML
  2003-01-25: ResourceLabel and ResourceField added
              additional font style attributes added
              all elements except the shape elements now able to be dynamic
 -->

<!ENTITY % boolean
   "true | false"
>

<!ENTITY % fontstyle
   "plain | bold | italic | bold-italic"
>

<!ENTITY % alignmentEnum
   "left | center | right"
>

<!ENTITY % valignmentEnum
   "top | middle | bottom"
>

<!ENTITY % orientations
  "portrait | landscape | reverse_landscape"
>

<!--

 The position of an element is declared either relative to the last
 defined element in the band (or (0,0) if the element is the first element),
 or it is positioned absolute to the top left corner of the current band.

 When positioning an element be aware to take care of the elements width.
 The next element should be placed at the absolute position x+width,
 or the elements will overwrite each other.

 All Fontstyles default to plain and all boolean styles to false. If no font is
 set either in Band nor in element, a compiled in default font is used.

-->
<!ENTITY % position
 "x           CDATA          #IMPLIED
  y           CDATA          #IMPLIED
  width       CDATA          #REQUIRED"
>
```

```
<!--

  Colors are specified in HTML Syntax, so use #FFFFFF for white and #000000
  for black when using RGB numeric values. You may also use defined named
  constants for the color, as "black", "white" and so on.

  The constants understood by the parser are:
  "black", "blue", "cyan", darkGray", "gray", "green", "lightGray", "magenta",
  "orange", "pink", "red", "white", "yellow"

  -->
<!ENTITY % basicform
" color          CDATA          #IMPLIED
name           CDATA          #IMPLIED"
 >

<!ENTITY % fontdef
" fontname       CDATA              #IMPLIED
  fontstyle      (%fontstyle;)      #IMPLIED
  fontsize       CDATA              #IMPLIED
  fsbold         (%boolean;)        #IMPLIED
  fsitalic       (%boolean;)        #IMPLIED
  fsunderline    (%boolean;)        #IMPLIED
  fsstrikethr    (%boolean;)        #IMPLIED
  font-embedded  (%boolean;)        #IMPLIED
  font-encoding  CDATA              #IMPLIED
  line-height    CDATA              #IMPLIED

  alignment      (%alignmentEnum;)  #IMPLIED
  vertical-alignment     (%valignmentEnum;) #IMPLIED"
 >

<!--
  Removed multiline function: This element was never implemented and now all text elements
  support linebreaks.
  -->
<!ENTITY % itemelements
  "(label | string-field | number-field | date-field |
imageref | image-field | imageurl-field | rectangle |
  resource-label | resource-field | line)*">


<!--

     Colordefinitions are given either as predefined names as defined
     in HTML or as HTML-RGB-Color-Reference ("#rrggbb");

 -->


<!--

   A report constists of several elements, which are all optional:

   * reportheader
     printed at the first page

   * reportfooter
     printed on the last page

   * a page header
     Printed before any content is printed to the page.

   * a page footer
     printed, after the last content for the page is printed
     The pagefooter is always positionated at the bottom of a page,
     regardless how much space of the page is filled.

   * one or more group definitions in the "groups" element
     If no groups are defined, a default group is created to contain
     all data elements of the current report.

   * the (optional) item band. This is where the data rows are printed.
     If no item band is defined, only printing is disabled. All
     calculations are performed regardless of the appearance of the
     items.

   Attributes:
    Width  = the width of the report in Java-Printing-Units
    Height = the height of the report

    All printing units are defined in 1/72 inches, the default printing
    resolution on java.awt.graphics.

    minPageFormat = a predefined page format, for instance "A4" or "US-LETTER"


  -->

<!ENTITY % pageFormats
  "(  PAPER10X11 | PAPER10X13 | PAPER10X14 | PAPER12X11 | PAPER15X11 | PAPER7X9 | PAPER8X10 |
      PAPER9X11 | PAPER9X12 | A0 | A1 | A2 | A3 | A3_TRANSVERSE | A3_EXTRA | A3_EXTRATRANSVERSE |
      A3_ROTATED | A4 | A4_TRANSVERSE | A4_EXTRA | A4_PLUS | A4_ROTATED | A4_SMALL | A5 |
      A5_TRANSVERSE | A5_EXTRA | A5_ROTATED | A6 | A6_ROTATED | A7 | A8 | A9 | A10 |
      ANSIC | ANSID | ANSIE | ARCHA | ARCHB | ARCHC | ARCHD | ARCHE | B0 | B1 | B2 | B3 | B4 |
      B4_ROTATED | B5 | B5_TRANSVERSE | B5_ROTATED | B6 | B6_ROTATED | B7 | B8 | B9 | B10 |
      C4 | C5 | C6 | COMM10 | DL | DOUBLEPOSTCARD | DOUBLEPOSTCARD_ROTATED | ENV9 | ENV10 |
      ENV11 | ENV12 | ENV14 | ENVC0 | ENVC1 | ENVC2 | ENVC3 | ENVC4 | ENVC5 | ENVC6 | ENVC65 | ENVC7 |
      ENVCHOU3 | ENVCHOU3_ROTATED | ENVCHOU4 | ENVCHOU4_ROTATED | ENVDL | ENVINVITE | ENVISOB4 | ENVISOB5 |
      ENVISOB6 | ENVITALIAN | ENVKAKU2 | ENVKAKU2_ROTATED | ENVKAKU3 | ENVKAKU3_ROTATED | ENVMONARCH |
      ENVPERSONAL | ENVPRC1 | ENVPRC1_ROTATED | ENVPRC2 | ENVPRC2_ROTATED | ENVPRC3 | ENVPRC3_ROTATED |
      ENVPRC4 | ENVPRC4_ROTATED | ENVPRC5 | ENVPRC5_ROTATED | ENVPRC6 | ENVPRC6_ROTATED | ENVPRC7 |
      ENVPRC7_ROTATED | ENVPRC8 | ENVPRC8_ROTATED | ENVPRC9 | ENVPRC9_ROTATED | ENVPRC10 | ENVPRC10_ROTATED |
      ENVYOU4 | ENVYOU4_ROTATED | EXECUTIVE | FANFOLDUS | FANFOLDGERMAN | FANFOLDGERMANLEGAL |
      FOLIO | ISOB0 | ISOB1 | ISOB2 | ISOB3 | ISOB4 | ISOB5 | ISOB5_EXTRA | ISOB6 | ISOB7 | ISOB8 | ISOB9 |
      ISOB10 | LEDGER | LEGAL | LEGAL_EXTRA | LETTER | LETTER_TRANSVERSE | LETTER_EXTRA | LETTER_EXTRATRANSVERSE |
      LETTER_PLUS | LETTER_ROTATED | LETTER_SMALL | MONARCH | NOTE | POSTCARD | POSTCARD_ROTATED | PRC16K |
      PRC16K_ROTATED | PRC32K | PRC32K_ROTATED | PRC32K_BIG | PRC32K_BIGROTATED | QUARTO | STATEMENT | SUPERA |
      SUPERB | TABLOID | TABLOIDEXTRA )"
>

<!ELEMENT report   (configuration?, reportheader?, reportfooter?, pageheader?, pagefooter?, groups?, items?, functions?)>

<!ATTLIST report
  width          CDATA           #IMPLIED
  height         CDATA           #IMPLIED
  name           CDATA           #IMPLIED
  pageformat     %pageFormats;   #IMPLIED
  orientation    (%orientations;) "portrait"
  leftmargin     CDATA           #IMPLIED
  rightmargin    CDATA           #IMPLIED
  topmargin      CDATA           #IMPLIED
  bottommargin   CDATA           #IMPLIED
>

<!--

  Configure this instance of the report. You may use all defined ReportConfigurationKeys
  as PropertyNames. See jfreereport.properties for more details.

  This can be used to define the PDF-FontEncoding for an Report.

  -->
<!ELEMENT configuration (property*)>

<!--

  The reportheader can contain any band-element.
  The height of the report header is ignored, if the header and footer is
  printed on an own page.

  As with every element container you may define default font settings
  for sub elements without an own font definition.

  In an header the ownpage is translated as pagebreak_after_print,
  on an footer that attribute is translated to pagebreak_before_print.

  -->
<!ELEMENT reportheader ( %itemelements; )>
<!ATTLIST reportheader
  ownpage         (%boolean;)    #IMPLIED
  height         CDATA           #IMPLIED
   %basicform;
  %fontdef;
>

<!ELEMENT reportfooter ( %itemelements; )>
<!ATTLIST reportfooter
  ownpage        (%boolean;)    #IMPLIED
  height         CDATA          #IMPLIED
   %basicform;
  %fontdef;
>



<!--

  The pageheader can contain any band-element.

  As with every element container you may define default font settings
  for sub elements without an own font definition.

  -->
<!ELEMENT pageheader ( %itemelements; )>
<!ATTLIST pageheader
  onfirstpage    (%boolean;)   #IMPLIED
  height         CDATA         #IMPLIED
  %basicform;
  %fontdef;
>
```

```
<!ELEMENT pagefooter ( %itemelements; )>
<!ATTLIST pagefooter
  onfirstpage    (%boolean;)   #IMPLIED
  onlastpage     (%boolean;)   #IMPLIED
  height         CDATA         #IMPLIED
  %basicform;
  %fontdef;
>

<!--

  The tag encapsulates all groups. This tag helps to keep parsing
  simple. If no groups are defined, a default group is created and
  contains all elements of the report datarow

  -->
<!ELEMENT groups (group*)>

<!--
  The fields are required
  -->
<!ELEMENT group  (groupheader?, groupfooter?, fields)>
<!ATTLIST group
  name           CDATA         #IMPLIED
>

<!--

  A group header is printed before a group starts. A group start
  is invoked when one element in groupelements changes and on the
  start of the report generation.

  If pagebreak is set to true, a page break will be forced before
  the group header is printed.

  If repeat is set to true, this header is repeated after an pagebreak
  if this group is still active and no other groupheader in an subgroup
  has the repeat flag set.
  -->
<!ELEMENT groupheader ( %itemelements; )>
<!ATTLIST groupheader
  pagebreak     (%boolean;)   #IMPLIED
  repeat        (%boolean;)   #IMPLIED
  height         CDATA        #IMPLIED
  %basicform;
  %fontdef;
>

<!--
  This pagebreak is a pagebreak before print ...
  -->
<!ELEMENT groupfooter ( %itemelements; )>
<!ATTLIST groupfooter
  pagebreak     (%boolean;)   #IMPLIED
  height         CDATA        #IMPLIED
  %basicform;
  %fontdef;
>

<!--
  the name of the elements that have to change for
  a group break.

  A report group may have more than one group element.
  A group element may only contain Strings defining the
  names of the items which form a group. This is not
  limited to items from the data model, you may also
  enter functions here.

 -->
<!ELEMENT fields  (field*)>
<!ELEMENT field  (#PCDATA)>

<!ELEMENT items   ( %itemelements; )>
<!ATTLIST items
  height         CDATA          #IMPLIED
  %basicform;
  %fontdef;
>

<!--

  A simple label, static text that does not change, the text
  contains a ResourceBundle key, which is looked up during the
  report processing.

  -->
<!ELEMENT resource-label   (#PCDATA)>
<!ATTLIST resource-label
  height         CDATA          #REQUIRED
  %basicform;
  %position;
  %fontdef;
  nullstring     CDATA          #IMPLIED
  dynamic        (%boolean;)    #IMPLIED
  resource-base  CDATA          #IMPLIED
>

<!--

  A text field. The field data contains a ResourceBundle key,
  which is looked up during the report processing.

  -->
<!ELEMENT resource-field   EMPTY>
<!ATTLIST resource-field
  height         CDATA          #REQUIRED
  %basicform;
  %position;
  %fontdef;
  fieldname   CDATA    #REQUIRED
  nullstring  CDATA    #IMPLIED
  dynamic     (%boolean;)   #IMPLIED
  resource-base  CDATA          #IMPLIED
>

<!--
  A simple label, static text that does not change
  -->
<!ELEMENT label   (#PCDATA)>
<!ATTLIST label
  height         CDATA          #REQUIRED
  %basicform;
  %position;
  %fontdef;
  nullstring     CDATA          #IMPLIED
  dynamic        (%boolean;)    #IMPLIED
>

<!--
  A simple text field
  -->
<!ELEMENT string-field   EMPTY>
<!ATTLIST string-field
  height         CDATA          #REQUIRED
  %basicform;
  %position;
  %fontdef;
  fieldname   CDATA    #REQUIRED
  nullstring  CDATA    #IMPLIED
  dynamic     (%boolean;)   #IMPLIED
>


<!ELEMENT number-field   EMPTY>
<!ATTLIST number-field
  height         CDATA          #REQUIRED
  %basicform;
  %position;
  %fontdef;
  format CDATA      #IMPLIED
  fieldname   CDATA    #REQUIRED
  nullstring  CDATA    #IMPLIED
  dynamic     (%boolean;)   #IMPLIED
>

<!ELEMENT date-field   EMPTY>
<!ATTLIST date-field
  height         CDATA          #REQUIRED
  %basicform;
  %position;
  %fontdef;
  format CDATA      #IMPLIED
  fieldname   CDATA    #REQUIRED
  nullstring  CDATA    #IMPLIED
  dynamic     (%boolean;)   #IMPLIED
>


<!--

   The image reference links an external image into the report.

  -->
<!ELEMENT imageref   EMPTY>
<!ATTLIST imageref
  %position;
  src         CDATA #REQUIRED
  height      CDATA #REQUIRED
  name        CDATA #IMPLIED
  dynamic     (%boolean;)   #IMPLIED
  scale       (%boolean;)   #IMPLIED
  keepAspectRatio     (%boolean;)   #IMPLIED
  dynamic        (%boolean;)    #IMPLIED
>
<!--

   The image reference links an external image into the report. This element expects an
   Graphics2D-Object in the datasource.

  -->
<!ELEMENT image-field   EMPTY>
<!ATTLIST image-field
  %position;
  fieldname   CDATA #REQUIRED
  height      CDATA #REQUIRED
  name        CDATA #IMPLIED
  dynamic     (%boolean;)   #IMPLIED
  scale       (%boolean;)   #IMPLIED
  keepAspectRatio     (%boolean;)   #IMPLIED
  dynamic        (%boolean;)    #IMPLIED
>
<!--

   The image reference links an external image into the report. This element expects an
   URL or URL-String in the datasource.

  -->
<!ELEMENT imageurl-field   EMPTY>
<!ATTLIST imageurl-field
  %position;
  height      CDATA #REQUIRED
  fieldname   CDATA #REQUIRED
  name        CDATA #IMPLIED
  dynamic     (%boolean;)   #IMPLIED
  scale       (%boolean;)   #IMPLIED
  keepAspectRatio     (%boolean;)   #IMPLIED
  dynamic        (%boolean;)    #IMPLIED
>


<!--

  Shapes

  -->
<!ELEMENT line      EMPTY>
<!ATTLIST line
  x1        CDATA       #REQUIRED
  y1        CDATA       #REQUIRED
  x2        CDATA       #REQUIRED
  y2        CDATA       #REQUIRED
color     CDATA       #IMPLIED
name      CDATA       #IMPLIED
weight    CDATA       #IMPLIED
>

<!--

  The rectangle is a filled rectangular area. No outline is drawn.

  -->
<!ELEMENT rectangle EMPTY>
<!ATTLIST rectangle
  %position;
  %basicform;
  height         CDATA   #REQUIRED
  draw     (%boolean;)   #IMPLIED
  fill     (%boolean;)   #IMPLIED
  weight    CDATA        #IMPLIED
>

<!--

  Functions are defined in a function library.
  Every referenced function has to be defined in the
  library in order to be loaded and executed correctly.

  -->
<!ELEMENT functions  (function | expression | data-ref | property-ref)*>

<!--

  A data reference is used to validate the table model against
the declared format of the datasource. It simply checks that
all fields declared are present and are assignable from the
given type.

The attribute class is used to validate the data models objects.
This function uses TableModel.getColumnClass() to query the
table models data types. If the tablemodel returns java.lang.Object
no check is done, java.lang.Object is considered as an indicator
that the table model does not know about it's internal data
```

```
structure.

The data-reference was never implemented and will not be implemented
  in future releases.

-->
<!ELEMENT data-ref  EMPTY>
<!ATTLIST data-ref
  name    CDATA   #REQUIRED
class   CDATA   #IMPLIED
>

<!--

  A reference to a report property. This property is predefined here and can be accessed
  as any datasource. The value defaults to null if no more data is given. The encoding parameter
  defaults to "text", "serialized-base64" is implemented later to allow serialized objects
  as value for the property.

-->
<!ELEMENT property-ref  (#PCDATA)>
<!ATTLIST property-ref
  name       CDATA   #REQUIRED
encoding   CDATA   #IMPLIED
>

<!--

  A defined function has a valid implementing class that implement
  the com.jrefinery.report.function.Function interface. Functions have access to the datarow and
  can access other functions or expressions or the datasource. Functions are statefull and maintain
  their state during the report generation. For stateless userdefined computations consider using
  an expression instead of functions, as expression are cheaper to compute and maintain when using
  huge reports.

  Function parameters are given by propery elements. For visual
  editing, function must obey to the java-beans rules (use get*/set*
  methods, perhaps provide beaninfo and so on)

  The deplevel attribute can be used to priorize the functions. Functions with an higher depencylevel
  are executed before any function with lower depency levels. Depencylevels lower than 0 are not allowed.
  -->
<!ELEMENT function   (properties?)>
<!ATTLIST function
 class      CDATA    #REQUIRED
name       CDATA    #REQUIRED
 deplevel   CDATA    #IMPLIED
>


<!--

  An expression is a stateless userdefined function. It can access the datarow and the reportproperties
  to perform its task. Using the datarow an expression has access to the datasource and other functions
  and expressions.

  Expression parameters are given by propery elements. For visual
  editing, Expressions must obey to the java-beans rules (use get*/set*
  methods, perhaps provide beaninfo and so on)

  The deplevel attribute can be used to priorize the functions. Functions with an higher depencylevel
  are executed before any function with lower depency levels. Depencylevels lower than 0 are not allowed.
  -->
<!ELEMENT expression   (properties?)>
<!ATTLIST expression
  class      CDATA    #REQUIRED
name       CDATA    #REQUIRED
 deplevel   CDATA    #IMPLIED
>

<!ELEMENT properties  (property*)>

<!ELEMENT property    (#PCDATA)>
<!ATTLIST property
  name       CDATA    #REQUIRED
  encoding   CDATA    #IMPLIED
>
```

A.3 The Extended Report Format DTD

The extended report format DTD:

```
<!--
  Report DTD for JFreeReport version 0.8.1.
  ChangeLog:
    11-Feb-2003 : Initial version

-->

<!ENTITY % orientations
  "portrait | landscape | reverse_landscape"
>

<!ENTITY % pageFormats
  "(  PAPER10X11 | PAPER10X13 | PAPER10X14 | PAPER12X11 | PAPER15X11 | PAPER7X9 | PAPER8X10 |
      PAPER9X11 | PAPER9X12 | A0 | A1 | A2 | A3 | A3_TRANSVERSE | A3_EXTRA | A3_EXTRATRANSVERSE |
      A3_ROTATED | A4 | A4_TRANSVERSE | A4_EXTRA | A4_PLUS | A4_ROTATED | A4_SMALL | A5 |
      A5_TRANSVERSE | A5_EXTRA | A5_ROTATED | A6 | A6_ROTATED | A7 | A8 | A9 | A10 |
      ANSIC | ANSID | ANSIE | ARCHA | ARCHB | ARCHC | ARCHD | ARCHE | B0 | B1 | B2 | B3 | B4 |
      B4_ROTATED | B5 | B5_TRANSVERSE | B5_ROTATED | B6 | B6_ROTATED | B7 | B8 | B9 | B10 |
      C4 | C5 | C6 | COMM10 | DL | DOUBLEPOSTCARD | DOUBLEPOSTCARD_ROTATED | ENV9 | ENV10 |
      ENV11 | ENV12 | ENV14 | ENVC0 | ENVC1 | ENVC2 | ENVC3 | ENVC4 | ENVC5 | ENVC6 | ENVC65 | ENVC7 |
      ENVCHOU3 | ENVCHOU3_ROTATED | ENVCHOU4 | ENVCHOU4_ROTATED | ENVDL | ENVINVITE | ENVISOB4 | ENVISOB5 |
      ENVISOB6 | ENVITALIAN | ENVKAKU2 | ENVKAKU2_ROTATED | ENVKAKU3 | ENVKAKU3_ROTATED | ENVMONARCH |
      ENVPERSONAL | ENVPRC1 | ENVPRC1_ROTATED | ENVPRC2 | ENVPRC2_ROTATED | ENVPRC3 | ENVPRC3_ROTATED |
      ENVPRC4 | ENVPRC4_ROTATED | ENVPRC5 | ENVPRC5_ROTATED | ENVPRC6 | ENVPRC6_ROTATED | ENVPRC7 |
      ENVPRC7_ROTATED | ENVPRC8 | ENVPRC8_ROTATED | ENVPRC9 | ENVPRC9_ROTATED | ENVPRC10 | ENVPRC10_ROTATED |
      ENVYOU4 | ENVYOU4_ROTATED | EXECUTIVE | FANFOLDUS | FANFOLDGERMAN | FANFOLDGERMANLEGAL |
      FOLIO | ISOB0 | ISOB1 | ISOB2 | ISOB3 | ISOB4 | ISOB5 | ISOB5_EXTRA | ISOB6 | ISOB7 | ISOB8 | ISOB9 |
      ISOB10 | LEDGER | LEGAL | LEGAL_EXTRA | LETTER | LETTER_TRANSVERSE | LETTER_EXTRA | LETTER_EXTRATRANSVERSE |
      LETTER_PLUS | LETTER_ROTATED | LETTER_SMALL | MONARCH | NOTE | POSTCARD | POSTCARD_ROTATED | PRC16K |
      PRC16K_ROTATED | PRC32K | PRC32K_ROTATED | PRC32K_BIG | PRC32K_BIGROTATED | QUARTO | STATEMENT | SUPERA |
      SUPERB | TABLOID | TABLOIDEXTRA )"
>

<!-- Report definition -->
<!ELEMENT report-definition (parser-config?, report-config?,
                             styles?, templates?, report-description?,
                             functions?, data-definition?)>
<!ATTLIST report-definition
  name CDATA   #IMPLIED
>

<!ELEMENT parser-config ((element-factory|stylekey-factory|template-factory|datadefinition-factory|object-factory|datasource-factory)*)>

<!ELEMENT stylekey-factory EMPTY>
<!ATTLIST stylekey-factory
  class      CDATA   #IMPLIED
>

<!ELEMENT object-factory EMPTY>
<!ATTLIST object-factory
  class      CDATA   #IMPLIED
>

<!ELEMENT datasource-factory EMPTY>
<!ATTLIST datasource-factory
  class      CDATA   #IMPLIED
>

<!ELEMENT template-factory EMPTY>
<!ATTLIST template-factory
  class      CDATA   #IMPLIED
>

<!ELEMENT datadefinition-factory EMPTY>
<!ATTLIST datadefinition-factory
  class      CDATA   #IMPLIED
>

<!ELEMENT element-factory EMPTY>
<!ATTLIST element-factory
  class      CDATA   #IMPLIED
>


<!ELEMENT report-config (defaultpageformat?, configuration?)>

<!ELEMENT defaultpageformat EMPTY>
<!ATTLIST defaultpageformat
  width          CDATA           #IMPLIED
  height         CDATA           #IMPLIED
  pageformat     %pageFormats;   #IMPLIED
  orientation    (%orientations;) "portrait"
  leftmargin     CDATA           #IMPLIED
  rightmargin    CDATA           #IMPLIED
  topmargin      CDATA           #IMPLIED
  bottommargin   CDATA           #IMPLIED
>

<!ELEMENT configuration (property*)>

<!ELEMENT property (#PCDATA)>
<!ATTLIST property
  name           CDATA           #REQUIRED
```

```
>

<!ELEMENT output-config (EMPTY)>

<!ELEMENT styles (style*)>
<!ATTLIST styles
  external       CDATA           #IMPLIED
>

<!ELEMENT style  (extends*, (basic-key|compound-key)*)>
<!ATTLIST style
  name           CDATA           #IMPLIED
>

<!ELEMENT default-style  (extends*, (basic-key|compound-key)*)>
<!ATTLIST default-style
  name           CDATA           #IMPLIED
>

<!ELEMENT extends  EMPTY>
<!ATTLIST extends
  name          CDATA  #REQUIRED
>

<!ELEMENT compound-key (basic-object|compound-object)*>
<!ATTLIST compound-key
  name           CDATA           #IMPLIED
  class          CDATA           #IMPLIED
>

<!ELEMENT basic-key    (#PCDATA)>
<!ATTLIST basic-key
  name           CDATA           #IMPLIED
  class          CDATA           #IMPLIED
>


<!ELEMENT compound-object (basic-object|compound-object)*>
<!ATTLIST compound-object
  name           CDATA           #REQUIRED
  class          CDATA           #IMPLIED
>

<!ELEMENT basic-object    (#PCDATA)>
<!ATTLIST basic-object
  name           CDATA           #REQUIRED
  class          CDATA           #IMPLIED
>

<!-- Not yet defined -->
<!ELEMENT templates    (template*)>
<!ATTLIST templates
  external       CDATA           #IMPLIED
>

<!ELEMENT template     (basic-object|compound-object)*>
<!ATTLIST template
  name           CDATA           #IMPLIED
  references     CDATA           #REQUIRED
>

<!ELEMENT report-description
  (report-header?, report-footer?, page-header?, page-footer?, itemband?, groups?)>

<!ELEMENT report-header   (style, default-style, (template | datasource), (band | element)*)>
<!ATTLIST report-header
  name           CDATA           #IMPLIED
>

<!ELEMENT report-footer   (style, default-style, (template | datasource), (band | element)*)>
<!ATTLIST report-footer
  name           CDATA           #IMPLIED
>

<!ELEMENT page-header   (style, default-style, (template | datasource), (band | element)*)>
<!ATTLIST page-header
  name           CDATA           #IMPLIED
>

<!ELEMENT page-footer   (style, default-style, (template | datasource), (band | element)*)>
<!ATTLIST page-footer
  name           CDATA           #IMPLIED
>

<!ELEMENT itemband   (style, default-style, (template | datasource), (band | element)*)>
<!ATTLIST itemband
  name           CDATA           #IMPLIED
>

<!ELEMENT groups     (group*)>
<!ELEMENT group      (fields, group-header?, group-footer?)>
```

```
<!ATTLIST group
  name           CDATA           #REQUIRED
>

<!ELEMENT fields     (field*)>
<!ELEMENT field      (#PCDATA)>

<!ELEMENT group-header   (style, default-style, (template | datasource), (band | element)*)>
<!ATTLIST group-header
  name           CDATA           #IMPLIED
>

<!ELEMENT group-footer   (style, default-style, (template | datasource), (band | element)*)>
<!ATTLIST group-footer
  name           CDATA           #IMPLIED
>

<!ELEMENT band  (style, default-style, (template | datasource)*, (band | element)*)>
<!ATTLIST band
  name           CDATA           #IMPLIED
>

<!ELEMENT element  (style, (template | datasource))>
<!ATTLIST element
  name           CDATA           #IMPLIED
  type           CDATA           #REQUIRED
>

<!ELEMENT datasource (datasource?,(basic-object|compound-object)*)>
<!ATTLIST datasource
  type          CDATA            #REQUIRED
>

<!--

  Functions are defined in a function library.
  Every referenced function has to be defined in the
  library in order to be loaded and executed correctly.

  -->
<!ELEMENT functions  (function | expression | data-ref | property-ref)*>
<!ELEMENT properties  (property*)>

<!--

  A reference to a report property. This property is predefined here and can be accessed
  like any other datasource. The value defaults to null if no more data is given. The
  encoding parameter defaults to "text", "serialized-base64" is implemented later to
  allow serialized objects as value for the property.

  The class parameter specifies the class of this report property value. The given class
  must have a valid ObjectDescription registered in the ClassFactory.

-->
<!ELEMENT property-ref  (#PCDATA)>
<!ATTLIST property-ref
  name       CDATA   #REQUIRED
encoding   CDATA   #IMPLIED
  class      CDATA   #IMPLIED
>

<!--

  A defined function has a valid implementing class that implement
  the com.jrefinery.report.function.Function interface. Functions have access to the datarow and
  can access other functions or expressions or the datasource. Functions are statefull and maintain
  their state during the report generation. For stateless userdefined computations consider using
  an expression instead of functions, as expression are cheaper to compute and maintain when using
  huge reports.

  Function parameters are given by propery elements. For visual
  editing, function must obey to the java-beans rules (use get*/set*
  methods, perhaps provide beaninfo and so on)

  The deplevel attribute can be used to priorize the functions. Functions with an higher depencylevel
  are executed before any function with lower depency levels. Depencylevels lower than 0 are not allowed.
  -->
<!ELEMENT function   (properties?)>
<!ATTLIST function
 class      CDATA    #REQUIRED
name       CDATA    #REQUIRED
 deplevel   CDATA    #IMPLIED
>


<!--

  An expression is a stateless userdefined function. It can access the datarow and the reportproperties
  to perform its task. Using the datarow an expression has access to the datasource and other functions
  and expressions.
  Expression parameters are given by propery elements. For visual
  editing, Expressions must obey to the java-beans rules (use get*/set*
  methods, perhaps provide beaninfo and so on)

  The deplevel attribute can be used to priorize the functions. Functions with an higher depencylevel
  are executed before any function with lower depency levels. Depencylevels lower than 0 are not allowed.
  -->
<!ELEMENT expression   (properties?)>
<!ATTLIST expression
  class      CDATA    #REQUIRED
name       CDATA    #REQUIRED
 deplevel   CDATA    #IMPLIED
>
```

B The GNU Lesser General Public Licence

B.1 Introduction

JFreeReport is licensed under the terms of the GNU Lesser General Public Licence (LGPL). The full text of this licence is reproduced in this appendix. You should read and understand this licence before using JFreeReport in your own projects.

If you are not familiar with the idea of *free software* and/or *open source software*, you can find out more at the following web-sites:

Organisation:	**Description:**
The Free Software Foundation	`http://www.fsf.org`
The Open Source Initiative	`http://www.opensource.org`

Please send e-mail to `david.gilbert@object-refinery.com` if you have any questions about the licensing of JFreeReport.

B.2 The Licence

The following licence has been used for the distribution of the JFreeReport class library:

object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to