The JasperReports Ultimate Guide

Version 1.0

Author:
Teodor Danciu

Table of contents

1 Introduction 4
2 API Overview 5
3 Main Tasks and Processes 10
3.1 XML Parsing 10
3.2 Compiling Report Designs 10
3.3 Report Design Preview 12
3.4 Filling Reports 13
3.5 Viewing Reports 15
3.6 Printing Reports 16
3.7 Exporting Reports 17
3.8 Object Loading and Saving 17
4 Report Designs 19
4.1 DTD Reference 19
4.2 XML Encoding 20
4.3 Report Properties 21
5 Report Data 25
5.1 Expressions 25
5.2 Parameters 26
5.2.1 Built-in Report Parameters 28
5.3 Data Source 29
5.4 Report Query 31
5.5 Fields 32
5.6 Variables 34
5.6.1 Calculations 35
5.6.2 Built-in Report Variables 37
6 Report Sections 38
6.1 Main Report Sections 39
6.2 Data Grouping 40
7 Scriptlets 43
8 Report Elements 44
8.1 Text Elements 48
8.1.1 Fonts and Unicode Support 49
8.1.2 Static Texts 53
8.1.3 Text Fields 53
8.2 Graphic Elements 56
8.2.1 Lines 57
8.2.2 Rectangles 58
8.2.3 Images 58

8.2.4 Charts and Graphics 60
8.3 Hyperlinks 60
8.4 Element Groups 62
9 Subreports 63
9.1 Subreport Parameters 65
9.2 Subreport Data Source 66
10 Advanced JasperReports 67
10.1 XML Report Designs Loading and Writing 67
10.2 Implementing Data Sources 67
10.3 Customizing Viewers 68
10.4 Exporting to New Output Formats 68

1 Introduction

The JasperReports library is a very powerful and flexible report-generating tool that has the ability to deliver rich content onto the screen, to the printer or into PDF, HTML or XML files. Hopefully, in the future, other output formats such as CSV, XLS, RTF and other will be supported.

The library is entirely written in Java and can be used in a variety of Java enabled applications, including J2EE or Web applications, to generate dynamic content. Its main purpose is to help creating page oriented, ready to print documents in a simple and flexible manner.

JasperReports organizes data according to the report design defined in an XML file. This data may come from various data sources including relational databases, collections or arrays of Java objects. Users can plug the reporting library to custom made data sources, by implementing a simple interface, as you will see later in this book.

In order to fill a report with data, the XML report design must be compiled first. Through compilation, a report design object is generated and then serialized in order to store it on disk or send it over the network. This serialized object is then used when the application wants to fill the specified report design with data. In fact, the compilation of a report design implies the compilation of all Java expressions defined in the XML file representing the report design. Various verifications are made at compilation time, to check the report design consistency. The result is a ready to fill report design that will be then used to generate documents on different sets of data.

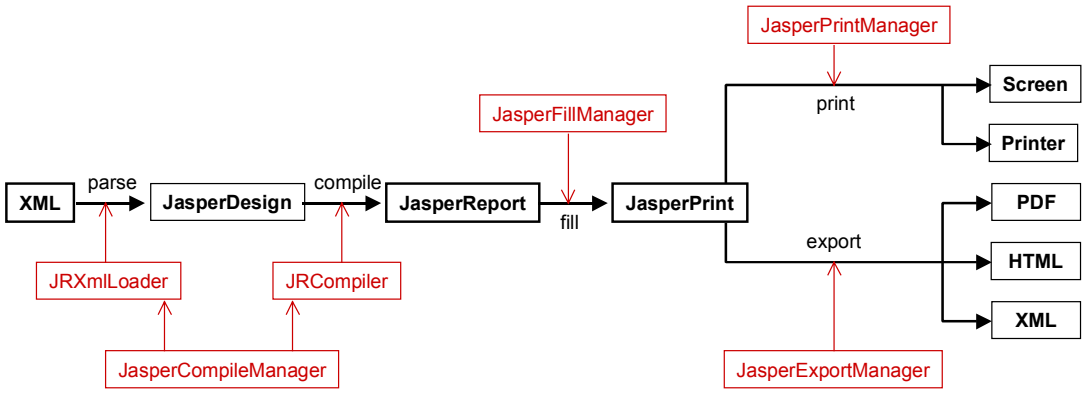
To fill a report design, the engine needs to receive the report data. This may come in various forms. Some of this data can be passed in as report parameters. But most of the data will be found in the report data source. The reporting engine can directly receive special data source objects from which to get the information to put on the report, or can deal itself with a supplied JDBC connection object, if that data is located in a relational database.

The result of the report filling operation is a new object that represents the ready to print document. This one is also serialized for storage on disk or network transfer. It can be viewed directly using the JasperReports built-in viewer or can be exported to other, more popular formats like PDF, HTML or XML.

2 API Overview

Most of the time, when using the JasperReports library, people will get to work only with a few classes and won't have to get to know the entire API, in order to benefit from all its features and functionality.

In this section we shall get a close look at the classes and interfaces which are significant when using the library and see how to make use of them in applications that need the reporting functionality that JasperReports offer.



- figure 1 -

Class `dori.jasper.engine.design.JasperDesign`

We begin with this class because instances of it represent the raw material, which the JasperReports library uses for report generating purposes. Such instances are obtained after the XML report design files are parsed by the library internal XML parsing procedures, for example, but can be build programmatically, by the application that uses JasperReports, if working with XML files is not an option. Among the supplied samples that come with the project source files, there is one called *noxmldesign* that you can check to see how to dynamically create a `dori.jasper.engine.design.JasperDesign` object without editing an XML report design file.

All instances of the `dori.jasper.engine.design.JasperDesign` class are subject to compilation before being used for filling and report generation. This is why they are considered the raw material for the library.

Class `dori.jasper.engine.JasperReport`

Instances of this class represent compiled report design objects. These can be obtained only as a result of the JasperReports report compilation process and are ready to use for filling with data and report generation.

Through compilation, along with various consistency checks and rearrangements of the report elements for more rapid later utilization, the library creates a temporary class file containing all the report expressions such as report variables expressions, text field and image expressions, group expressions, etc.
This temporary Java source file is compiled on the fly using either the Java compiler classes from the JDK used to run the application. If the `tools.jar` file is not found in the classpath in order to do that, the compilation will go ahead anyway by launching at runtime the `javac.exe` compiler, behind the scenes. The bytecodes of the resulting class file are stored in the resulting `dori.jasper.engine.JasperReport` for using when filling the report with data, to evaluate the various report expressions at runtime.

Class `dori.jasper.engine.JasperCompileManager`

This is the class that exposes all the library functionality concerning the report compilation.
It has various methods that allow the users to compile XML report designs found in files on disk or that come from input streams. It also lets you compile in-memory report designs by directly passing a `dori.jasper.engine.design.JasperDesign` object and receiving the corresponding `dori.jasper.engine.JasperReport` object.

Other utility methods include report design verification and XML report design generation for in-memory constructed `dori.jasper.engine.design.JasperDesign` class instances. Those are very useful especially in GUI tools that simplify the report design work.

Class `dori.jasper.engine.JasperPrint`

After a compiled report design is filled with data, the resulting document comes in the form of a `dori.jasper.engine.JasperPrint` instance. Such an object can be viewed directly using the JasperReports build-in report viewer, or can be serialized for disk storage and later use, or for sending it over the network.

The instances of this class represent the output of the report filling process of the JasperReports library and can be considered as a custom format for storing full featured, page oriented documents. They can be transformed into other more popular formats like PDF, HTML, XML or other by using the library's export functionality.

Interface `dori.jasper.engine.JRDataSource`

JasperReports is a very flexible reporting tool as far as the source of the report data is concerned. It lets people use any kind of data source they might want to, as long as they can provide an appropriate implementation of this interface, so that the reporting engine can interpret and retrieve the data from that data source when filling the reports.

Normally, every time a report is being filled, an instance of this interface is always supplied or created behind the scenes by the reporting engine.

Class `dori.jasper.engine.JRResultSetDataSource`

This is a default implementation of the `dori.jasper.engine.JRDataSource` interface.
Since most of the reports are generated using data that comes from a relational database, JasperReports includes by default this implementation that wraps a `java.sql.ResultSet` object.

This class can be instantiated on purpose, to wrap already loaded result sets, before passing it to the report filling routines, but it is also used by the reporting engine to wrap the data retrieved from the database after having executed through JDBC the report query, if present.

Class `dori.jasper.engine.data.JRTableModelDataSource`

This class represents another default implementation of the `dori.jasper.engine.JRDataSource` interface that is shipped with the library. It wraps a `javax.swing.table.TableModel` object and can be used in Java Swing applications to generate reports using data that has already been loaded into on-screen tables.

Class `dori.jasper.engine.JREmptyDataSource`

Being the most simple implementation of `the dori.jasper.engine.JRDataSource` interface, this class can be used in reports that do not display data from the supplied data source, but rather from parameters, and when only the number of virtual rows in the data source is important.

Many of the provided samples such as *fonts*, *images*, *shapes* and *unicode* use an instance of this class when filling the reports, to simulate a data source with one record in it, but with all the fields `null`.

Class `dori.jasper.engine.JasperFillManager`

This class is the façade to the report filling functionality of the JasperReports library.
It exposes a variety of methods that receive a report design in the form of an object, file or input stream and produce a document also in various output forms: object, file or output stream.

But along with the report design, the report filling engine has to receive also the data source from which to retrieve data and the values for the report parameters, in order to generate the documents.
Parameter values are always supplied in a `java.util.Map` object in which the keys are the report parameter names.

The data source can be supplied in two different forms, depending on the situation:

Normally, it has to be is supplied as a `dori.jasper.engine.JRDataSource` object, like already mentioned above.

But with the majority of reports in the reporting world being filled with data from relational databases, JasperReports has a built-in default behavior that lets people specify an SQL query in the report design itself. This SQL query is executed in order to retrieve the data to use when filling the report at runtime.
In such cases, the only thing JasperReports needs is a `java.sqlConnection` object, instead of the usual data source object. It needs this connection object to connect to the relational database management system through JDBC and execute the report query.
It will automatically creates a `dori.jasper.engine.JRResultSetDataSource` behind the scenes to wrap the `java.sql.ResultSet` object returned after the execution of the query and passes it to the normal report filling process.

Class `dori.jasper.engine.JRAbstractScriptlet`

Scriptlets are a very powerful feature of the JasperReports library. They allow users to write custom code that will be executed by the reporting engine during the report filling process. This user code can deal with report data manipulation and gets executed at well-defined moments such as page, column or group breaks, opening a whole new range of possibilities in customizing the content of the generated documents.

Class `dori.jasper.engine.JRDefaultScriptlet`

This is a convenience subclass of the `dori.jasper.engine.JRAbstractScriptlet` class. Most of the time users will chose to subclass this when working with scriptlets, so they won't have to implement all the abstract methods declared in the abstract class.

Class `dori.jasper.engine.JasperPrintManager`

We are talking here about a Java reporting tool and what reporting tools are meant for is printing.
After having filed a report, we have the option of viewing it, exporting it into a different format and last but not least printing it.

In JasperReports, we can print reports using this particular manager class, which is a façade to the printing functionality exposed by the library.

We can find here various methods that send to the printer entire documents or only portions of it, either by displaying the print dialog or not.

The content of a page from a JasperReports document can be displayed by generating a `java.awt.Image` object for it using this manager class.

Class `dori.jasper.engine.JasperExportManager`

As already mentioned, JasperReports allows transforming generated documents from its proprietary format into more popular documents formats such as PDF, HTML or XML. With time, this part of the JasperReports functionality will be extended to support other formats like CSV, XSL and other.

This manager class has various methods that ca process data that comes from different sources and goes to different destinations: files, input and output streams, etc.

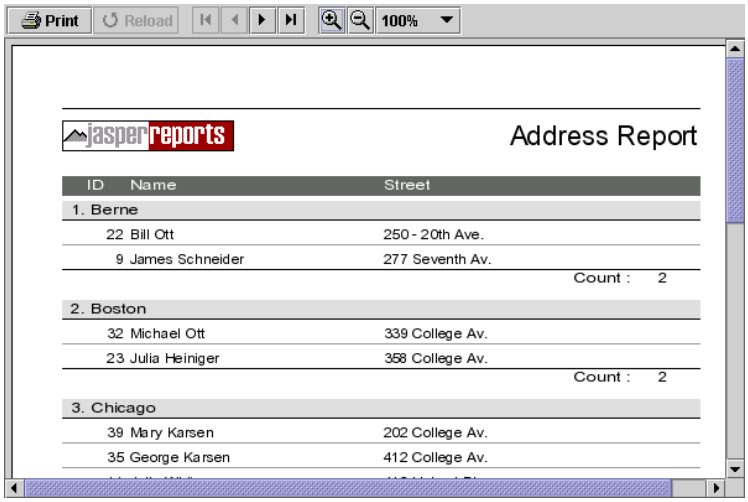
Class `dori.jasper.engine.JasperRunManager`

Sometimes is useful to produce documents only in a popular format such as PDF or HTML, without having to store on disk the serialized, intermediate `dori.jasper.egine.JasperPrint` object, produced by the report filling process.
This can be achieved using this manager class which immediately exports the document produced by the report filling process into the desired output format.
The use of this manager class is shown and can be tested in the supplied *webapp* sample, where PDF and HTML content is produced on the fly.

Class `dori.jasper.view.JRViewer`

This class is different from the rest of the classes listed above in the way that it is more like pluggable visual component than an utility class.

It can be used in Swing based applications to view the reports generated by the JasperReports library.



- figure 2 -

This visual component is not meant to satisfy everybody. It was included in the main library more like a demo component, to show how the core printing functionality can be used to display the reports in Swing based applications, by generating `java.awt.Image` objects for the document pages, using the `dori.jasper.engine.JasperPrintManager` class.

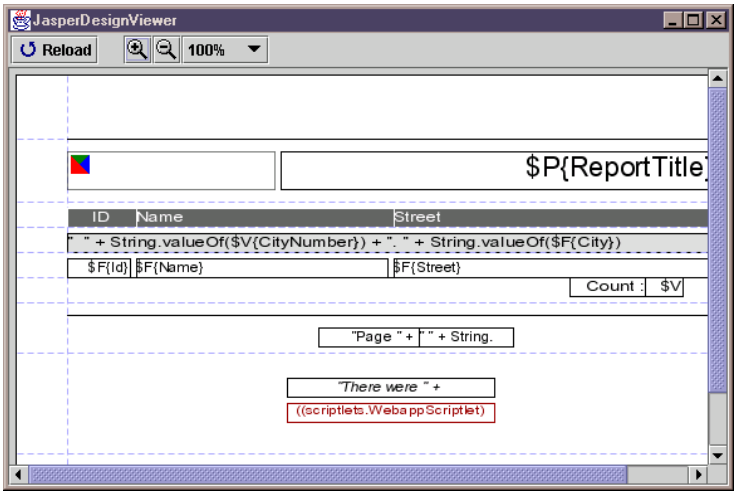
The preferred way to adapt this component to a particular application needs is by subclassing it.

Class `dori.jasper.view.JasperViewer`

This is also more like a didactical purpose class that uses the `dori.jasper.view.JRViewer` component to display reports. It represents a simple Java Swing application that can load and display reports. It is used in almost all of the supplied samples to display the generated documents.

Class `dori.jasper.view.JasperDesignViewer`

Usually, an application that uses the JasperReports library for reporting purposes will never get to work with this class. This can be used at design time to preview the report templates before going into production and help with the design work. It was included in the main library as a development tool in order to make up for the missing visual designer.



- figure 3 -

This is also used in all the samples to preview the report designs, either in the raw XML form or the compiled form.

Class `dori.jasper.engine.util.JRLoader`

All JasperReports main processes, like report compilation, report filling and exporting, often work with serialized objects. Sometimes it is useful to manually load those serialized objects before submitting them to the desired JasperReport process.

This is why we have the `dori.jasper.engine.util.JRLoader` class, which is a utility class that helps loading serialized objects found in various locations such as files, URLs or input streams.

The most interesting method exposed by this class is the `loadObjectFromLocation(String location)` method.
When calling this method to load an object from the supplied location, the program will first try to interpret the location as a valid URL. If this fails, then the program will consider that the supplied location is the name of a file on disk and will try to read from it. If no file is found at that location, it will try to locate a resource through classpath that would correspond to the location. Only after this third try fails, an exception will be thrown.

3 Main Tasks and Processes

In this chapter we shall see what you have to know in order to parse your XML report designs, compile them, fill them with data, view them, print them, or export them to other formats.

3.1 XML Parsing

JasperReports uses the SAX 2.0 API to parse the XML files. However, it is not tided to a particular SAX 2.0 implementation, like Xerces for examples, but instead you are able to decide at runtime what XML parser you are using.

To instantiate the parser class, JasperReports uses the `createXMLReader()` method of the `org.xml.sax.helpers.XMLReaderFactory` class.
In this case, it will be necessary at runtime to set the `org.xml.sax.driver` Java system property to the full class name of the SAX driver, as specified in the SAX 2.0 documentation.

You can achieve this in two ways. We shall explain both using the Xerces XML parser, just like we do it in the provided samples. If you use a different SAX 2.0 XML parser, you have to modify the name of the parser class accordingly.

The first way you can set a system property is by using the `-D` switch in the command line when you launch the Java Virtual Machine:

```
java -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser MySAXApp
sample.xml
```

In all the provided samples we use the ANT build tool to perform different tasks. We supply this system property to the JVM using the `<sysproperty>` element of the `<java>` built-in task:

```
<sysproperty key="org.xml.sax.driver"
value="org.apache.xerces.parsers.SAXParser"/>
```

The second way to set a system property is by using the `java.lang.System.setProperty(String key, String value)` method like this:

```
System.setProperty("org.xml.sax.driver",
"org.apache.xerces.parsers.SAXParser");
```

Check the `jsp/compile.jsp` and `WEB-INF/classes/servlets/CompileServlet.java` files in the w*ebapp* sample provided, to see this in action.

3.2 Compiling Report Designs

In order to generate a report, one has to create the report's design first, either by editing an XML file or by programmatically building a `dori.jasper.engine.design.JasperDesign` object. In this book, we shall mainly deal with the XML approach, because it is the preferred way to use the JasperReports library at least for the moment, and we'll have the chance to better understand its behavior.

It is very likely that the present and future GUI tools that are and will be developed to help and simplify the report design work will directly use the JasperReports API to create the report design objects, without the need to pass through the XML form.

But this book does not try to document the use of such GUI tools and concentrates only on the JasperReports core functionality. This is why the entire book is oriented towards explaining the content

and syntax of the XML report designs. Once this approach understood, the other, programmatic approach should be more then intuitive.

As already mentioned, the XML report designs are the raw material that the library uses to generate reports. This is because this XML content has to be parsed and loaded into a `dori.jasper.engine.design.JasperDesign` object that has to suffer the report compilation process before being ready to be filed with data by the reporting engine.

Note that most of the time, the report compilation should be considered more like a development time job. You should compile your application report designs just like you compile your Java source files, and you should ship them already compiled, with your application, to be installed on the deployment platform. That's because in the majority of cases, the report designs are static and few applications need to offer their users the possibility to dynamically generate report designs, that would need to be compiled at runtime.

The main purpose of the report compilation process is to produce and load the bytecodes of a class containing all the report expressions. This dynamically created class will be used when filling the report to evaluate all those report expressions.

But before proceeding with this class generation, the engine verifies the report design for consistency and will not continue if at least one validation check fails. We shall see what are the conditions for a report design to be considered valid in the following chapters of this book.

For now, we only need to know how the report compilation works, so that we can make sure it can be performed successfully.

There are at least three important aspects concerning the way the bytecodes for this class containing all the report expressions are obtained:

- temporary working directory;
- Java compiler used;
- classpath;

In order to be able to compile a Java source file, this file must be created and saved on disk. The output of the Java compilation process is also a file with the `.class` extension. This is why JasperReports needs access to a temporary, working directory for it to create the class containing the report expressions and to compile it. After the report compilation task is finished, those temporary class files will be automatically deleted and the resulting bytecodes are stored in the resulting `dori.jasper.engine.JasperReport`, which can be serialized itself and stored on disk, if desired.

By default, the temporary working directory is the current directory when launching the JVM, and it is obtained interrogating the `user.dir` system property. It can easily be changed, by supplying a value to a system property called `jasper.reports.compile.temp`. This is useful especially in Web environment when you don't want to end up using the same directory that contains the batch files that launch the Web server, as a temporary working location for the report compilation process.

The second aspect mentioned concerns the Java compiler used to compile the report expressions class. First, the reporting engine tries to compile the Java source file using the `sun.tools.javac.Main` class. This approach may succeed only if the `tools.jar` file that contains this class, usually found in the `/lib` directory of the JDK installation directory, is available through classpath.
If loading the compiler class `sun.tools.javac.Main` fails, the program tries to dynamically launch the Java compilation process, just like we would normally do it from the command line, using the javac.exe program found in the `/bin` directory of the JDK installation directory.

This is why in the project tree available for download, copying the `tools.jar` file from the JDK location into the `/lib` directory of the JasperReports project is an optional operation. If the `tools.jar` file is not found in classpath, JasperReports displays a warning and continues as mentioned.

When compiling Java source files, the most important thing seems to be the classpath. If the Java compiler does not find in the supplied classpath all the classes that are referenced by the source files it tries to compile, the whole process fails and stops, the errors being displayed on the console.

The same happens also when JasperReports compiles the report expressions class. It is important then to make sure we supply to the Java compiler at runtime the correct classpath for the compilation process to succeed. For instance, we have to make sure that in the classpath we supply the custom classes that we might use in the report expressions.
There is a default behavior for this aspect also. If no special classpath for compiling the report classes is supplied, the engine will use the current JVM classpath returned by the system property `java.class.path`. This default behavior can be overridden by putting the desired classpath in the system property called `jasper.reports.compile.class.path`.

You can see the `jsp/compile.jsp` and `WEB-INF/classes/servlets/CompileServlet.java` files in the *webapp* sample provided, for the code snippet that makes use of this Java system properties to override the default behavior of the JasperReports report compilation process.

Most of the time, compiling a report requires only a simple call to the JasperReports library like in the following line of code:

```
dori.jasper.engine.JasperCompileManager.compileReport(myXmlFileName);
```

This call will produce the compiled report design and will store it in a file with the `.jasper` extension, located in the same directory as the supplied XML report design file.

3.3 Report Design Preview

The JasperReports library is not shipped with an advanced GUI tool to help the design work. At this time, there are at least 4 projects that try to provide such a tool.

However, the library contains a very helpful visual component that allows report creators to preview the report designs as they build them.

The `dori.jasper.view.JasperDesigner` class is a simple Swing based Java application that can load and display a report design either in its XML form or in the compiled form. Even if it is not a complex GUI application, lacking advance functionality like the drag and dropping of the visual report elements, it is very helpful instrument. All the supplied samples were created using this design viewer.

All the supplied samples have already prepared ANT tasks in their `build.xml` files that will launch this design viewer for you to see the report designs.
In fact there are 2 ANT tasks for each sample report: `viewDesign` and `viewDesignXML`.
The first one loads the compiled report design that is normally found in the `.jasper` file. The second one loads the XML report design, being the most useful since you can edit the XML file and push the *Reload* button to immediately see the modification appearing on the screen.

If you have the ANT build tool installed on your system, in order to preview a sample report design, you simply go to the desired sample directory and launch from the command line something like this:

```
>ant viewDesignXML
```

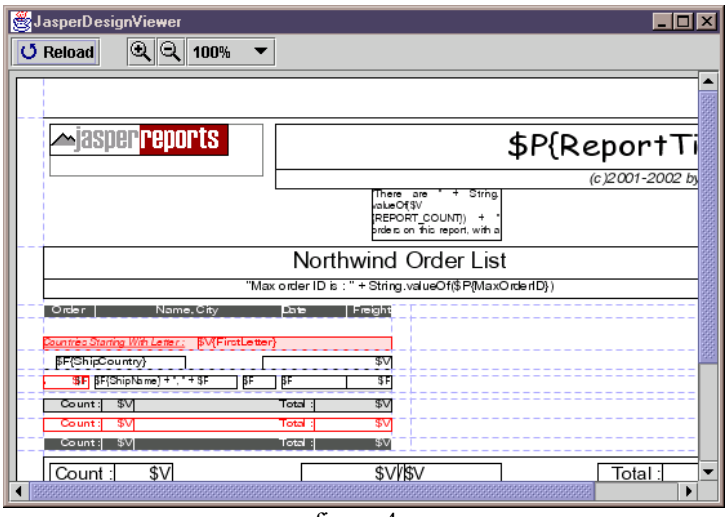
or

```
>ant viewDesign
```

In case you do not have ANT installed and you don't want to get it (you'll miss all the fun), here's the complete command line that will launch the design viewer to preview one of the sample XML report designs. We assume that you have a copy of the entire JasperReports source tree on your system (Windows command line syntax):

```
>java -classpath ./;../../../lib/commons-digester.jar;
../../../lib/commons-beanutils.jar;../../../lib/commons-collections.jar;
../../../lib/xerces.jar;../../../lib/jasperreports.jar
-Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser
dori.jasper.view.JasperDesignViewer -XML -FFirstJasper.xml
```

By launching this command line, you should be able to see this window:



- figure 4 -

3.4 Filling Reports

The report filling process is the most important of all the JasperReports library functionality. It represents the main objective for this piece of software, as it is the process that manipulates sets of data in order to produce high quality documents, just like any reporting tool is supposed to do.

There are tree things that should be supplied to the report filling process as input:

- report design (report template);
- parameters;
- data source.

The output is always a single, final document that is ready to be viewed, printed or exported to other formats.

We have already seen that in order to fill a report, we are going to use the `dori.jasper.engine.JasperFillManager` class. This class has various methods that allow us to fill report designs that are located on disk, come from input streams or are supplied directly as in-memory `dori.jasper.engine.JasperReport` objects.
The output produced always corresponds to the type of the input received. That is, when receiving a file name for the report design, the generated report will be also placed in a file on disk. When the report design is read from an input stream, the generated report will be written to an output stream and so forth.

It might be that those various utility methods for filling the reports are not sufficient for a particular application, who for example would want to load report designs as resources from classpath and output the generated documents to files on disk, at a certain location.

In such cases, the developers should consider loading the report design objects before passing them to the report filling routines, using the `dori.jasper.engine.util.JRLoader` utility class. This way, they could retrieve some report design properties such as the report name, so they can construct the name of the resulting document, to place at the desired disk location.

There are a lot of report filling scenarios that could be imagined in a real world application, and the report filling manager class only tries to cover a portion of them, considered to be used more often. But it should be no problem for anybody who might want to customize the report-filling process using the library's basic functionality, as suggested above.

The report parameter values are supplied always packed in a `java.util.Map` object, which has the parameter names as its keys.

As for the third thing that the report filling process expects to receive, the data source, there are two different scenarios.

Normally, the engine works with an instance of the `dori.jasper.engine.JRDataSource` interface, from which it extracts the data when filling the report.
And the façade class `dori.jasper.engine.JasperFillManager` has a full set of methods that receive a `dori.jasper.engine.JRDataSource` object as the data source of the report that is going to be filled.

But there is another set of report filling methods in this manager class that receive a `java.sql.Connection` object as a parameters, instead of expected data source object.
This is because most of the time reports are generated using data that comes from tables found in relational databases.
Users have the possibility to specify the SQL query needed to retrieve the report data from the database, in the report design itself. At runtime, the only thing the engine would need would be a JDBC connection object to use in order to connect to the desired relational database, execute the SQL query and retrieve the report data.
Behind the scenes, the engine will still use a special `dori.jasper.engine.JRDataSource` object, but this is preformed transparently for the calling program.

There are at least 4 sample applications provided with the project, which fill the reports using data from an also supplied HSQL database server. Those are *jasper*, *query*, *scriptlet* and *subreport* samples. To run them, you have to start the HSQL database server first, by going to the `/demo/hsqldb` directory of the JasperReports project and launching this command:

```
>ant
```

or

```
>ant runServer
```

Note that you have to have ANT build tool installed on your system for you to launch the server in such as simple manner. Otherwise you could still launch it like this:

```
>java -classpath ./;../../lib/hsqldb.jar org.hsqldb.Server
```

Here's a simple code snippet taken from the *query* sample that shows how to fill a report:

```
//Preparing parameters
Map parameters = new HashMap();
parameters.put("ReportTitle", "Address Report");
parameters.put("FilterClause", "'Boston', 'Chicago', 'Oslo'");
parameters.put("OrderClause", "City");

//Invoking the filling process
JasperFillManager.fillReportToFile(fileName, parameters, getConnection());
```

3.5 Viewing Reports

The output of the report filling process is always a `dori.jasper.engine.JasperPrint` object.
If we serialize this object and store it on disk, normally in a `.jrprint` file, we could say that this is the proprietary format in which JasperReports stores its generated documents.

In order to view the generated reports in this proprietary format or in the proprietary XML format produced by the internal XML exporter, JasperReports has a built-in viewer. It is a Swing based component that can be easily integrated in other Java applications that want to offer this functionality, without the need for them to export the documents in more popular formats, so that they can be viewed. The `dori.jasper.view.JRViewer` class represents this visual component. It can be customized to respond to a particular application needs by subclassing it. This way we could add or remove buttons from the existing toolbar it displays, or perform other modifications.

This is shown in the supplied *webapp* sample, where the `JRViewerPlus` class adds a new button to the existing toolbar of this report viewer component.

JasperReports comes also with an included simple Swing application that uses the visual component for viewing the reports. It helps viewing reports stored on disk, in the JasperReports `.jrprint` proprietary format as we called it or in the XML format produced by the default XML exporter.

This simple Java Swing application is implemented in the `dori.jasper.view.JasperViewer` class and it is used in almost all the provided samples to view the generated reports.

If you have the ANT build tool installed on your system, in order to view a sample report, you go to the desired sample directory and launch the following from the command line:

```
>ant view
```

or

```
>ant viewXML
```

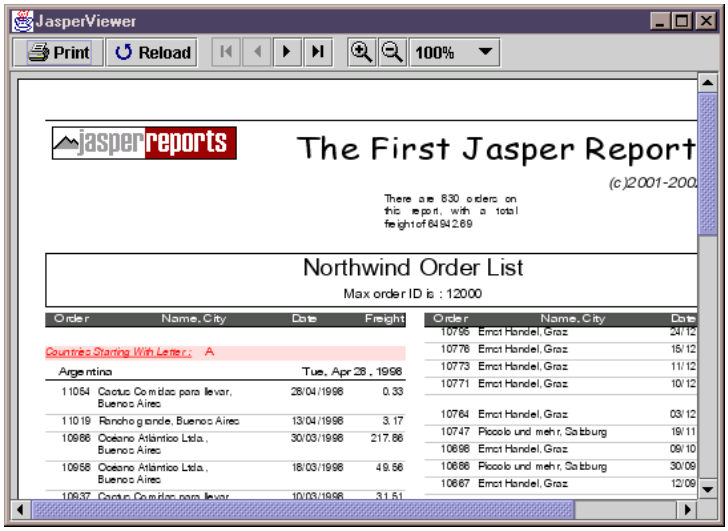
Here are the corresponding complete command lines in case you don't have ANT installed, to view one of the sample report both in the `.jrprint` and XML formats (Windows command line syntax):

```
>java -classpath ./;../../../lib/jasperreports.jar
dori.jasper.view.JasperViewer -FFirstJasper.jrprint
```

or

```
>java -classpath ./;../../../lib/commons-digester.jar;
../../../lib/commons-beanutils.jar;
../../../lib/commons-collections.jar;
../../../lib/xerces.jar;../../../lib/jasperreports.jar
-Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser
dori.jasper.view.JasperViewer -XML -FFirstJasper.jrpxml
```

By launching these command lines, you should be able to see this window:



- figure 5 -

The viewer application implemented in the `dori.jasper.view.JasperViewer` class should be considered more like a demo application that shows how the `dori.jasper.view.JRViewer` component can be used in Swing applications to display reports.
Those who will try to use it directly in their applications, by calling the public and static `viewReport()` methods it exposes, will end up noticing that when they close the report viewer frame, their application will unexpectedly terminate. This is because the `JasperViewer` class makes a call to the `System.exit(0)` and a way around this would be to subclass it and remove the `java.awt.event.WindowListener` it has registered by default.

3.6 Printing Reports

The main objective of the JasperReports library, if not of all existing reporting tools, is to create ready to print documents. The majority of reports that are generated by applications end up or are supposed to end up on paper.

We can print the documents generated by the JasperReports library using the `dori.jasper.engine.JasperPrintManager` class. Of course, documents can also be printed after they are exported to other formats such as HTML or PDF. But we are going to explain how to use the specialized manager class mentioned, to print documents that are stored or transferred in the JasperReports proprietary format (`dori.jasper.engine.JasperPrint` objects).

Among the various methods that the `dori.jasper.engine.JasperPrintManager` class exposes, we can find some that allow printing a whole document, a single page or a page range, with and without displaying the print dialog.

Here's how you could print an entire document without displaying the standard print dialog:

```
dori.jasper.engine.JasperPrintManager.printReport(myReport, false);
```

And now another code snippet that shows how to print all the pages from 5 to 11 of your document, after having displayed the standard print dialog:

```
dori.jasper.engine.JasperPrintManager.printPages(myReport, 4, 10, true);
```

3.7 Exporting Reports

In some application environments, it is useful to transform the JasperReports generated documents, from the proprietary format into other, more popular formats like PDF or HTML. Doing this, you can make sure other people can view those reports without having to install special viewers on their systems, especially when sending the documents over the network.

There is a façade class in JasperReports also for this type of functionality. Its name is `dori.jasper.engine.JasperExportmanager` and can be used to obtain PDF, HTML or XML content for the documents produced by the report filling process.

Exporting means taking a `dori.jasper.engine.JasperPrint` object, which represents a JasperReport document and transform it in a different format. The main reason to export the reports into other formats is to allow more and more people to view those reports. HTML reports can be viewed by anybody these days, since at least one browser is available on any system. Viewing JasperReports documents in their proprietary form would require the installation of special software on the target platform, at least in the form of a Java applet if not more.

In conclusion, the ability to export reports into other formats is a very useful feature and with time, more and more output formats will be supported.

Those who want to export their reports into other, new formats, have to implement a special interface called `dori.jasper.engine.JRExporter` or to extend the corresponding `dori.jasper.engine.JRAbstractExporter` class.

For the moment, the library is shipped with 3 special exporter classes that produce PDF, HTML and XML. Those are found in the `dori.jasper.engine.export` package, but can be used by calling the appropriate methods on the façade manager class mentioned previously.

Here's how you could export your report to HTML format:

```
dori.jasper.engine.JasperExportManager.exportReportToHtmlFile(myReport);
```

3.8 Object Loading and Saving

When using the JasperReports library functionality, you'll often get to work with serialized objects such as compiled report designs, generated reports, etc.
Sometimes you'll have to manually load serialized objects from different sources like files and input streams or to serialize the object you produce yourself using the library's core functionality.

There are two utility classes, specially created for these kinds of operations, which are often used by the reporting engine itself:

```
dori.jasper.engine.util.JRLoader
dori.jasper.engine.util.JRSaver.
```

The first one exposes various methods that allow people to load serialized object from different types of sources like files, URLs, input streams or classpath resources.
The most interesting utility method found in this class, is the one called `loadObjectFromLocation`.

It receives a `java.lang.String` as a parameter and returns the serialized object loaded from that particular location. First, the program tries to see whether the supplied location is a valid URL. If it is, it will try to load a serialized object from that URL. If it is not a valid URL, the program will assume that the parameter represents the name of a file and will try to locate on disk in order to read the serialized object from there. If it is no file is found, the program will finally assume that the specified location represents a classpath resource name and will read from there. Only if those operations fail, an exception will be thrown.

The counter part of this object loading utility class is the `dori.jasper.engine.util.JRSaver` class that you could use to serialize your objects and put them into files on disk or send them over the network through output streams.

There is another important aspect that we mention in this section because it has to do with object loading. It is about loading already generated reports or final JasperReports documents that have been previously exported to the XML format.
Unlike the above loader, it is not about loading serialized objects, but about parsing XML content and creating a `dori.jasper.engine.JasperPrint` object that will mirror the document found in that XML content.

This can be achieved using the `dori.jasper.engine.xml.JRPrintXmlLoader` class, which has public static methods that can create in-memory document objects by parsing XML content read from files or input streams.

4 Report Designs

The report design represents a template that will be used by the JasperReports engine to deliver dynamic content to the printer, to the screen or to the Web. Data stored in the database is organized during the report filling process according to this report design to obtain ready to print, page oriented documents.

Generally speaking, a report design contains all the information concerning the structure and the aspect of the documents that will be generated when the data will be provided. This information concerns the position and the content of various text or graphic elements that will appear on the document, their appearance, the custom calculations, data grouping and data manipulation that should be performed when generating the documents, etc.

Normally, the report designs are defined in XML files with a special structure that we shall see in detail later and are subject to the JasperReports compilation process before being filled with data. But they also can be constructed in-memory, programmatically, using the JasperReports API. There is a sample called *noxmldesign* shipped with the JasperReports project source files that shows how to directly create in-memory report designs, without editing any XML files at all.

4.1 DTD Reference

When working with XML report designs, JasperReports uses its own internal DTD files to validate the XML content it receives for processing. If the XML validation is passed, it means that the supplied report design corresponds to the JasperReports required XML structure and syntax and the engine is able to generate the compiled version of the report design.

Valid XML report designs always point to the JasperReports internal DTD files for validation. Without the DTD reference specified, the report compilation process fails abruptly. This should not be considered a too much burden for anybody since the DTD reference is always the same and can simply be copied from previous report designs. At the beginning you will copy it from the supplied samples.

As already mentioned, the engine recognizes only the DTD references that point to its internal DTD files. You cannot make a copy of the DTD files found among the library source files and point to that copy in your XML report designs. If you want to do that, you will also have to alter the code of some of the library classes including the `dori.jasper.engine.xml.JRXmlDigester` class.
If you ever encounter problems such as the engine not finding its own internal DTD files due to some resource loading problems, make sure you have eliminated every possible cause before deciding to use external DTD files. Encountering such a problem is very unlikely since the resource loading mechanism of the library was improved with time.

There are only two valid DTD references for the XML report designs and they are the following:

```
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN"
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
```

or

```
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN"
"http://www.jasperreports.com/dtds/jasperreport.dtd">
```

The root element of an XML report design is `<jasperReport>` and this is how an usual JasperReports XML report design file looks:

```
<?xml version="1.0"?>
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN"
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">

<jasperReport name="name_of_the_report" ... >
...
</jasperReports>
```

The first 3 points make it for the report design properties and settings and the other 3 for the suppressed various report design elements such as report parameters, fields, variables, groups, report sections, etc. We shall see all of them in detail in the following chapters of this book.

4.2 XML Encoding

When creating XML report designs in different languages, a special attention should be accorded to the encoding attribute that can be used in the header of the XML file. By default, if no value is specified for this attribute, the XML parser uses "UTF-8" as the encoding for the content of the XML file.

This important because the report design often contains localized static texts, which are introduced when manually editing the XML file.

For most of the West European languages, the "ISO-8859-1" encoding, also known as LATIN1, should be sufficient to deal with special characters like é, â, è, ç, that we have in French for example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN"
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">

<jasperReport name="name_of_the_report" ... >
...
</jasperReports>
```

To find out what is the exact encoding type to specify when editing XML files in a particular language, you have to check the XML documentation. FIXME

4.3 Report Properties

We have already seen that `<jasperReport>` is the root element of an XML report design. In this section will get to know in detail what are the properties of a report design objects and what is are the XML attributes that correspond to them.

XML Syntax

```
<!ELEMENT jasperReport (reportFont*, parameter*, queryString?, field*,
variable*, group*, title?, pageHeader?, columnHeader?, detail?,
columnFooter?, pageFooter?, summary?)>

<!ATTLIST jasperReport
       name NMTOKEN #REQUIRED
       columnCount NMTOKEN "1"
       printOrder (Vertical | Horizontal) "Vertical"
       pageWidth NMTOKEN "595"
       pageHeight NMTOKEN "842"
       orientation (Portrait | Landscape) "Portrait"
       whenNoDataType (NoPages | BlankPage | AllSectionsNoDetail) "NoPages"
       columnWidth NMTOKEN "555"
       columnSpacing NMTOKEN "0"
       leftMargin NMTOKEN "20"
       rightMargin NMTOKEN "20"
       topMargin NMTOKEN "30"
       bottomMargin NMTOKEN "30"
       isTitleNewPage (true | false) "false"
       isSummaryNewPage (true | false) "false"
       scriptletClass NMTOKEN #IMPLIED
>
```

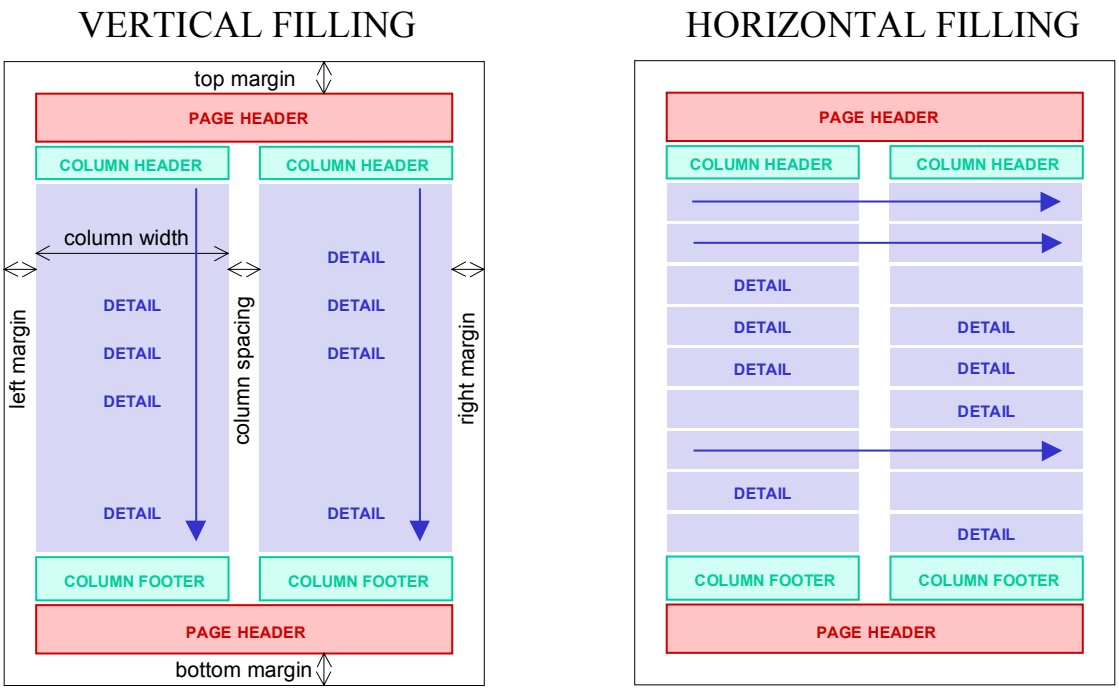
Report Name

Every report design has to have a name. Its name is important because the library uses it when generating files, especially when the default behavior is preferred for compiling, filling or exporting the report.

The name of the report is specified using the `name` attribute of the `<jasperReport>` element and is mandatory. Spaces are not allowed in the report name, which has to be one word.

Column Count

JasperReports allows creating reports with more than one column on each page, like in the following picture, where we can see the layout of a report with two columns:



- figure 6 -

By default, the reporting engine creates report with one column on each page.

Print Order

For the reports having more that one column, is important to specify the order in which the columns will be filled and this can be done using the `printOrder` attribute of the `<jasperReport>` element.

There are two possible situations:

- *Vertical filling*: Selecting this option will ensure the columns are filled from top to bottom and left to right (`printOrder="Vertical"`).
- *Horizontal filling*: Columns are filled from left to right and top to bottom (`printOrder="Horizontal"`).

The default print order is: `printOrder="Vertical"`

Page Size

There are two attributes at this level to specify the page size of the document that is going to be generated: `pageWidth` and `pageHeight`.
Like all the other JasperReports attributes that represent element dimensions and position, those should be specified in pixels. JasperReports uses the default Java resolution of 72 dots per inch.
This means that a `pageWidth="595"` will make about 8.26 inches, which is roughly the width of an A4 paper.

The default page size corresponds to an A4 paper:

```
pageWith="595" pageHeight="842"
```

Page Orientation

The `orientation` attribute is used to specify whether we are creating documents using the "Portrait" or the "Landscape" formats.

JasperReports requires you to adapt the page width and the page height when switching from "Portrait" documents to "Landscape" or vice-versa.

Let's see an example:
We assume that we want to create an A4 report using the "Portrait" layout.
An A4 has approximately this size:

```
pageWidth="595" pageHeight="842" orientation="Portrait"
```

If we decide to use the "Landscape" layout for our A4 document, we have to make sure we modify the page with and page height accordingly, like in the following:

```
pageWidth="842" pageHeight="595" orientation="Landscape"
```

This is because JasperReports has to know exactly the absolute width and height of the pages it will draw on, and does not necessarily consider the value that we supply in the orientation attribute, at least not at report filling time.

This orientation attribute is only useful at report printing time, to inform the printer about the page orientation or in some special exporters.

The default page orientation is "Portrait".

Page Margins

Once the page size decided, you can specify what margins should the reporting engine preserve when generating the reports. And there are 4 attributes for the job: `topMargin`, `leftMargin`, `bottomMargin` and `rightMargin` (figure 6).

There is a 20 pixels default margin for the top and bottom of the page and a 30 pixels default margin for the right and left margins.

Column Size and Spacing

Reports may have more that one column, as we have already seen when we have talked about the `columnCount` attribute above.
But the reporting engine has to know how large a column can be and what space should it let between columns. There are two attributes for this job: `columnWidth` and `columnSpacing`.

There is also a validation check performed when we compile the report designs, that do not let us create reports in which the width of the overall columns and the space between does not fit on the specified page width and page margins.

Since there is only one column by default, the default column spacing is 0 pixels and the default column width is equal to the default page width, minus the default left and right margins, which make 555 pixels.

Empty Data Source Behavior

Sometimes the data source that we supply to our reports has no records in it. In this case, it is not clear what the output should be.

Some may expect to see a blank document in these situations and others might want to have some of the report sections displayed anyway.

There is an attribute called `whenNoDataType` that lets you decide how the generated document should look when there is no data in the data source supplied to it.

There are 3 possibilities you can choose from:

- *Empty document*: The generated document will have no pages in it. Viewers might throw an error when trying to load such documents (`whenNoDataType="NoPages"`).
- *Blank page*: The generated document will contain a single blank page (`whenNoDataType="BlankPage"`).
- *All sections displayed*: All the report sections except the detail section will appear in the generated document (`whenNoDataType="AllSectionsNoDetail"`).

The default value for this attribute is `whenNoDataType="NoPages"`.

Title and Summary Sections Placement

If you want to have the title section or the summary section displayed on separate pages, all you have to do is to set one or both of the following attributes to "true": `isTitleNewPage` and `isSummaryNewPage`.

Both this boolean attributes are set to false by default.

Note that even if you choose to display the summary section on the remaining space of the last page, a new page will automatically be started if the report has more than one column and the second column was already started on this last page.

Scriptlet Class

The `scriptletClass` attribute lets you specify the name of the scriptlet class designed for the current report. You learn more about scriptlets in the *7 Scriptlets* chapter of this book dedicated to them.

If no value is supplied to this attribute, the reporting engine will use a `dori.jasper.engine.JRDefaultScriptlet` instance anyway.

5 Report Data

When we have talked about the report filling process, we have said that there are those 3 things that have to be supplied as input to it: the report design, the parameter values and the data source of the report.

In the previous chapter we have already seen some of the aspects regarding the reports designs and now we are going to take a close look at the other two: the parameters and the report data source. They represent the only source of data that the reporting engine will use when filling the report.
This data will be organized according to the template defined in the report design and will produce a ready to print, page oriented document, as you might expect from any reporting tool.

5.1 Expressions

Expressions are a powerful feature of JasperReports. They can be used for declaring report variables that perform various calculations, for data grouping on the report, to specify report text fields content or to further customize the appearance of objects on the report.

Basically, all report expressions are Java expressions that can reference report parameters, report fields and report variables, using a special syntax.

In an XML report design there are several elements that define expressions: `<variableExpression>`, `<initialValueExpression>`, `<groupExpression>`, `<printWhenExpression>`, `<imageExpression>`, `<textFieldExpression>` and others.

Since all the JasperReports expressions are real Java expressions, you can use in them any class that you might want to, as long as you refer to it using its complete class name (including the package). You also have to make sure the classes you are using in the report expressions are available in the classpath when you compile your report and when you fill it with data.

The report expressions would be useless, if there would be no way to reference in them the report parameters, the report fields or the declared report variables. Fortunately, there is a special JasperReports syntax that allows you to introduce such references in the expressions you create in the XML report design.

Report parameter references are introduced using the `$P{}` character sequence like in the following example:

```
<textFieldExpression>
  $P{ReportTitle}
</textFieldExpression>
```

This example assumes that we have a report parameter called `ReportTitle` declared in the report design, who's class is `java.lang.String`. The text field will display the value of this parameter when the report will be filled.

In order to use a report field reference in an expression, the name of the field must be put between `$F{` and `}` character sequences. For example, if we want to display in a text field, on the report, the concatenated values of two data source fields, we can define an expression like this one:

```
<textFieldExpression>
  $F{FirstName} + " " + $F{LastName}
</textFieldExpression>
```

The expression can be even more complex:

```
<textFieldExpression>
  $F{FirstName} + " " + $F{LastName} + " was hired on " +
  (new SimpleDateFormat("MM/dd/yyyy")).format($F{HireDate}) + "."
</textFieldExpression>
```

To reference a report variable in an expression, we must put the name of the variable between `$V{` and `}` like in the example below:

```
<textFieldExpression>
  "Total quantity : " + $V{QuantitySum} + " kg."
</textFieldExpression>
```

As you can see, the parameter, field and variable references introduced by the special JasperReports syntax are in fact real Java objects. And knowing their class from the parameter, field or variable declaration mate in the report design, we are able to even call methods on those object references in our expressions.

Here's how we might be able to extract and display the first letter from a `java.lang.String` report field:

```
<textFieldExpression>
  $F{FirstName}.substring(0, 1)
</textFieldExpression>
```

5.2 Parameters

Parameters are object references that are passed-in to the report filling operations. They are very useful for passing to the report engine data that it can not normally find in its data source.
For example, we could pass to the report engine the name of the user that has launched the report filling operation, if we want it to appear on the report, or we could dynamically change the title of our report.

XML Syntax

```
<!ELEMENT parameter (parameterDescription?, defaultValueExpression?)>

<!ATTLIST parameter
       name NMTOKEN #REQUIRED
       class NMTOKEN #REQUIRED
       isForPrompting (true | false) "true"
>

<!ELEMENT parameterDescription (#PCDATA)>

<!ELEMENT defaultValueExpression (#PCDATA)>
```

Declaring a parameter in a report design is very simple and it requires specifying only its name and its class:

```
<parameter name="ReportTitle" class="java.lang.String"/>
<parameter name="MaxOrderID" class="java.lang.Integer"/>
<parameter name="SummaryImage" class="java.awt.Image"/>
```

The supplied values for the report parameters can be used in the various report expressions, in the report SQL query or even in the report scriptlet class. We are going to see all this in the following special sections of this book that will tread each report expression, the query and the scriptlets.

Here are the components that make a report parameter definition complete:

Parameter Name

The `name` attribute of the `<parameter>` element is mandatory and allows referencing the parameter by its declared name. The naming conventions of JasperReports are similar to the naming conventions of the Java language, in regards with variable declaration. That means that the parameter name should be a single word, with no special characters in it, like a dot or a comma.

Parameter Class

The second mandatory attribute for a report parameter is the one who specifies the class name for the parameter values. The class attribute can have any value as long it represents a class name that is available in the classpath both at report compile time and report filling time.

Prompting for Parameter Values

In some GUI applications, it would be useful to have a way to establish the set of the report parameters for which the application should request user input, before launching the report filling process.
It would be also useful to have a way to specify the text description that would prompt for the user input for each of those parameters.

This is why we have the boolean `isForPrompting` attribute in the parameter declaration sequence and the inner `<parameterDescription>` element.

In the following example, we can see the declaration of a text parameter, along with the description that could be used at runtime when requesting for the user to input the parameter value, in a custom made dialog window:

```
<parameter name="Comments" class="java.lang.String" isForPrompting="true">
  <parameterDesciption>
    <![CDATA[
      Please type here the report comments if any
    ]]>
  </parameterDesciption>
</parameter>
```

You have probably noticed the `<![CDATA[` and `]]>` character sequences that delimit the parameter description. Those are part of the XML specific syntax that will instruct the XML parser to not parse the text inside. This allows you to use XML special characters like the >, <, " and others in your texts. You'll see this syntax used in other examples throughout this book and the samples.

Parameter Default Value

The parameter values are supplied to the report filling process packed in a `java.util.Map` object with the parameter names as the keys. In this way you are not obliged to supply a value for each parameter every time, if you don't want to.
If you do not supply a value for a parameter, its value will be considered to be `null`. But not if you are specifying a default value expression in the report design, for this particular report parameter. This expression is only evaluated in case you don't supply a value for the given parameter.

Here's a `java.util.Date` parameter whose value will be the current date, if you do not supply a specific date value when filling the report:

```
<parameter name="MyDate" class="java.util.Date">
  <defaultValueExpression>
    new java.util.Date()
  </defaultValueExpression>
</parameter>
```

In the default value expression of a parameter, we can only use previously defined report parameters.

5.2.1 Built-in Report Parameters

Every report design contains some predefined report parameters, along with the ones that the report design creator decides to introduce.
These built-in parameters are presented below.

Parameter `REPORT_PARAMETERS_MAP`

This is a built-in parameter that will always point to the `java.util.Map` object that contains the user-defined parameters that we pass when calling the report filling process.

This parameter is especially useful when you want to pass to the subreports the same set of report parameters that the master report has received.

Parameter `REPORT_CONNECTION`

This report parameter points to the `java.sql.Connection` object that was supplied to the engine in order to use for the execution through JDBC of the SQL report query, if it is the case.

It has a value different than `null` only if the report (or subreport) has indeed received a `java.sql.Connection` when the report filling process was launched and not a `dori.jasper.engine.JRDataSource` instance.

This is also useful for passing to the subreports the same JDBC connection object that was used by the master report. You can see this in action in the supplied *subreport* sample.

Parameter `REPORT_DATASOURCE`

When filling a report, we always have a data source object either directly supplied by the parent application or created behind the scenes by the reporting engine when a JDBC connection is supplied.
This built-in parameter will allow us access to the report's data source in the report expressions or in the scriptlets, for any reason we might have to do that.

Parameter `REPORT_SCRIPTLET`

Even if the report does not use scriptlets, this built-in parameter will point to a `dori.jasper.engine.JRAbstractScriptlet` instance, which is a `dori.jasper.engine.JRDefaultScriptlet` object, in this case.

But when using scriptlets, this reference to the scriptlet class instance that is created when filling the report would allow calling specific methods on it, to manipulate or to use the data that the scriptlet object has prepared during the filling process. This is shown on the last page of the *scriptlet* sample report when we make a call to this scriptlets object.

5.3 Data Source

When filling the report, the JasperReports engine iterates through the records of the supplied data source object and generates every section according to the template defined in the report design.

Normally, the engine expects to receive a `dori.jasper.engine.JRDataSource` object as the data source of the report that it has to fill. But as we shall see, there is a feature that lets users supply a JDCB connection object instead of the usual data source object, when the report data is found in a relational database.

The `dori.jasper.engine.JRDataSource` interface is very simple and we have to deal with only two methods if we want to implement it:

```
public boolean next() throws JRException;

public Object getFieldValue(JRField jrField) throws JRException;
```

The `next()` method is called on the data source object by the reporting engine when iterating through the data, at report filling time. The second method listed provides the value for each report field in the current data source record.

It is very important to know that the only way to retrieve data from the data source is by using the report fields. A data source object is more like a table with columns and rows containing data in the table cells. The rows of this table are the records through which the reporting engine iterates when filling the report and each column should be mapped to a report field, so that we can make use of the data source content in the report expressions.

There are several default implementations of the `dori.jasper.engine.JRDataSource` interface, and we shall take a closer look to each of them:

Class dori.jasper.engine.JRResultSetDataSource

This is a very useful implementation of the `dori.jasper.engine.JRDataSource` interface, because it wraps a `java.sql.ResultSet` object. With the majority of reports being generated using data located in relational databases, it is very likely that this is the most used implementation for the data source interface.

What is interesting to know is that you might end up using it even if you do not instantiate yourself this class, when filling your reports. This is what happens:

If you choose to specify in your report design, the SQL query to retrieve the report data from certain tables located in a relational database, the reporting engine will do that for you by executing the specified SQL query and wrapping the returned `java.sql.ResultSet` object in a `dori.jasper.engine.JRResultSetDataSource` instance. The only thing the engine would need is a `java.sql.Connection` object for it to execute the query as mentioned. You supply this connection object instead of supplying the usual data source object.

This can be seen in samples like *jasper*, *scriptlet*, *subreport* and *query*.

Of course, you could execute the SQL query in the parent application, outside JasperReports, if you want to or the situation forces you to. And then you could manually wrap the `java.sql.ResultSet` obtained, using an instance of this data source class, before calling the report filling process.

The most important thing you should know when using this type of data source is that you have to declare a report field for each column in the result set. The name of the report field has to be same as the name of the column it maps as well as the data type.
For maximum portability, as stated in the JDBC documentation, the values from a `java.sql.ResultSet` object should be retrieved from left to right and only once, so in order to do that you might consider declaring the report fields in the same order as they appear in the SQL query.

Class `dori.jasper.engine.JREmptyDataSource`

As already mentioned, this implementation is a very handy one, since it allows generating reports in which the data do not necessarily comes from the data source, but probably from the parameters and only the number of the virtual records in the data source is important.

Many of the provided samples such as *fonts*, *images*, *shapes* and *unicode* use an instance of this class when filling the reports, to simulate a data source with one record in it, but with all the fields `null`.

Class `dori.jasper.engine.data.JRTableModelDataSource`

This default implementation of the `dori.jasper.engine.JRDataSource` interface wraps a `javax.swing.table.TableModel` object and can be used in Java Swing application to generate reports using data that has already been loaded into on-screen tables.

There are two ways to use this type of data source:

Normally, in order to retrieve data from it, you have to declare a report field for each column in the `javax.swing.table.TableModel` object, baring the same name as the column it maps.
But sometimes this is impossible since the report fields have to respect the Java naming conventions for declaring variables and the table columns do not necessarily do that.

Fortunately, you can still map such columns into report fields using their index instead of their names.
For example, a table column named "Product Description" could never be mapped into a report field named "Product Description", because of the repot compiling error you will get due to the space character present in the report field name.
But if you know that this particular column is the third column in the table model object (index = 2), then you could name the corresponding field "COLUMN_2" and use the column data without problems.

An example is provided in the *datasource* sample.

Class `dori.jasper.engine.data.JRBeanArrayDataSource`

This implementation that is also shipped with the library wraps an array of JavaBeans and uses reflection to retrieve report field values.

A JavaBean object represents one record in this type of data source. If we have a report field name "ProductDescription", when retrieving the value for this field, the program will try to call through reflection a method called `getProductDescription()` on the current JavaBean object.

For the boolean fields, the program will also try the ***is*** prefix, if the usual ***get*** prefix for the JavaBean object properties fails to return a value, as stated in the JavaBeans specifications.

This class is used for demonstrative purposes in the *datasource* sample.

Class `dori.jasper.engine.data.JRBeanCollectionDataSource`

This implementation of the `dori.jasper.engine.JRDataSource` interface is very similar to the previous one in that it uses reflection and the JavaBeans naming conventions, but wraps a `java.util.Collection` object in stead of an array of JavaBean objects.

You can see this class being used also in the *datasource* sample provided with the project.

5.4 Report Query

In order to fill a report, we have to provide the reporting engine with the report data, or at least instruct it about how to get this data for us.

JasperReports normally expects to receive a `dori.jasper.engine.JRDataSource` object as the report data source, but it was also enhanced to work with JDBC so that it can retrieve data from relation databases if required.

The library allows users to specify in their report design, the SQL query that should be executed at runtime to retrieve the report data, if this data is located in relational databases, which is a very common situation.

The SQL query specified in the report design is taken into account and executed only if a `java.sql.Connection` object is supplied instead of the normal `dori.jasper.engine.JRDataSource` object when filling the report.

This query can be introduced in the XML report design using the `<queryString>` element. If present, this element comes after the report parameter declarations and before the report fields.

XML Syntax

```
<!ELEMENT queryString (#PCDATA)>
```

Here's is a simple SQL query that will retrieve data from a table called Orders placed in a relational database:

```
<queryString><![CDATA[SELECT * FROM Orders]]></queryString>
```

An import aspect is the use of report parameters in the query string of the report, in order to be able to further customize the data set retrieved from the database. Those parameters could act like dynamic filters in the query that supplies data for the report and are introduced using a special syntax, similar to the one used in report expressions.

There are two possible ways to use parameters in the query:

1. The parameters are used like normal `java.sql.PreparedStatement` parameters, using the following syntax:

```
<queryString>
  <!CDATA[
   SELECT * FROM Orders WHERE OrderID <= $P{MaxOrderID} ORDER BY ShipCountry
  ]]>
</queryString>
```

2. Sometimes is useful to use parameters to dynamically modify portions of the SQL query or to pass the entire SQL query as a parameter to the report filling routines. In such a case, the syntax differs a little, like in the following example. Notice the **!** character:

```
<queryString>
  <!CDATA[
    SELECT * FROM $P!{MyTable} ORDER BY $P!{OrderByClause}
  ]]>
</queryString>
```

What is different in this second example? The parameters used make it for the missing table name in the FROM clause and the missing column names in the ORDER BY clause. You cannot use normal IN parameters to dynamically change portions of your query that you execute using a `java.sql.PreparedStatement` object, and this seems to be the case here.

The special syntax that introduces the parameter values in this example makes sure that the value that we supply for those parameters will replace the parameter references in the query, before it is sent to the database server using a `java.sql.PreparedStatement` object.

In fact what happens is that the reporting engine first deals with the `$P!{}` parameter references by using their values to obtain the final form of the SQL query and only after that will transform the rest of the `$P{}` normal parameter references into usual IN parameters used when working with JDBC prepared statements.

For more details about what type of parameters you have to use in your report queries, you have to be more familiar with the JDBC technology and especially with the use of the `java.sql.PreparedStatement` interface and its parameters.

This second type of parameter reference used in the SQL query allows you to pass the entire SQL query at runtime if you like:

```
<queryString>$P!{MySQLQuery}</queryString>
```

You cannot put other parameter references into a parameter value. That is, when supplying the entire SQL query as a report parameter, the engine does not expect to find parameter references in the query you send to it and considers this query to be finished and ready to send "as is", to the database server.

Some of the provided samples like *jasper*, *subreport*, *scriptlet*, and *webapp* use internal SQL queries to retrieve data. From this point of view, the most interesting sample for you to check is the one called *query*.

5.5 Fields

The report fields represent the only way to map data from the data source into the report design and to use this data in the report expressions, to obtain the desired output.

When declaring report fields, you have to make sure that the data source you will supply at report filling time will be able to provide values for all those fields.
For example, in case you use the `dori.jasper.engine.JRResultSetDataSource` implementation, and this happens when the report's SQL query is used, you have to make sure there will be a column for each field in the result set obtained after the execution of the query. The corresponding column has to bare the same name and have the same data type as the filed that maps it.

XML Syntax

```
<!ELEMENT field (fieldDescription?)>

<!ATTLIST field
      name NMTOKEN #REQUIRED
      class (java.lang.Object | java.lang.Boolean | java.lang.Byte |
java.util.Date | java.sql.Timestamp | java.lang.Double | java.lang.Float |
java.lang.Integer | java.io.InputStream | java.lang.Long | java.lang.Short |
java.math.BigDecimal | java.lang.String) "java.lang.String"
>

<!ELEMENT fieldDescription (#PCDATA)>
```

Here's small example that shows what fields you have to declare to map the columns of a certain database table. We shall consider a table called Employees, with the following structure:

Column Name	Data Type	Length
EmployeeID	int	4
LastName	varchar	50
FirstName	varchar	50
HireDate	datetime	8

The report fields should declare the field as seen below:

```
<field name="EmployeeID" class="java.lang.Integer"/>
<field name="LastName" class="java.lang.String"/>
<field name="FirstName" class="java.lang.String"/>
<field name="HireDate" class="java.util.Date"/>
```

If we declare a field that does not have a corresponding column in the result set, an exception will be thrown at runtime. The columns that are present in the result set produced by the execution of the SQL query that do not have corresponding fields in the report design, do not affect the report filling operations, but they also wont be accessible for display on the report.

Here are the components of a report field definition:

Field Name

The `name` attribute of the `<field>` element is mandatory and allows referencing the field in report expressions, by its declared name, which should be a single word, without special characters in it, like a dot or a comma.

Field Class

The second attribute for a report field is the one who specifies the class name for the field values.
Its default value is `java.lang.String`, but it can be changed to one of the following acceptable class names:

```
java.lang.Object
java.lang.Boolean
java.lang.Byte
java.util.Date
java.sql.Timestamp
java.lang.Double
java.lang.Float
java.lang.Integer
java.io.InputStream
java.lang.Long
java.lang.Short
java.math.BigDecimal
```

When working with custom made data sources that have fields with values that are instances of custom made classes, the class name of those particular fields should be `java.lang.Object`, because unlike the parameter definitions, you can only chose a class name from the above list.
When working with those fields in the report expressions, you can cast them to their known class, making sure that class is available in the classpath, both at report compiling time and report filling time.

Field Description

This additional text chunk that can accompany a field might prove very useful when implementing a custom data source for example. You could store in it a key or whatever information you might need in order to retrieve the field's value from the custom data source at runtime.

By using this optional `<fieldDesciption>` element, you could easily overcome restrictions regarding the field naming conventions, using the field description instead of the field name, when retrieving the field values from the data source.

```
<field name="PersonName" class="java.lang.String" isForPrompting="true">
  <fieldDesciption>PERSON NAME</fieldDesciption>
</field>
```

5.6 Variables

Report variables are special objects build on top of a report expression. They can be used to simplify the report design by declaring only once an expression that is heavily used throughout the report design or to perform various calculations on the corresponding expression.

XML Syntax

```
<!ELEMENT variable (variableExpression?, initialValueExpression?)>

<!ATTLIST variable
       name NMTOKEN #REQUIRED
       class NMTOKEN "java.lang.String"
       resetType (None | Report | Page | Column | Group) "Report"
       resetGroup CDATA #IMPLIED
       calculation (Nothing | Count | Sum | Average | Lowest | Highest |
StandardDeviation | Variance | System) "Nothing"
>

<!ELEMENT variableExpression (#PCDATA)>

<!ELEMENT initialValueExpression (#PCDATA)>
```

In its expression, a variable can reference other report variables, but only if those referenced variables were previously defined in the report design. So, the order in which the variables are declared in a report design is very important.

Variable Name

Just like for the parameters and fields, the `name` attribute of the `<variable>` element is mandatory and allows referencing the variable by its declared name in the report expressions. The naming conventions that we mentioned for the parameters and fields also apply to variables.

Variable Class

The class attribute contains the class name of the class to which the variable values belong. The default is `java.lang.String`, but you can declare report variables of any class you might want to as long as the class you chose is available in the classpath, both at report compilation time and report filling time.

Reset Type

The value of a report variable can change with every iteration, but it can be brought back to the value returned by its initial value expression at specified moments in time, during the report filling process. This behavior is controlled using the `resetType` attribute, which indicates when the variable should be reinitialized during the report filling process.

There are 5 reset types for a variable:

- *No Reset*: The variable will never be initialized using its initial value expression and will only contain values obtain by evaluating the variable's expression (`resetType="None"`).
- *Report Level Reset*: The variable is initialized only once, at the beginning of the report filling process, with the value returned by the variable's initial value expression (`resetType="Report"`).
- *Page Level Reset*: The variable is reinitialized at the beginning of each new page (`resetType="Page"`).
- *Column Level Reset*: The variable is reinitialized at the beginning of each new column (`resetType="Column"`).
- *Group Level Reset*: The variable is reinitialized every time the group specified by the `resetGroup` attributes breaks (`resetType="Group"`).

The default value for this attribute is `resetType="Report"`.

Reset Group

If present, the `resetGroup` attribute contains the name of a report group and works only in conjunction with the `resetType` attribute, who's value must be `resetType="Group"`.

5.6.1 Calculations

As mentioned, variables can perform built-in types of calculations on their corresponding expression values. Here are all the possible values for the `calculation` attribute of the `<variable>` element:

Calculation `Nothing`

This is the default calculation type that a variable performs. It means that the variable's value is recalculated with every iteration in the data source and that the value returned is obtained by simply evaluating the variable's expression.

Calculation `Count`

A count variable will always count for the non-null values returned after evaluating the variable main expression, with every iteration in the data source. Count variables must be always of a numeric type, but they can have non-numeric expressions as their main expression, since the engine does not care about the expression type, but only counts for the non-null values returned, regardless of their type.

Only the variable's initial value expression should be also numeric and compatible with the variable's type, since this value will be directly assigned to the count variable when initialized.

Calculation `Sum`

The reporting engine can sum up the values returned by the variable's main expression if you choose this type of calculation, but make sure the variable has a numeric type. We cannot calculate the sum of a `java.lang.String` or `java.util.Date` type of a report variable.

Calculation `Average`

The reporting engine can also calculate the average for the series of values obtained by evaluating the variable's expression for each record in the data source. This type of calculation can also be performed only for numeric variables.

In order to calculate the average, the engine creates behind the scenes a helper report variable that calculates the sum of the values and uses it to calculate the average for those values. This helper sum variable gets its name from the corresponding average variable suffixed with "_SUM" sequence.

For example, if you declare an average calculating variable named `MyAverageVariable` for some numeric expression, the reporting engine will also create a report variable `MyAverageVariable_SUM` to help calculating the average.

You can use the helper variables in other report expressions, if you want to, just like you would have declared them yourself.

The reporting engine also needs a count variable along with the helper sum variable, in order to calculate the average. But for "Report", "Page" and "Column" reset types, it makes use of the built-in count variables that we shall see in the following section (*5.6.2 Built-in Report Variables*). For the `resetType="Group"` case, it creates a helper count variable, behind the scenes, with the "_COUNT" suffix added to the original average variable name. This is because it cannot use the built-in group count variable due to the order in which the variables are declared and thus evaluated.

Calculation `Lowest` and `Highest`

You can choose this type of calculation when you want to obtain the lowest or the highest value in the series of values obtained by evaluating the variable's expression for each data source record.

Calculation `StandardDeviation` and `Variance`

In some special reports, you might want to perform more advanced types of calculations on numeric expression and JasperReports has built-in algorithms to obtain the standard deviation and the variance for the series of values returned by evaluation a report variable's expression.

Just like for the variables that calculate the average, the engine creates and uses helper report variables for first obtaining the sum and the count that correspond to your current series of values. The name for those helper variables that are created behind the scenes is obtained by suffixing the user variable with the "_SUM" or "_COUNT" suffix and they can be used in other report expressions like any other report variable.

For variables that calculate the standard deviation, there is always a helper variable present, that first calculates the variance for the series of values and it has the "_VARIANCE" suffix added to its name.

Calculation `System`

This type of calculation can be chose only when you don't want the engine to take care of calculating any value for your variable. That means you are calculating the value for that variable yourself, almost certainly using the scriptlets functionality of JasperReports.

For this type of calculation, the only thing the engine does is to conserve the value you have calculated yourself, from one iteration in the data source, to the next.

Examples:

Here is a simple report variable declaration that calculates the sum for a numeric report field called "Quantity":

```
<variable name="QuantitySum" class="java.lang.Double" calculation="Sum">
  <variableExpression>$F{Quantity}</variableExpression>
</variable>
```

If we want to have the sum of this field for each page, here's the complete variable declaration required:

```
<variable name="QuantitySum" class="java.lang.Double" resetType="Page"
calculation="Sum">
  <variableExpression>$F{Quantity}</variableExpression>
  <initialValueExpression>new Double(0)</initialValueExpression>
</variable>
```

In this example above, our page sum variable will be initialized with zero at the beginning of each new page.

5.6.2 Built-in Report Variables

There are also the following built-in system variables, ready to use in expressions:

Variable `PAGE_NUMBER`

This variable contains as its value the current page number. At the end of the report filling process, it will contain the total number of pages for the resulting document.

It can be used to display both the current page number and the total number of pages using a special feature of the JasperReports text field elements, which is the `evaluationTime` attribute. You can see this happening in most of the sample. Check the *jasper* sample for an example.

Variable `COLUMN_NUMBER`

Built-in variable that contains the current column number.
For example, on a report with 3 columns, on the second page, we shall probably have columns #4, #5 and #6, if attributes such as `isTitleNewPage` and other do not intervene.

Variable `REPORT_COUNT`

After the finishing the iteration through the data source, this report variable contains the total number of the records that were processed.

Variable `PAGE_COUNT`

This variable contains the number of records that were processed when generating the current page.

Variable `COLUMN_COUNT`

This variable contains the number of records that were processed when generating the current column.

Variable `GroupName_COUNT`

When declaring a report group, the engine will automatically create a count variable that will calculate the number of records that make up the current group (number of records processed between group ruptures).

The name for this variable comes from the name of the group it corresponds to, suffixed with the "_COUNT" sequence. It can be use like any other report variable, in any report expression (even in the current group expression like you can see done in the "BreakGroup" of the *jasper* sample).

6 Report Sections

When building a report design we need to define the content and the layout of its sections. The entire structure of the report design is based on the following sections: `<title>`, `<pageHeader>`, `<columnHeader>`, `<groupHeader>`, `<detail>`, `<groupFooter>`, `<columnFoter>`, `<pageFooter>`, `<summary>`.

Sections are portions of the report template that have a specified height and width and can contain report elements like lines, rectangles, images or text fields. Those sections are filled repeatedly at report generating time and make up the final document that is being produced. When declaring the content and layout of a report section, in an XML report design, we use the generic element `<band>`.

XML Syntax

```
<!ELEMENT band (printWhenExpression?, (line | rectangle | image | staticText
| textField | subreport | elementGroup)*)>

<!ATTLIST band
       height NMTOKEN "0"
>

<!ELEMENT printWhenExpression (#PCDATA)>
```

Report sections are sometimes referred as report bands and represent a feature that almost all report tools have and use in the same way.

Band Height

The attribute height available in a report band declaration specifies the height in pixels for that particular band and is very important in the overall report design.

The elements contained by a certain report band should always fit the band's dimensions, to avoid potential bad results when generating the reports. The engine issues a warning if it finds elements outside the band borders, when compiling report designs.

Skipping Bands

All the report sections allow you to define a report expression that will be evaluated at runtime in order to decide if that particular section should be generated or skipped, when producing the document.

This expression is introduced by the `<printWhenExpression>` that is available in any `<band>` element of the XML report design and should always return a `java.lang.Boolean` object or `null`.

6.1 Main Report Sections

A minimal report design can contain no report section at all, because each one of them is optional. But such a minimal report design won't produce very interesting documents.

XML Syntax

```
<!ELEMENT title (band?)>

<!ELEMENT pageHeader (band?)>

<!ELEMENT columnHeader (band?)>

<!ELEMENT detail (band?)>

<!ELEMENT columnFooter (band?)>

<!ELEMENT pageFooter (band?)>

<!ELEMENT summary (band?)>
```

So let's take a closer look at each report section and see how it behaves.

Title

This is the first section of the report. It is generated only once during the report filling process and makes it for the beginning of the resulting document.

Being the first section of the report means that it will precede even the page header section. Those who want to have the page header printed somehow before the title section will have to copy the elements present on the page header also at the beginning of the title section. They could suppress the actual page header on the first page using the `<printWhenExpression>`, based on the `PAGE_NUMBER` report variable.

As we have already seen in the *4.3 Report Properties* paragraph, the title section could be followed by a page break, if the attribute `isTitleNewPage` is set to `"true"`.

Page Header

This section appears at the top of each page in the generated document.

Column Header

This section appears at the top of each column in the generated document.

Detail

For each record in the data source, the engine will try to generate this section.

Column Footer

This section appears at the bottom of each column in the generated document. It never stretches downward to acquire the content of its containing text fields and will always remain of declared fixed height.

Page Footer

This section appears at the bottom of each page in the generated document. Just like the column footer section above, the page footer never stretches downwards to acquire the content of its containing text fields and will always remain of declared fixed height.

Summary

This section is generated only once per report and appears at the end of the generated document, but is not necessarily the last section being generated.

That's because in some cases, the column footer or/and page footer of the last page can follow it.

As mentioned in the *4.3 Report Properties* paragraph, the summary section can start a new page of its own, by setting the `isSummaryNewPage` attribute to `"true"`. Even if this attribute remains `false`, the summary section always starts a new page if it does not fit on the remaining space of the last page or if the report has more than one column and on the last page it has already started a second column.

If the main report sections that we have seen here are not sufficient for what you need, maybe you should consider introducing supplementary sections like group headers and group footers.
We are now going to see how to group data on the report.

6.2 Data Grouping

Groups represent a flexible way to organize data on a report. A report group is represented by sequence of consecutive records in the data source that have something in common, like the value of a certain report field for example.

A report group has 3 components:
- group expression;
- group header section;
- group footer section.

The value of the associated group expression is what makes group records stick tighter. This value is the thing that they have in common. When the value of the group expression changes during the iteration through the data source at report filling time, a group rupture occurs and the corresponding group sections `<groupFooter>` and `<groupHeader>` are inserted in the resulting document.

We can have as many groups as we want on a report. The order of groups declared in a report design is important because groups contain each other. One group contains the following group and so on. And when a larger group encounters a rupture, all subsequent groups are reinitialized.

Data grouping works as expected only when the records in the data source are already ordered accordingly to the group expressions used in the report.
For example, if you want to group some products by country and city of the manufacturer, the engine expects to find the records in the data source already ordered by country and city. If not, you should expect to find records belonging to a specific country or city in different parts of the resulting document, because JasperReports does not sort the data source for you, before using it.

XML Syntax

```
<!ELEMENT group (groupExpression?, groupHeader?, groupFooter?)>

<!ATTLIST group
       name NMTOKEN #REQUIRED
       isStartNewColumn (true | false) "false"
       isStartNewPage (true | false) "false"
       isResetPageNumber (true | false) "false"
       isReprintHeaderOnEachPage (true | false) "false"
       minHeightToStartNewPage NMTOKEN "0"
>

<!ELEMENT groupExpression (#PCDATA)>

<!ELEMENT groupHeader (band?)>

<!ELEMENT groupFooter (band?)>
```

Group Name

The name unequivocally identifies the group and can be used in other XML attributes, when you want to refer a particular report group. The name of a group is mandatory and obeys the same naming convention that we mentioned for the report parameters, fields and report variables.

Starting New Page/Column When Group Breaks

Sometimes is useful to introduce a page or column break when a new group starts, probably because that particular group is more important and should start on a page or column of its own.
To instruct the engine to start a new page or column for a certain group, instead of going to print it on the remaining space at the bottom of the page or column, you have to set to `"true"` either the `isStartNewPage` or `isStartNewColumn` attribute.

Those two attributes are the only settings in the entire library that let you voluntary introduce page breaks. In all other situation, the reporting engine introduces page breaks automatically, if it needs to.
However, in some report designs, you probably want to introduce page breaks on purpose, because some particular report section of yours is larger than one page. You can achieve that by introducing special dummy groups as you can see in the *Tips & Tricks* section of the freely available documentation, published on the *JasperReports site*.

However, if you don't want to consistently introduce page or column breaks for a particular group, but you rather do that only if the remaining space at the bottom of the page or column is too small, you should consider using the `minHeightToStartNewPage` attribute.
This attribute specifies the minimum amount of remaining vertical space required so that the group does not start a new page of its own. It is measured in pixels.

Resetting Page Number

If required, report groups have the power to reset the built-in report variable which contains the current page number (variable `PAGE_NUMBER`). This could be achieved by setting the `isResetPageNumber` attribute to `"true"`.

Group Header

This section is the one that marks the start of a new group in the resulting document, and it is inserted in the document every time the value of the group expression changes during the iteration through the data source.

Group Footer

Every time a report group changes, the engine adds the corresponding group footer section before starting the new group or when the report ends.

Check the provided samples like *jasper*, *datasource* or *query*, to see how report groups can be used.

7 Scriptlets

All the data displayed on a report comes from the report parameters and from the report fields. This data can be processed using the report variables and their expressions.
There are specific moments in time when variable processing occurs. Some variables are initialized according to their reset type when the report starts, or when a page or column break is encountered, or when a group changes. Furthermore, variables are evaluated every time new data is fetched from the data source (for every row).

But only simple variable expressions cannot always implement complex functionality. This is where scriptlets intervene. Scriptlets are sequences of Java code that are executed every time a report event occurs. Through scriptlets, users have the possibility to affect the values stored by the report variables. Since scriptlets work mainly with report variables, is important to have full control over the exact moment the scriptlet is executed.

JasperReports allows the execution of custom Java code BEFORE or AFTER it initializes the report variables according to their reset type: `Report`, `Page`, `Column` or `Group`.
In order to make use of this functionality, users only have to create a scriptlet class by extending one of the following two classes:

```
dori.jasper.engine.JRAbstractScriptlet
dori.jasper.engine.JRDefaultScriptlet
```

The complete name of this custom scriptlet class (including the package) has to be specified in the `scriptletClass` attribute of the `<jasperReport>` element and has to be available in the classpath, at report filling time, so that the engine could instantiate it on the fly. If no value is specified for the `scriptletClass` attribute, the engine will instantiate the `JRDefaultScriptlet` class.

When creating a JasperReports scriptlet class, there are several methods that developers should implement or override, like: `beforeReportInit()`, `afterReportInit()`, `beforePageInit()`, `afterPageInit()`, `beforeGroupInit()`, `afterGroupInit()`, etc. Those methods will be called by the report engine at the appropriate time, when filling the report.

For more complex reports, if you need to use very complicate report expressions, for grouping or displaying data, maybe you should consider transferring this complexity to a separate class to which you then make calls from simplified report expressions. The scriptlet class is perfect for transferring this complexity to. This is because the reporting engine supplies you with a reference to the scriptlet object it creates on the fly using the `REPORT_SCRIPTLET` built-in parameter.

Check the *scriptlet* sample to see this type of functionality used.

8 Report Elements

The generated reports would be empty if you would not put some report elements in the report design. The report elements are displayable objects like static texts, text fields, images, lines or rectangles, that you put in your report design sections so that they appear in the final document.

As you can see, the report elements come in two flavors:
- *Text elements*: static texts and text fields that display dynamic content;
- *Graphic elements*: lines, rectangles and images.

We shall see those two element categories and their particularities in the following sections. For now we are going to present in detail the element properties that both categories share.

When you add a report element to one of your report sections, you have to specify the relative position of this element in that particular section and its size, along with other general report element properties like color, transparency, stretch behavior, etc.

The properties that are common to all types of report elements are grouped in the `<reportElement>` tag that can appear in the declaration of all report elements.

XML Syntax

```
<!ELEMENT reportElement (printWhenExpression?)>

<!ATTLIST reportElement
       positionType (Float | FixRelativeToTop | FixRelativeToBottom)
"FixRelativeToTop"
       isPrintRepeatedValues (true | false) "true"
       mode (Opaque | Transparent) #IMPLIED
       x NMTOKEN #REQUIRED
       y NMTOKEN #REQUIRED
       width NMTOKEN #REQUIRED
       height NMTOKEN #REQUIRED
       isRemoveLineWhenBlank (true | false) "false"
       isPrintInFirstWholeBand (true | false) "false"
       isPrintWhenDetailOverflows (true | false) "false"
       printWhenGroupChanges CDATA #IMPLIED
       forecolor CDATA #IMPLIED
       backcolor CDATA #IMPLIED
>

<!ELEMENT printWhenExpression (#PCDATA)>
```

Absolute Position

The `x` and `y` attributes of any report element are mandatory and represent x and y coordinates, measured in pixels, that mark the absolute position of the top-left corner of the specified element within its parent report section.

Relative Position

Some report elements such as the text fields have special properties that allow them to stretch downwards in order to acquire all the information they have to display. Their height is calculated at runtime and may affect the other neighboring elements present in the same report section, especially those placed immediately below them.

The `positionType` attribute specifies the behavior that the report element should have if the layout of the report section in which it is been place is affected by stretch.

There are 3 possible values for the `positionType` attribute:

- *Floating position*: The element will float in its parent section if it is pushed downwards by other elements fount above it. It will try to conserve the distance between it and the neighboring elements placed immediately above (`positionType="Float"`).
- *Fixed position relative to the top of the parent band*: The current report element will simply ignore what happens to the other section elements and tries to conserve the y offset measured from the top of its parent report section (`positionType="FixRelativeToTop"`).
- *Fixed position relative to the bottom of the parent band*: If the height of the parent report section is affected by elements that stretch, the current element will try to conserve the original distance between its bottom margin and the bottom of the band (`positionType="FixRelativeToBottom"`).

A report element called `e2` will float when another report element called `e1` stretches, only if these three conditions are met:

- ✓ `e2` has `postitionType="Float"`
- ✓ `e1.y + e1.height < e2.y`
- ✓ `e1.width + e2.width >= max(e1.x + e1.width, e2.x + e2.width) - min(e1.x, e2.x)`

The second and the third conditions together say that the element `e2` must be placed below the `e1`.

By default, all elements have a fixed position relative to the top of the band.

To see how element stretching and element floating work together, check the *stretch* sample provided.

Element Size

The `width` and `height` attributes are mandatory and represent the size of the report element measured in pixels. Additional element settings that have to do with the element stretching mechanism will determine the reporting engine to sometimes ignore the specified element height. But this attribute remains mandatory since even when the height is calculated dynamically, the element will not be smaller than the original specified height.

Element Color

There are two attributes that represent colors: `forecolor` and `backcolor`. The fore color is the one used to draw the text of the text elements and the border of the graphic elements. The back color is the one used to fill the background of the specified report element, if it is not transparent.

You could specify colors using the decimal or hexadecimal representation of the integer number corresponding to the desired color. The preferred way to specify colors in XML is using the hexadecimal representation, because it allows controlling the level for each base color of the RGB system.

For example, you can display some text in red if you set the `forecolor` attribute of the corresponding text field like this:

```
forecolor="#FF0000"
```

The equivalent using the decimal representation would be:

```
forecolor="16711680"
```

but the inconvenience is evident.

The default fore color is black and the default back color is white.

Element Transparency

Report elements can be either transparent or opaque, depending on the value you specify for the attribute `mode`.

The default value for this attribute depends on the type of the report element. Graphic elements like rectangles and lines are opaque by default, but the images are transparent. Both static texts and text fields are transparent by default, and so are the subreport elements.

Skipping Element Display

The engine can decide at runtime if it really should display a report element, if you use the `<printWhenExpression>` that is available for all types of report elements.
If present, this report expression should return a `java.lang.Boolean` object or `null` and is evaluated every time the section containing the current element is being generated, to see if this particular element should appear or not in the report.
If the expression returns `null`, it is equivalent to returning `java.lang.Boolean.FALSE` and if the expression is missing, the report element will get printed every time, that is if other setting do not intervene, as we shall see below.

Reprinting Elements on Section Overflows

When generating a report section, the engine might be forced to start a new page or column, because the remaining space at the bottom of the current page or column was not sufficient for all the section elements to fit in, probably because some elements have stretched.

In such cases, you might want to reprint some of your already displayed elements, on the new page or column, to recreate the context in which the page/column break occurred.
To achieve this, you have to set `isPrintWhenDetailOverflows="true"` for all those report elements you want to reappear on the next page or column.

Suppressing Repeating Values Display

First, let's see what exactly a "repeating value" is.

It very much depends on the type of the report element we are talking about.

For text field elements, this is very intuitive. In the following list containing person names taken from an usual phone book, you can see that for some consecutive lines, the value of the "Family Name" column repeats itself (those are only dummy phone numbers ☺).

Family Name	First Name	Phone
Johnson	Adam	256.12.35
Johnson	Christine	589.54.52
Johnson	Peter	546.85.95
Johnson	Richard	125.49.56
Smith	John	469.85.45
Smith	Laura	459.86.54
Smith	Denise	884.51.25

You might want to suppress the repeating "Family Name" values and print something like this:

Family Name	First Name	Phone
Johnson	Adam	256.12.35
	Christine	589.54.52
	Peter	546.85.95
	Richard	125.49.56
Smith	John	469.85.45
	Laura	459.86.54
	Denise	884.51.25

You can do that, if for the text field that displays the family name, you set:

```
isPrintRepeatedValues="false"
```

The static text elements behave in the same way. As you would expect, their value always repeats and in fact it never changes, until the end of the report. This is why we call them static texts. So, if you set `isPrintRepeatedValues="false"` for one of your `<staticText>` elements, you should expect to see it displayed only once, the first time, at the beginning of the report, and never again.

Now, what about graphic elements?

An image is considered to be repeating itself if its bytes are exactly the same from one occurrence to the next. This could only happen if you choose to cache your images using the `isUsingCache` attribute available in the `<image>` element and if the corresponding `<imageExpression>` returns the same value from one iteration to the next (the same file name, the same URL, etc).

Lines and rectangles are always repeating themselves, because they are static elements, just like the static texts we have seen above. So, when deciding to not display repeating values for a line or a rectangle, you should expect to see it displayed only once, at the beginning of the report and then ignored until the end of the report.

The `isPrintRepeatedValues` attribute works only if the corresponding `<printWhenExpression>` is missing. If this is not missing, it will always dictate if the element should be printed or not, regardless of the repeating values.

If you decide to not display the repeating values for some of your report elements, you have the possibility to soften or refine this behavior, by indicating the exceptional occasions to which you might want to have a particular value redisplayed, during the report generation process.

When the repeating value spans on multiple pages or columns, you have the possibility to redisplay this repeating value at least once for every page or column.
By setting `isPrintInFirstWholeBand="true"`, you make sure that the report element will reappear in the first band of a new page or column that is not an overflow from a previous page or column.

Also, if the repeating value you have suppressed spans on multiple groups, you have the possibility to make it reappearing at the beginning of a certain report group, is you specify the name of that particular group in the `printWhenGroupChanges` attribute.

Removing Blank Space

When report elements are not displayed for some reason: `<printWhenExpression>` evaluated to `Boolean.FALSE`, or repeated value being suppressed, a blank space remains where that report element would have stood.
This blank space also appears if a text field displays only blank characters or an empty text.

There is a way to eliminate this unwanted blank space, on the vertical axis, only if some conditions are met.

For example, if you have three successive text fields, one on top of the other like this:

```
TextField1
TextField2
TextField3
```

If the second one has an empty string as its value, or contains a repeated value that you chose to suppress, the output would look like this:

```
TextField1

TextField3
```

In order to eliminate the gap between the first text field and the third, you have to set `isRemoveLineWhenBlank="true"` for your second text field. You would obtain something like this:

```
TextField1
TextField3
```

But there are certain conditions that have to be met in order for this functionality to work. The blank space will not be removed, if your second text field shares some vertical space with other report elements that are printed even this second text fields of your does not print.

For example, you might have some vertical lines on the sides of your report section like this:

```
|   TextField1   |
|                |
|   TextField3   |
```

or you might have a rectangle that draws a box around your text fields:

```
------------------
|   TextField1   |
|                |
|   TextField3   |
------------------
```

or even other text elements that are placed on the same horizontal with your second text field:

```
Label1   TextField1
Label2
Label3   TextField3
```

In all those situations, the blank space between the first and the third text field cannot be remove, because it is being used by other report elements that are printed as you can see.

The blank vertical space between elements can be removed using the `isRemoveWhenBlank` attribute, only if it is not used by other elements, as explained above.

8.1 Text Elements

There are two kinds of text elements in JasperReports: static texts and text fields.
As their names suggest it, the first are text elements with a fixed, static content, who does not change during the report filling process and are used especially for introducing labels on the final document.
Text fields however, have an associated expression, which is evaluated at runtime to produce the text content that will be displayed.

Both types of text elements share some properties and those are introduced using a `<textElement>` element. We are now going to see them in detail.

XML Syntax

```
<!ELEMENT textElement (font?)>

<!ATTLIST textElement
      textAlignment (Left | Center | Right | Justified) "Left"
      lineSpacing (Single | 1_1_2 | Double) "Single"
>
```

Text Alignment

You can specify how the content of a text element should be aligned using the `textAlignment` attribute and choosing one of the 4 possible values: `"Left"`, `"Center"`, `"Right"` or `"Justified"`.

Text Line Spacing

The amount of space between consecutive lines of text can be set using the `lineSpacing` attribute:

- *Single*: The paragraph text advances normally using an offset equal to the text line height (`lineSpacing="Single"`).
- *1.5 Lines*: The offset between two consecutive text lines is of 1 ½ lines (`lineSpacing="1_1_2"`).
- *Double*: The space between text lines is double the height of a single text line (`lineSpacing="Double"`).

The font settings for the text elements are also part of the `<textElement>` tag, but we are going to see them in detail, in the following separate section of this book.

8.1.1 Fonts and Unicode Support

Each text element present on your report can have its own font settings. Those settings can be specified using the `<font>` tag available in the `<textElement>` tag.

Since most of the time, in a report design, there are only a few types of fonts used, that are shared by different text elements, there's no point forcing XML report design creators to specify the same font settings for each text element, over and over again. But rather they could reference a report level font declaration and adjust only some of the font settings, on the spot, if a particular text element requires it.

Report Fonts

A report font is in fact a collection of font settings declared at report level that can be reused throughout the entire report design, when setting the font properties of text elements.

XML Syntax

```
<!ELEMENT reportFont EMPTY>

<!ATTLIST reportFont
       name NMTOKEN #REQUIRED
       isDefault (true | false) "false"
       fontName CDATA "sansserif"
       size NMTOKEN "10"
       isBold (true | false) "false"
       isItalic (true | false) "false"
       isUnderline (true | false) "false"
       isStrikeThrough (true | false) "false"
       pdfFontName CDATA "Helvetica"
       pdfEncoding CDATA "CP1252"
       isPdfEmbedded (true | false) "false"
>
```

Report Font Name

The `name` attribute of a `<reportFont>` element is mandatory and must be unique, because it will be used when referencing the corresponding report font throughout the report.

Default Report Font

You can use `isDefault="true"` for one of your report font declarations, to mark the report font that you want to be used by the reporting engine as the default base font, when dealing with text elements that do not reference a particular report font. This default font will also be used by the text elements that do not have any font settings at all.

All the other report font properties are the same as those for a normal `<font>` element that we are going to see below.

XML Syntax

```
<!ELEMENT font EMPTY>

<!ATTLIST font
       reportFont NMTOKEN #IMPLIED
       fontName CDATA #IMPLIED
       size NMTOKEN #IMPLIED
       isBold (true | false) #IMPLIED
       isItalic (true | false) #IMPLIED
       isUnderline (true | false) #IMPLIED
       isStrikeThrough (true | false) #IMPLIED
       pdfFontName CDATA #IMPLIED
       pdfEncoding CDATA #IMPLIED
       isPdfEmbedded (true | false) #IMPLIED
>
```

Referencing a Report Font

When introducing the font settings for a text element of your report, you have the possibility to use a report font declaration as a base, for those font settings you want to obtain.
All the attributes of the <font> element, if present, are used only to override the attributes with the same name that are present in the report font declaration referenced using the `reportFont` attribute.

For example, if we have a report font like the following:

```
<reportFont
       name="Arial_Normal"
       isDefault="true"
       fontName="Arial"
       size="8"
       pdfFontName="Helvetica"
       pdfEncoding="Cp1252"
       isPdfEmbedded="false"/>
```

and we want to create a text field that has basically the same font settings like those in this report font, but only a greater size, the only thing we should do is to reference this report font using the `reportFont` attribute and specify the desired font size like this:

```
<textElement>
       <font reportFont="Arial_Normal" size="14"/>
</textElement>
```

When the `reportFont` attribute is missing, the default report font is used as base font.

Font Name

In Java, there are two types of fonts: physical fonts and logical fonts. Physical fonts are the actual font libraries consisting of, for example, TrueType or PostScript Type 1 fonts. The physical fonts may be Arial, Time, Helvetica, Courier, or any number of other fonts, including international fonts.

Logical fonts are the five font types that have been recognized by the Java platform since version 1.0: Serif, Sans-serif, Monospaced, Dialog, and DialogInput. These logical fonts are not actual font libraries that are installed anywhere on your system. They are merely font-type names recognized by the Java runtime, which must be mapped to some physical font that is installed on your system.

In the `fontName` attribute of the `<font>` element or the `<reportFont>` element, you have to specify the name of a physical font or the name of a logical font. You only have to make sure the font you specify really exists and is available on your system.

For more details about fonts in Java, check the Java Tutorial or the JDK documentation.

Font Size

The font size is measured in points and can be specified using the `size` attribute.

Font Styles and Decorations

There are 4 boolean attributes available in the `<font>` and `<reportFont>` elements that control the font style and/or decoration. Those are `isBold`, `isItalic`, `isUnderline` and `isStrikeThrough` and their significance should be evident to anybody.

PDF Font Name

When exporting reports to PDF format, the JasperReports library uses the iText library.

As their name states it (Portable Document Format) the PDF files can be viewed on various platforms and you can be sure they will always look the same. This is partially because in this format there is a special way of dealing with fonts.

If you want to design your reports so that they eventually be exported to PDF, you have to make sure you choose the appropriate PDF font settings that correspond to the Java font settings of your text elements.

The iText library knows how to deal with built-in fonts and TTF files. It recognizes the following built-in font names:

```
Courier
Courier-Bold
Courier-BoldOblique
Courier-Oblique
Helvetica
Helvetica-Bold
Helvetica-BoldOblique
Helvetica-Oblique
Symbol
Times-Roman
Times-Bold
Times-BoldItalic
Times-Italic
ZapfDingbats
```

The iText library requires us to specify either a built-in font name from the above list, either the name of a TTF file that it can locate on disk, every time we work with fonts. The font name introduced by the `fontName` attribute previously explained is of no use when exporting to PDF. This is why we have special font attributes, so that we are able to specify the font settings that the iText library expects from us.

The `pdfFontName` attribute can contain the name of a PDF built-in font from the above list or the name of a TTF file that can be located on disk at runtime, when exporting to PDF.

It is for the report design creator to choose the right value for the `pdfFontName` attribute that would perfectly corresponds to the Java physical or logical font specified using the `fontName` attribute. If those two fonts, one used by the Java viewers and printers and the other used in the PDF format, do not represent in fact the same font, or do not at least look alike, you might get unexpected results when exporting to PDF format.

Additional PDF fonts can be installed on your system if you choose one of the Acrobat Reader's font packs. For example, by installing the Asian font pack from Adobe on your system, you would be able to use for the `pdfFontName` attribute font names like:

Language	PDF Font Name
Simplified Chinese	`STSong-Light`
Traditional Chinese	`MHei-Medium`
	`MSung-Light`
Japanese	`HeiseiKakuGo-W5`
	`HeiseiMin-W3`
Korean	`HYGoThic-Medium`
	`HYSMyeongJo-Medium`

For more details about how to work with fonts when generating PDF documents, check the *iText library documentation*.

PDF Encoding

When creating reports in different languages and wanting to export them to PDF, you have to make sure that you choose the appropriate character encoding type.

For example, an encoding type widely used in Europe is `Cp1252`, also known as `LATIN1`. Other possible encoding types are:

Character Set	Encoding
Latin 2: Eastern Europe	`Cp1250`
Cyrillic	`Cp1251`
Greek	`Cp1253`
Turkish	`Cp1254`
Windows Baltic	`Cp1257`
Simplified Chinese	`UniGB-UCS2-H` `UniGB-UCS2-V`
Traditional Chinese	`UniCNS-UCS2-H` `UniCNS-UCS2-V`

Japanese	`UniJIS-UCS2-H`
	`UniJIS-UCS2-V`
	`UniJIS-UCS2-HW-H`
	`UniJIS-UCS2-HW-V`

Korean	`UniKS-UCS2-H` `UniKS-UCS2-V`

You can find more details about how to work with fonts and character encoding when generating PDF documents, here, in the *iText library documentation*.

PDF Embedded Fonts

If you want to use a TTF file when exporting your reports to PDF format and you want to make sure everybody will be able to view it without problem, you have to make sure that at least one of the following conditions are met:

- they all have that TTF font installed on their systems;
- you embed the font in the PDF document itself.

Its not easy to comply with the first condition and this is why the preferred way to do it is to embed the TTF in the generated PDF documents that you are distributing.

You can do that by setting the `isPdfEmbedded` attribute to `"true"`.

Further details about how to embed fonts in the PDF documents you can find in the *iText documentation*. A very useful example you can find in the *unicode* sample provided with the project.

8.1.2 Static Texts

Static texts are text elements with fixed content, which does not change during the report filling process. They are used mostly to introduce static text label in the generated documents.

XML Syntax

```
<!ELEMENT staticText (reportElement, textElement?, text?)>

<!ELEMENT text (#PCDATA)>
```

As you can see from the above presented syntax, besides element general properties and text specific properties that we have already explained, a static text definition has in addition only the `<text>` tag, which introduces the fixed text content of the static text element.

8.1.3 Text Fields

Unlike static text elements, which do not change their text content, text fields have an associated expression that is evaluated with every iteration in the data source, in order to obtain the text content that has to be displayed.

XML Syntax

```
<!ELEMENT textField (reportElement, textElement?, textFieldExpression?,
anchorNameExpression?, hyperlinkReferenceExpression?,
hyperlinkAnchorExpression?, hyperlinkPageExpression?)>

<!ATTLIST textField
       isStretchWithOverflow (true | false) "false"
       evaluationTime (Now | Report | Page | Column | Group) "Now"
       evaluationGroup CDATA #IMPLIED
       pattern CDATA #IMPLIED
       isBlankWhenNull (true | false) "false"
       hyperlinkType (None | Reference | LocalAnchor | LocalPage |
RemoteAnchor | RemotePage) "None"
>

<!ELEMENT textFieldExpression (#PCDATA)>

<!ATTLIST textFieldExpression
       class (java.lang.Boolean | java.lang.Byte | java.util.Date |
java.sql.Timestamp | java.lang.Double | java.lang.Float | java.lang.Integer
| java.lang.Long | java.lang.Short | java.math.BigDecimal |
java.lang.String) "java.lang.String"
>
```

Variable Height Text Fields

Given the fact that text fields have a dynamic content, most of the time you wont be able to exactly anticipate the amount of space you have to provide for your text fields so that they can display all their content.

If the space you reserve for your text fields is not sufficient, the text content will be truncated so that it fits in the available area.

This scenario is not always acceptable and you can let the reporting engine to calculate itself at runtime the amount of space required to display the entire content of the text field and automatically adjust the size of the report element.

You can achieve this by setting the `isStretchWithOverflow` to `"true"` for the particular text field elements you are interested in. By doing this, you make sure that if the specified height for the text field is not sufficient, it will automatically be increased (never decreased) in order to be able to display the entire text content.

When text fields are affected by this stretch mechanism, the entire report section to which they belong to will be also stretched.

Evaluating Text Fields

Normally, all the report expressions are evaluated immediately, using the current values of all the parameters, fields and variables at that particular moment. It is like making a photo of all data, for every iteration in the data source, during the report filling process.

This means that at any particular time, you won't have access to values that are going to be calculated later, in the report filling process. It perfectly makes sense, since all the variables are calculated step by step and reach their final value only when the iteration arrives at the end of the data source range they cover.
For example, a report variable that calculates the sum of a field for each page will not contain the expected sum until the end of the page is reached. That's because the sum is calculated step by step,

when iterating through the data source records, and at any particular time, the sum will be only partial, since not all the records of the specified range have been processed.
If this is the case, how to display the page sum of a this field, on the page header, since this value will be known only when the end of the page is reached. At the beginning of the page, when generating the page header, our sum variable would contain zero, or its initial value.

Fortunately, JasperReports has a very interesting feature that lets you decide the exact moment you want the text field expression to be evaluated, avoiding the default behavior which makes this expression be evaluated immediately, when generating the current report section.
It is the `evaluationTime` attribute we are talking about. It can have one of the following values:

- *Immediate evaluation*: The text field expression is evaluated when filling the current band (`evaluationTime="Now"`).
- *End of report evaluation*: The text field expression is evaluated when reaching the end of the report (`evaluationTime="Report"`).
- *End of page evaluation*: The text field expression is evaluated when reaching the end of the current page (`evaluationTime="Page"`).
- *End of column evaluation*: The text field expression is evaluated when reaching the end of the current column (`evaluationTime="Column"`).
- *End of group evaluation*: The text field expression is evaluated when the group specified by the `evaluationGroup` attribute changes (`evaluationTime="Group"`).

The default value for this attribute is `"Now"`, as already mentioned. In the example presented above, you could easily specify `evaluationTime="Page"` for the text field placed in the page header section, so that it displays the value of the sum variable only when reaching the end of the current page.

The only restriction you should be aware of, when deciding to avoid the immediate evaluation of the text field expression, is that in such cases, the text field will never stretch in order to acquire all its content.
This is because the text element height is calculated when the report section is generated and even the engine will come back later with the text content of the text field, the element height will not be adapted, because it will ruin the already created layout.

Suppressing Null Values Display

If the text field expression returns `null`, your text field will display the "null" text in the generated document. A simple way to avoid this is to set the `isBlankWhenNull` attribute to `"true"`. By doing this, the text field will cease to display "null" and will display an empty string. This way nothing will appear on your document if the text field value is `null`.

Formatting Output

Of course, when dealing with numeric or date/time values, you could use the Java API to format the output of the text field expressions yourself. But there is a more convenient way to do it: by using the pattern attribute available in the `<textField>` element.

The value you should supply to this attribute is the same that you would supply if it were for you to format the value using either the `java.text.DecimalFormat` class or `java.text.SimpleDateFormat` class, depending on the type of value to format.

In fact, what the engine does is to instantiate the `java.text.DecimalFormat` class if the text field expression returns subclasses of the `java.lang.Number` class or to instantiate the `java.text.SimpleDataFormat` if the text field expression return `java.util.Date` or `java.sql.Timestamp` objects.

For more detail about the syntax of this `pattern` attribute, check the Java API documentation for those two classes: `java.text.DecimalFormat` and `java.text.SimpleDateFormat`.

Text Field Expression

We have already talked about the text field expression. There is nothing more to say about it except that it is introduced by the `<textFieldExpression>` element and can return values from only a limited range of classes listed below:

```
java.lang.Boolean
java.lang.Byte
java.util.Date
java.sql.Timestamp
java.lang.Double
java.lang.Float
java.lang.Integer
java.lang.Long
java.lang.Short
java.math.BigDecimal
java.lang.String
```

If the text field expression class is not specified using the `class` attribute, it is assumed to be `java.lang.String`, by default.

8.2 Graphic Elements

The second major category of report elements, besides text elements that we have seen, are the graphic elements. In this category we have lines, rectangles and images.

They all have some properties in common and those are grouped under the attributes of the `<graphicElement>` tag.

XML Syntax

```
<!ELEMENT graphicElement EMPTY>

<!ATTLIST graphicElement
       stretchType (NoStretch | RelativeToTallestObject |
RelativeToBandHeight) "NoStretch"
       pen (None | Thin | 1Point | 2Point | 4Point | Dotted) #IMPLIED
       fill (Solid) "Solid"
>
```

Stretch Behavior

The `stretchType` attribute of a graphic element can be used to customize the stretch behavior of the element when on the same report section there are text fields that stretch themselves because their text content is too large to fit in the original text field height.
When stretchable text fields are present on a report section, the height of the report section itself will be affected be stretch.

A graphic element can respond to the modification of the report section layout in three ways:

- *Won't stretch*: The graphic element preserves its original specified height (`strechType="NoStretch"`).
- *Stretching relative to the parent band height*: The graphic element will adapt its height to match the new height of the report section it placed on, which has been affected by stretch (`stretchType="RelativeToBandHeight"`).
- *Stretching relative to the tallest element in group*: You have the possibility to group the elements of a report section in multiple imbricate groups, if you like. The only reason you might have for grouping your report elements is to be able to customize their stretch behavior. Details about how to group elements are supplied in the section *8.4 Element Groups* that will follow. Graphic

elements can be made to automatically adapt their height to fit the amount of stretch suffered by the tallest element in the group that they are part of (`stretchType="RelativeToTallestObject"`).

Border Thickness

Unlike text elements, the graphic elements always have a border. You can control the type and thickness of it using the `pen` attribute. Remember that the color of the border comes from the `forecolor` attribute presented when describing the `<reportElement>` tag, in a previous chapter.

Here are all the possible types for a graphic element border:

- *No border*: The graphic element will not display any border around it (`pen="None"`).
- *Thin border*: The border around the graphic element will be half a point thick (`pen="Thin"`).
- *1 point thick border*: Normal border (`pen="1Point"`).
- *2 points thick border*: Thick border (`pen="2Point"`).
- *4 point thick border*: (`pen="4Point"`).
- *Dotted border*: The border will be 1 point thick and made of dots (`pen="Dotted"`).

The default border around graphic elements depends on their type. Lines and rectangles have a normal 1 point thick border by default. Images however, do not display any border, by default.

Background Fill Style

The `fill` attribute specifies the style of the background for the graphic elements, but the only style supported for the moment is the solid fill style, which is also the default (`fill="Solid"`).

8.2.1 Lines

When displaying a line element, JasperReports draws one of the two diagonals of the rectangle represented by the `x`, `y`, `width` and `height` attributes specified for this particular line element.

XML Syntax

```
<!ELEMENT line (reportElement, graphicElement?)>

<!ATTLIST line
      direction (TopDown | BottomUp) "TopDown"
>
```

Line Direction

Which one of the two diagonals of the rectangle should be drawn can be decided using the direction attribute:

- The diagonal that starts in the top-left corner of the rectangle and goes to the bottom left corner is drawn in case you set `direction="TopDown"`.
- The line will start in the bottom-left corner and will go to the upper-right if you choose `direction="BottomUp"`.

You can draw vertical lines by specifying `width="0"` and horizontal lines setting `height="0"`. For such lines the direction is not important.

The default direction for a line is `"TopDown"`.

8.2.2 Rectangles

The rectangles are the most basic graphic elements. This is why there are no supplementary settings to make for the declaration of a rectangle element, besides those already seen when talking about the `<reportElement>` and `<graphicElement>` tags.

XML Syntax

```
<!ELEMENT rectangle (reportElement, graphicElement?)>
```

For more detailed examples of lines and rectangles, check the *shapes* sample.

8.2.3 Images

The most complex graphic elements that you can have on a report are the images.

Just like for the text field elements, their content is dynamically evaluated at runtime, using a report expression.

XML Syntax

```
<!ELEMENT image (reportElement, graphicElement?, imageExpression?,
anchorNameExpression?, hyperlinkReferenceExpression?,
hyperlinkAnchorExpression?, hyperlinkPageExpression?)>

<!ATTLIST image
       scaleImage (Clip | FillFrame | RetainShape) "RetainShape"
       isUsingCache (true | false) "true"
       evaluationTime (Now | Report | Page | Column | Group) "Now"
       evaluationGroup CDATA #IMPLIED
       hyperlinkType (None | Reference | LocalAnchor | LocalPage |
RemoteAnchor | RemotePage) "None"
>

<!ELEMENT imageExpression (#PCDATA)>

<!ATTLIST imageExpression
       class (java.lang.String | java.io.File | java.net.URL |
java.io.InputStream | java.awt.Image) "java.lang.String"
>
```

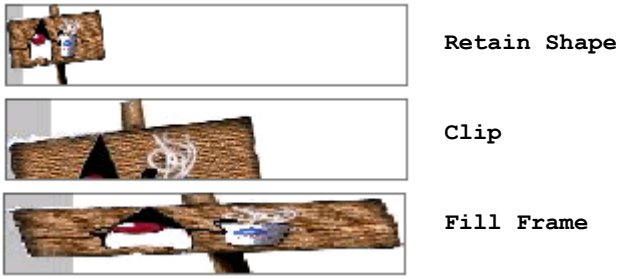
Scaling Images

Given the fact that images are loaded at runtime, there is no way knowing their exact size, when creating the report design. It might be that the dimensions of the image element specified at design time do not correspond to the actual image loaded at runtime.

This is why you have to decide how you expect the image to behave in order to adapt to the original image element dimensions you specified in the report design. There is the `scaleImage` attribute that allows you to do that, by choosing one of its 3 possible values:

- *Clipping the image*: If the actual image is larger than the image element size, it will be cut off so that it keeps its original resolution, and only the region that fits the specified size will be displayed (`scaleImage="Clip"`).

- *Forcing the image size*: If the dimensions of the actual image do not fit those specified for the image element that displays it, the image can be forced to obey them and stretch itself so that it fits in the designated output area. It will be deformed if necessary (`scaleImage="FillFrame"`).
- *Keeping image proportions*: If the actual image does not fit into the image element, it can be adapted to those dimensions without needing to deform it and keep its original proportions (`scaleImage="RetainShape"`).



- figure 7 -

Caching Images

All image elements have dynamic content. There are no special elements to introduce static images on the reports, like we have special static text elements.
However, most of the time, the images on a report are in fact static and do not necessarily come from the data source or from parameters. In the majority of cases, they are loaded from files on disk and represent logos and other static resources.

If we have to display the same image multiple times on a report, if it is about a logo appearing on the page header for example, there is no point on loading the image file every time we have to display it.
We can instruct the reporting engine to cache this particular image. This way we are making sure that the image will be loaded from disk or from its particular location only once and then it will only be reused every time it has to be displayed.

By setting the `isUsingCache` attribute to `"true"`, the reporting engine will try to recognize previously loaded images using their specified source. For example, it will recognize an image if the image source is a file name that it has already loaded, or if it is the same URL.

This caching functionality is available only for image elements that have expressions returning `java.lang.String` objects as the image source, representing file names, URLs or classpath resources. That's because the engine uses the image source string as the key to recognize that it is the same image that it has already cached.

Evaluating Images

As we have already seen when talking about text fields, you have the possibility to postpone the evaluation of the image expression, which by default is performed immediately.
This would allow you to display in some region of a document, images that are going to be built or chose only later in the report filling process, due to complex algorithms or whatever.

The same attributes, `evaluationTime` and `evaluationGroup`, that we have talked about in the text fields section are available also in the `<image>` element. The `evaluationTime` attribute can have the following values:

- *Immediate evaluation*: The image expression is evaluated when filling the current band (`evaluationTime="Now"`).
- *End of report evaluation*: The image expression is evaluated when reaching the end of the report (`evaluationTime="Report"`).
- *End of page evaluation*: The image expression is evaluated when reaching the end of the current page (`evaluationTime="Page"`).

- *End of column evaluation*: The image expression is evaluated when reaching the end of the current column (`evaluationTime="Column"`).
- *End of group evaluation*: The image expression is evaluated when the group specified by the `evaluationGroup` attribute changes (`evaluationTime="Group"`).

The default value for this attribute is `"Now"`.

Image Expression

The value returned by the image expression is used as the source for the image that is going to be displayed. The image expression is introduced by the `<imageExpression>` element and can return values from only a limited range of classes listed below:

```
java.lang.String
java.io.File
java.net.URL
java.io.InputStream
java.awt.Image
```

When the image expression returns a `java.lang.String` value, the engine will try to see whether the value represents an URL from which to load the image. If it is not a valid URL representation, it will try to locate a file on disk and load the image from it, assuming that the value represents a file name. If no file is found, it will finally assume that the string value represents the location of a classpath resource and will try to load the image from there. Only if all those fail, an exception will be thrown.

If the image expression class is not specified using the `class` attribute, it is assumed to be `java.lang.String`, by default.

The *images* sample provided with the project contains several examples of image elements.

8.2.4 Charts and Graphics

The JasperReports library does not produce charts and graphics itself. This is not one of its goals.
However, it can easily integrate charts and graphics produces by other, more specialized Java libraries.

The great majority of available Java libraries that produce charts and graphics can output to image files or to in-memory Java image objects. This is why it shouldn't be hard for anybody to put a chart or a graphic generated by one of those libraries into a JasperReports document using a normal image element that we have presented in the previous section of this book.

You can see this working in the sample called *chart*, which comes with the project.

8.3 Hyperlinks

JasperReports allows you to create drill-down reports, to introduce tables of contents in your documents or to redirect viewers to other external documents using special report elements called hyperlinks.

Hyperlinks are special elements that contain a reference to a local destination within the current document or to an external resource to which the viewer of the document will be redirected if he or she clicks on that particular hyperlink element.
Hyperlinks are not the only actors in this viewer-redirecting scenario. There has to be a way for you to specify what are the destinations in a document. These local destinations are called anchors.

There are no special report elements that introduce hyperlinks or anchors in a report design, but rather some special setting that make an usual report element to be a hyperlink or/and an anchor.

In JasperReports, only text field elements and image elements can be hyperlinks or anchors. This is because for both types of elements, there are special settings that allow you to specify the hyperlink reference to which the hyperlink will point or the name of the local anchor. Note that a particular text field or image can be both anchor and hyperlink at the same time.

XML Syntax

```
<!ELEMENT anchorNameExpression (#PCDATA)>

<!ELEMENT hyperlinkReferenceExpression (#PCDATA)>

<!ELEMENT hyperlinkAnchorExpression (#PCDATA)>

<!ELEMENT hyperlinkPageExpression (#PCDATA)>
```

Hyperlink Type

When presenting the XML syntax for text field elements and image elements, you probably saw that there was an attribute called `hyperlinkType`, which we didn't explain at that moment. We are going to do that right now and here are the possible values for this attribute along with their significance:

- *No hyperlink*: By default, neither the text fields nor the images represent hyperlinks, even if the special hyperlink expressions are present (`hyperlinkType="None"`).
- *External reference*: The current hyperlink points to an external resource specified by the corresponding `<hyperlinkReferenceExpression>` element, usually an URL (`hyperlinkType="Reference"`).
- *Local anchor*: The current hyperlink points to a local anchor specified by the corresponding `<hyperlinkAnchorExpression>` element (`hyperlinkType="LocalAnchor"`).
- *Local page*: The current hyperlink points to a 1 based page index within the current document specified by the corresponding `<hyperlinkPageExpression>` element (`hyperlinkType="LocalPage"`).
- *Remote anchor*: The current hyperlink points to an anchor specified by the `<hyperlinkAnchorExpression>` element, within an external document indicated by the corresponding `<hyperlinkReferenceExpression>` element (`hyperlinkType="RemoteAnchor"`).
- *Remote page*: The current hyperlink points to a 1 based page index specified by the `<hyperlinkPageExpression>` element, within an external document indicated by the corresponding `<hyperlinkReferenceExpression>` element (`hyperlinkType="RemotePage"`).

Anchor Expression

If present in a text field or image element declaration, the `<anchorNameExpression>` tag will transform that particular text field or image into a local anchor of the resulting document, to which hyperlinks can point. The anchor will bare the name returned after evaluating the anchor name expression, which should always return `java.lang.String` values.

Hyperlink Expressions

Depending on the current hyperlink type, one or two of the following expressions will be evaluated and used to build the reference to which the hyperlink element will point:

```
<hyperlinkReferenceExpression>
<hyperlinkAnchorExpression>
<hyperlinkPageExpression>
```

What is important to know is that the first two should always return `java.lang.String` and the third should return `java.lang.Integer` values.

There is a special sample called *hyperlink*, provided with the projects, which shows how this type of report elements can be used.

8.4 Element Groups

Report elements placed in any report section can be arranged in multiple imbricate groups. The only reason you might have for grouping your elements is to be able to customize the stretch behavior of the graphic elements, as explained in the section *8.2 Graphic Elements*.

The `stretchType` attribute, available for graphic elements, has among its possible values one called "`RelativeToTallestObject`". When choosing this option, the engine will try to identify the object from the same group with the current graphic element, which suffered the biggest amount of stretch. It will then adapt the height of the current graphic element to the height of this tallest element of the group.

But for this to work, you have to group your elements. This is done using the `<elementGroup>` and `</elementGroup>` tags to mark the elements that are part of the same group.

XML Syntax

```
<!ELEMENT elementGroup (line | rectangle | image | staticText | textField |
subreport | elementGroup)*>
```

Element groups can contain other nested element groups and there is no limit on the number of the nested element groups.

Check the *stretch* sample, to see how element grouping works.

9 Subreports

Subreports are an import feature for a report-generating tool. They allow the creation of more complex reports and simplify the design work. The subreports are very useful when creating master-detail type of reports or when the structure of a single report is not sufficient to describe the complexity of the desired output document.

A subreport is in fact a normal report that is been incorporated as apart of another report. You can imbricate your subreports and make a subreport that contains itself other subreports, the nesting level not being limited.

On the other hand, a subreport is also a special kind of a report element that helps you introduce a subreport into the parent report.

There's nothing more to say about subreports, seen as normal reports, because they are compiled and filled just like normal reports are and we already have seen all that in previous chapters. In fact, any report design can be used as a subreport when incorporated into another report design, without the need to change anything inside it.

What we are going to see now, are the details concerning the `<subreport>` element that you use when introducing subreports into master reports.

XML Syntax

```
<!ELEMENT subreport (reportElement, parametersMapExpression?,
subreportParameter*, (connectionExpression | dataSourceExpression)?,
subreportExpression?)>

<!ATTLIST subreport
      isUsingCache (true | false) "true"
>

<!ELEMENT parametersMapExpression (#PCDATA)>

<!ELEMENT subreportParameter (subreportParameterExpression?)>

<!ATTLIST subreportParameter
      name NMTOKEN #REQUIRED
>

<!ELEMENT subreportParameterExpression (#PCDATA)>

<!ELEMENT connectionExpression (#PCDATA)>

<!ELEMENT dataSourceExpression (#PCDATA)>

<!ELEMENT subreportExpression (#PCDATA)>

<!ATTLIST subreportExpression
      class (java.lang.String | java.io.File | java.net.URL |
java.io.InputStream | dori.jasper.engine.JasperReport) "java.lang.String"
>
```

Subreport Expression

Just like normal report designs, subreport designs are in fact `dori.jasper.engine.JasperReport` objects. Those are obtained after compiling a `dori.jasper.engine.design.JasperDesign` object as seen in the *3.2 Compiling Report Designs* section of this book.

We have seen that the text field elements have an expression that will be evaluated to obtain the text content to display. The image elements have an expression representing the source of the image to display. In the same way, subreport elements have an expression that is evaluated at runtime, in order to obtain the source of the `dori.jasper.engine.JasperReport` object to load.

The so-called subreport expression is introduced by the `<subreportExpression>` element and can return values from the following classes:

```
java.lang.String
java.io.File
java.net.URL
java.io.InputStream
dori.jasper.engine.JasperReport
```

When the subreport expression returns a `java.lang.String` value, the engine will try to see whether the value represents an URL from which to load the subreport design object. If it is not a valid URL representation, it will try to locate a file on disk and load the subreport design from it, assuming that the value represents a file name. If no file is found, it will finally assume that the string value represents the location of a classpath resource and will try to load the subreport design from there. Only if all those fail, an exception will be thrown.

If the image expression class is not specified using the `class` attribute, it is assumed to be `java.lang.String`, by default.

Caching Subreports

A subreport element can load different subreport designs with every evaluation, giving you great flexibility in shaping you documents.
However, most of the time, the subreport elements on a report are in fact static and their source do not necessarily change with every new evaluation of the subreport expression. In the majority of cases, the subreport designs are loaded from fixed locations: files on disk or static URLs.

If the same subreport design is filled multiple times on a report, there is no point on loading the subreport design object from the source file every time we have to fill it with data.
We can instruct the reporting engine to cache this particular subreport design object. This way we are making sure that the subreport design will be loaded from disk or from its particular location only once and then it will only be reused every time it has to be filled.

By setting the `isUsingCache` attribute to `"true"`, the reporting engine will try to recognize previously loaded subreport design objects, using their specified source. For example, it will recognize a subreport object if its source is a file name that it has already loaded, or if it is the same URL.

This caching functionality is available only for subreport elements that have expressions returning `java.lang.String` objects as the subreport design source, representing file names, URLs or classpath resources. That's because the engine uses the subreport source string as the key to recognize that it is the same subreport design that it has already cached.

9.1 Subreport Parameters

Since subreports are normal reports themselves, they are compiled or filled in the same way. This means that they also require a data source from which to get the data when they are filled and that can also receive parameters, for the additional information they have to use when being filled.

There are two ways to supply the parameter values to a subreport and they can be used simultaneously, if desired.

You can supply a map containing the parameter values, like we do when filling a normal report with data, using one of the `fillReportXXX()` methods exposed by the `dori.jasper.engine.JasperFillManager` class (see the *3.4 Filling Reports* chapter to refresh your memory).
This can be achieved if you use the `<parametersMapExpression>` element, which introduces the expression that will be evaluated in order to obtain the specified parameter map. This expression should always return a `java.util.Map` object in which the keys are the parameter names.

In addition to or instead of supplying the parameter values in a map, you can supply the parameter values individually, one by one, using a `<subreportParameter>` element for each parameter you are interested in. In such case, you have to specify the name of the corresponding parameter using the mandatory `name` attribute and you have to provide an expression that will be evaluated at runtime to obtain the value for that particular parameter, value that will be supplied to the subreport filling routines.

Note that you can use both ways to provide subreport parameter values, simultaneously. When this happens, the parameter values specified individually, using the <subreportParameter> element, will override the parameters values present in the parameter map, that correspond to the same subreport parameter. If the map does not contain corresponding parameter values already, the individually specified parameter values will be added to the map.

Attention! When you supply the subreport parameter values, you have to be aware that the reporting engine will affect the `java.util.Map` object it receives, adding the built-in report parameter values that correspond to the subreport. This map is also affected by the individually specified subreport parameter values, as already explained above.
In order to avoid altering the original `java.util.Map` object that you send, you can wrap it in a different map, before supplying it to the subreport filling process like this:

```
new HashMap(myOriginalMap)
```

This way, your original map object remains unaffected and modifications are made to the wrapping map object.
This is useful especially when you want to supply to your subreport the same set of parameters that the master report has received and you use the built-in `REPORT_PARAMETERS_MAP` report parameter of the master report. However, you don't want to affect the value of this built-in parameter and you will wrap it like this:

```
<parametersMapExpression>
    new HashMap($P{REPORT_PARAMETERS_MAP})
</parametersMapExpression>
```

9.2 Subreport Data Source

Subreports need a data source in order to generate their content, just like normal report do.
In the *3.4 Filling Reports* chapter of this book we have seen that when filling a report you have to supply either a data source object or a connection object, depending on that particular report type. That is if it has an internal SQL query and you want to have it executed to obtain the report data or you supply the report data yourself.

Subreports behave in the same way and expect to receive the same kind of input when they are being filled.

You can supply to your subreport either a data source using the `<dataSourceExpression>` element or a JDBC connection for the engine to execute the subreport's internal SQL query using the `<connectionExpression>` element. These two XML elements cannot be both present at the same time in a subreport element declaration. This is because you cannot supply both a data source and a connection for your subreport. You have to decide on one of them and stick to it.

The report engine expects that the data source expression returns a `dori.jasper.engine.JRDataSource` object and that the connection expression returns a `java.sql.Connnection` object, whichever is present.

You can see how subreports work, if you check the *subreport* sample.

10 Advanced JasperReports

Previous chapters have presented the core functionality that most people will get to use when working with the JasperReports library.

However, some complex requirements of your specific applications might force you to dig deeper into the JasperReports functionality in order to adapt it to suit you needs.

In the following sections we are going to take a closer look at those aspects that are likely to interest you if you'll want to make full benefit from the use of the JasperReports library.

10.1 XML Report Designs Loading and Writing

In the *3.2 Compiling Report Designs* chapter we have explained how report designs pass from their initial XML form into the compiled form, before being used to generate full-featured documents.

The engine first parses the XML report design and creates the in-memory representation of it by instantiating and preparing a `dori.jasper.engine.design.JasperDesign` object. This object is then subject to various validation checks and suffers the compilation process that produces a corresponding `dori.jasper.engine.JasperReport` object.

But in certain cases, in your application, you might want to manually load the XML report design into a `dori.jasper.engine.design.JasperDesign` object, without immediately compiling it. Such scenarios might be common for applications that programmatically create report designs and use the XML form to store them temporary or permanently.

Loading `dori.jasper.engine.design.JasperDesign` objects from XML report design can be easily by calling one of the public static `load()` methods exposed by the `dori.jasper.engine.xml.JRXmlLoader` class. This way you can load report design object from XML content store in filed or that is been read from input streams.

The process opposed to the XML report design loading process is the generation of the XML form for a given report design object.

As seen above, sometimes report designs are created programmatically, using the JasperReports API. The report design objects obtained this way can be serialized, for disk storage or transfer over the network, but they also can be stored in XML format.

You can obtain the XML representation of a given report design object by using one of the public static `writeReport()` methods exposed by the `dori.jasper.engine.xml.JRXmlWriter` utility class.

10.2 Implementing Data Sources

JasperReports library comes with several default implementations of the `dori.jasper.engine.JRDataSource` interface. This interface is used to supply the report data when invoking the report filling process, as explained in the previous chapters of this book.
These default implementations let you generate reports using data from relational databases retrieved through JDBC, from Java Swing tables or from collections and arrays of JavaBeans objects.

But maybe your application data that your are trying to display in your reports has a special structure or is organized in a very particular way preventing you from using any of the default implementations of the data source interface that come with the library.
In such situations, you will have to create custom implementations for the `dori.jasper.engine.JRDataSource` interface, in order to wrap your special report data, so that the reporting engine can understand and use it when generating the reports for you.

Creating a custom implementation for the `dori.jasper.engine.JRDataSource` interface is not very difficult since you have to implement only two methods.
The first one, the `next()` method, is called by the reporting engine every time it wants the current pointer to advance to the next virtual record in the data source.
The other, the `getFieldValue()` method, is called by the reporting engine with every iteration in the data source to retrieve the value for each report field.

10.3 Customizing Viewers

The JasperReports library comes with built-in viewers that allow you to display the reports stored in the library's proprietary format or to preview your report designs when you create them.
These viewers are represented by the following two classes:

- `dori.jasper.view.JasperViewer` : You use this class to view generated reports, either as in-memory objects or serialized objects on disk or even stored in XML format.
- `dori.jasper.view.JasperDesignViewer` : This class can be used to preview report designs, either in XML form or compiled form.

But these default viewers might not suit everybody's needs and therefore you might consider customizing them so they adapt to certain application requirements.
In order to do that, you should be aware that these viewers use in fact other, more basic visual components that come with the JasperReports library.
The report viewer mentioned above use the visual component represented by the `dori.jasper.view.JRViewer` class and its companions. It is in fact a special `javax.swing.JPanel` component that is capable of displaying generated reports and that can be easily incorporated in other Java Swing based applications or applets.
If the functionality of this basic visual component is not sufficient for what you need, you can adapt it by subclassing it. If for example you would want to have an extra button on the toolbar of this viewer, you might consider extending the component and add that button yourself in the new visual component you obtain by subclassing.

This can be seen in the *webapp* sample, where the "Printer Applet" displays a customized version of the report viewer with an extra button in the toolbar.

Another very important issue is that the default report viewer that comes with the library does not know how to deal with document hyperlinks that point to external resources. It only deals with local references and can redirect the viewer to corresponding local anchor.
However, JasperReports offers you the possibility to handle yourself the clicks made on document hyperlinks that point to external documents and not local anchors.

The only thing you have to do in order to achieve this, is to implement the `dori.jasper.view.JRHyperlinkListener` interface and to add to register with the viewer component an instance of this listener class, using the `addHyperlinkListener()` method exposed by the `dori.jasper.view.JRViewer` class.
By doing this, you make sure the viewer will also call your implementation of the `gotoHyperlink()` method in which you handle yourself the external references.

10.4 Exporting to New Output Formats

The JasperReports library continually evolves and improves. Among the features that are likely to be introduced with time is the ability to export to new documents formats, besides PDF, HTML and XML.

In order to extend diversify in this direction, without affecting the existing functionality, JasperReports provides those interested in this subject an interface for them to implement, in case they want to create exporter classes that transform the generated documents into new output formats.
This way, if you need to export your reports into a special output format that is not yet available in the core library, you might decide to implement yourself the `dori.jasper.engine.JRExporter` interface.

Before deciding to implement this interface, it is important for you to understand how the implementation is expected to function.

All the input data the exporter might need should be supplied to it using the so-called exporter parameters, before the exporting process is started.
This is because the exporting process will be always invoked by calling the `exportReport()` method of the `dori.jasper.engine.JRExporter` interface, and this method does not receive any parameters itself, when called. The exporter parameters have to be already set using the `setParameter()` method on the exporter instance you are working with, before launching the export task.
You might choose to bulk set all your exporter parameters using the `setParameters()` method which receives a `java.util.Map` object containing the parameter values. The keys in this map should be instances of the `dori.jasper.engine.JRExporterParameter` class, as you would supply when individually calling the `setParameter()` method for each of your exporter parameters.

Note that no matter what the type of output your exporter produces, you will be using parameters to indicate to the exporter where to place or send this output.
Such parameter might be called OUT parameters.
For example, if you want your exporter to send the output it produces to an output stream, you will supply the `java.io.OutputStream` object reference to the exporter using a parameter, probably identified by the `dori.jasper.engine.JRExporterParameter.OUTPUT_STREAM` constant.
It is recommended to use the public constants of the `dori.jasper.engine.JRExporterParameter` class to identify the parameters you set in your exporters and only if you don't find one available for a particular setting you have to make in your exporter, to extend this class to add new constants. This can be seen for the `dori.jasper.engine.export.JRXmlExporter`, where special parameter identifier where created by subclassing the `dori.jasper.engine.JRExporterParameter` class in the `dori.jasper.engine.export.JRXmlExporterParameter` class.

You don't have to start from scratch when implementing the exporter interface, because there is a convenience abstract class called `dori.jasper.engine.JRAbstractExporter` that at least deals with parameter management for you.