
# iREPORT ULTIMATE GUIDE

http://www.jaspersoft.com

This is version 0509-UGI35-1 of the *iReport Ultimate Guide*.

# TABLE OF CONTENTS

# 1 INTRODUCTION

iReport is an open source authoring tool that can create complex reports from any kind of Java application through the JasperReports library. It is written in 100% pure Java and is distributed with source code according to the GNU General Public License.

Through an intuitive and rich graphic interface, iReport lets you rapidly create any kind of report very easily. iReport enables engineers who are just learning this technology to access all the functions of JasperReports as well as helping skilled users to save a lot of time during the development of very elaborate reports.

With Version 3.1, iReport was almost completely rewritten, with the new application based on the NetBeans rich client platform. Even though the user interface appears pretty much the same, a complete new design of the iReport core and the use of the NetBeans platform will allow us to quickly create amazing new features, making iReport even more easy to use and learn.

## 1.1 Features of iReport

The following list describes some of the most important features of iReport:

- 100% support of JasperReports XML tags
- WYSIWYG editor for the creation of reports. It has complete tools for drawing rectangles, lines, ellipses, text fields, labels, charts, sub-reports and crosstabs
- Built-in editor with syntax highlighting for writing expressions
- Support for Unicode and non-Latin languages (Russian, Chinese, Japanese, Korean, etc.)
- Browser for document structure
- Integrated report compiler, filler, and exporter
- Support for all databases accessible by JDBC
- Virtual support for all kinds of data sources
- Wizard for creating reports and sub-report automatically
- Support for document templates.
- TrueType fonts support
- Support for localization
- Extensibility through plug-ins
- Support for charts
- Management of a library of standard objects (e.g., numbers of pages)
- Drag-and-drop functionality
- Unlimited undo/redo

- Wizard for creating crosstabs
- Styles library
- Integrated preview
- Error manager
- JasperServer repository explorer
- Integrated SQL and MDX query designer

## 1.2 The iReport Community

The iReport team comprises many skilled and experienced programmers who come from every part of the world. They work daily to add new functionality and fix bugs. The iReport web site is at http://ireport.sourceforge.net. If you need help with iReport, there is a discussion forum in English. This is the place where you can send requests for help and technical questions about the use of the program, as well as post comments and discuss implementation choices or propose new functionality. There is no guarantee for a prompt reply, but requests are usually satisfied within a few days' time. This service is free. If you need information concerning commercial support, you can write to sales@jaspersoft.com.

Please report bugs at the following address:

http://jasperforge.org/tracker/?group_id=83

At the project site, there is a system to send requests for enhancement (RFE). There is also the ability to suggest patches and integrative code. All members of the iReport team value feedback from the user community and seriously consider all suggestions, criticism and advice coming from iReport users.

## 1.3 Code Used in this Book

JasperReports supports the following languages for expressions:

- Java
- JavaScript
- Groovy

All the sample expressions used in this guide are written in JavaScript.

# 2 GETTING STARTED

In this chapter you will learn the basic requirements to use iReport, where you can get it and how to install it.

## 2.1 Platform Requirements

iReport needs the Sun Java 2 SDK to run, Version 1.5 or newer. If you want to build the tool from the source code, or write a plug-in, you will also need NetBeans IDE and the NetBeans platform 6.0.1.

As for hardware, like all Java programs, iReport consumes a lot of RAM, so it is necessary to have at least 256MB of memory available as well as about 50MB of free disk space.

## 2.2 Download

You can download iReport from the dedicated project page on SourceForge where you can always find the last released iReport distribution (http://sourceforge.net/projects/ireport). Four different distributions are available:

- `iReport-nb-x.x.x.zip`: This is the official binary distribution in ZIP format.
- `iReport-nb-x.x.x.tgz`: This is the official binary distribution in TAR GZ format.
- `iReport-nb-x-x-x-src.zip`: This is the official distribution of sources in ZIP format.
- `iReport-nb-x.x.x-windows-installer.exe`: This is the official Win32 installer.

`nb-`*`x.x.x`* represents the version number of iReport. Every distribution contains all needed libraries from third parties necessary to use the program and additional files, such as templates and base documentation in HTML format. (The string "`nb`" identifies the NetBeans version and is used to differentiate it from previous versions of iReport.

iReport is also available as native plug-in for NetBeans IDE 6.x. You can download the plug-in from SourceForge or from the NetBeans plug-in portal:

http://www.netbeans.org/pluginPortal

At the time of writing we are planning a OS X distribution, to support Macintosh systems, that may be available in the future.

## 2.3 Development Versions

If you want to test pre-release versions of iReport, you can directly access the developmental source repository with SVN. In this case, you must have an SVN client (my favorite is Tortoise SVN). If you don't have one, you will need to create an account on `http://www.jasperforge.org/` in order to access the repository.

Pre-release iReport source code is no longer available on SourceForge CVS Server.

The URL of the SVN repository for iReport is:

https://jasperforge.org/svn/repos/ireportfornetbeans

## 2.4 Compiling iReport

The distribution with sources contains a NetBeans project. In order to compile the project and run iReport you need NetBeans IDE and the platform 6.0.1. If you are using NetBeans 6.0, the platform is the same as the IDE, otherwise you'll need to download the platform separately at this URL:

http://download.netbeans.org/netbeans/6.0/final/zip/netbeans-6.0.1-200801291616-mml.zip

Download `iReport-nb-x.x.x-src.zip`, unzip it into the directory of your choice, for example, `c:\devel` (or `/usr/devel` on a Unix system).

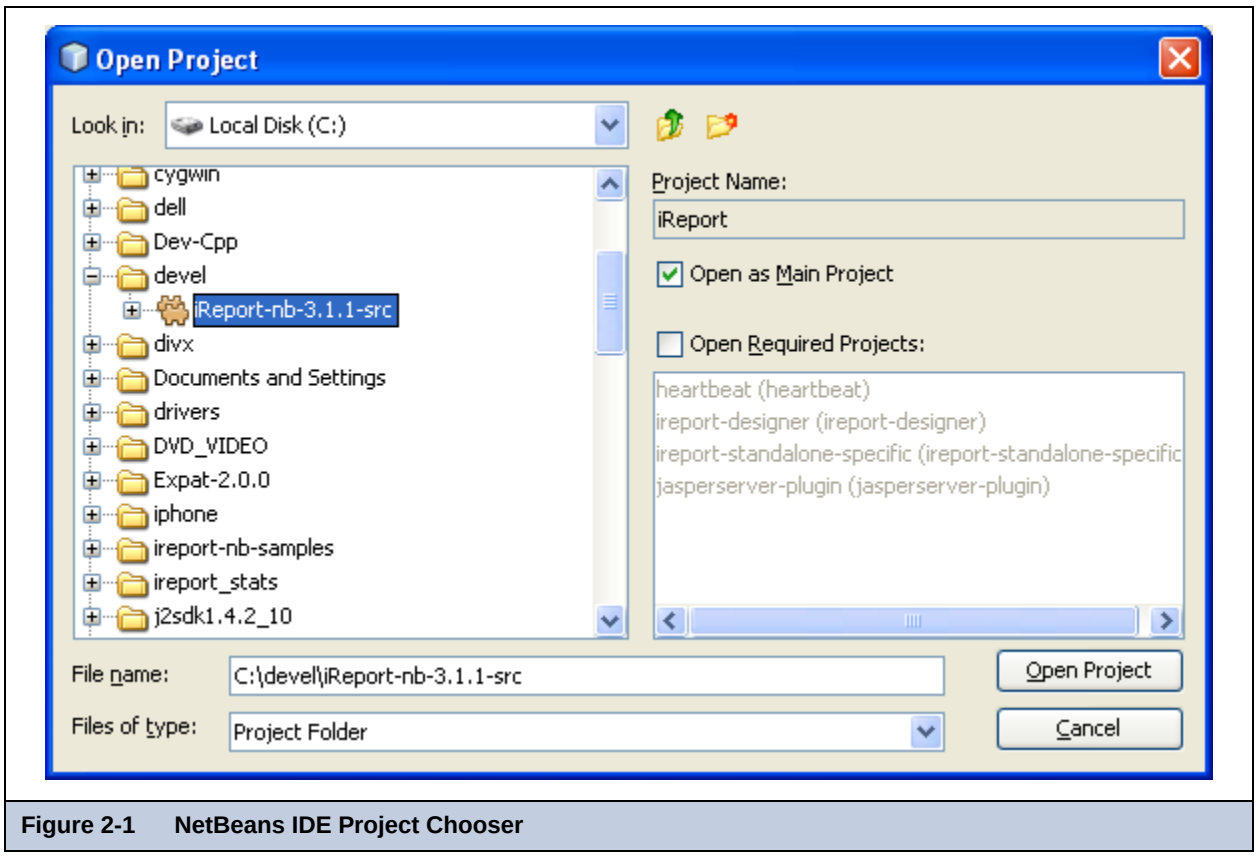



**Figure 2-1 NetBeans IDE Project Chooser**

Run NetBeans IDE and open the iReport project (see Figure 2-1).

The project is actually a suite that contains several sub-projects or *modules*. To run iReport just click the “Run main project” button on the tool bar.

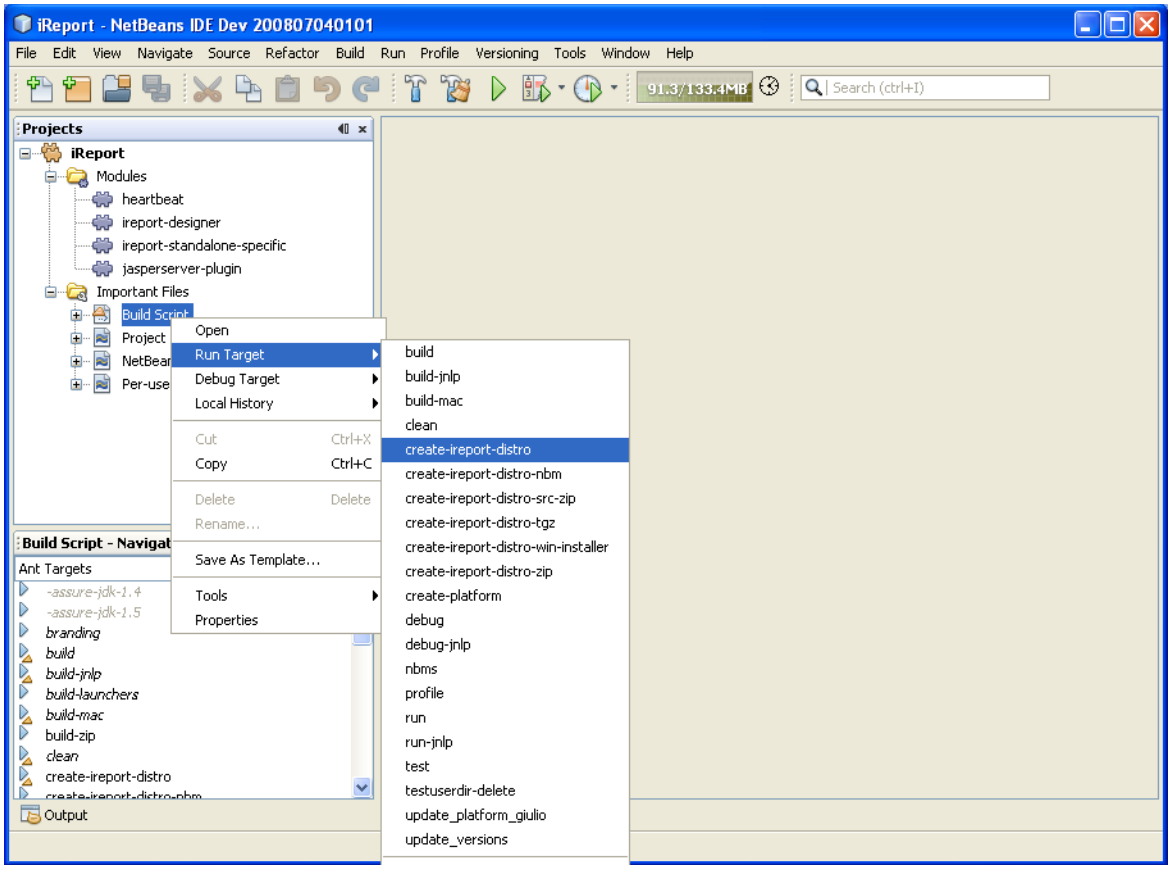



**Figure 2-2 The build script contains a set of targets to create several iReport distributions**

If you want to build all the distributions, run the `create-ireport-distro` target provided in the build script. To do it, select the `build.xml` (Build Script) file located in the special project folder “Important Files”, right click the file and select the appropriate target to run (see **Figure 2-2**).

## 2.5 Installing iReport

If you download the binary version of iReport:

Unpack the distribution archive into the directory of your choice, for example, `c:\devel` (or `/usr/devel` on a UNIX system).

1. Open a command prompt or a shell, go to the directory where the archive was unpacked, go to the iReport directory, then to the `bin` subdirectory and enter:

   ```
   C:\devel\iReport-nb-3.1.1\bin>ireport.exe
   ```

   For a Unix system, enter:

   ```
   root> ./ireport
   ```

   (In this case, you must configure the installation script to be executable with a `chmod +x` command.)

## 2.6 The Windows Installer

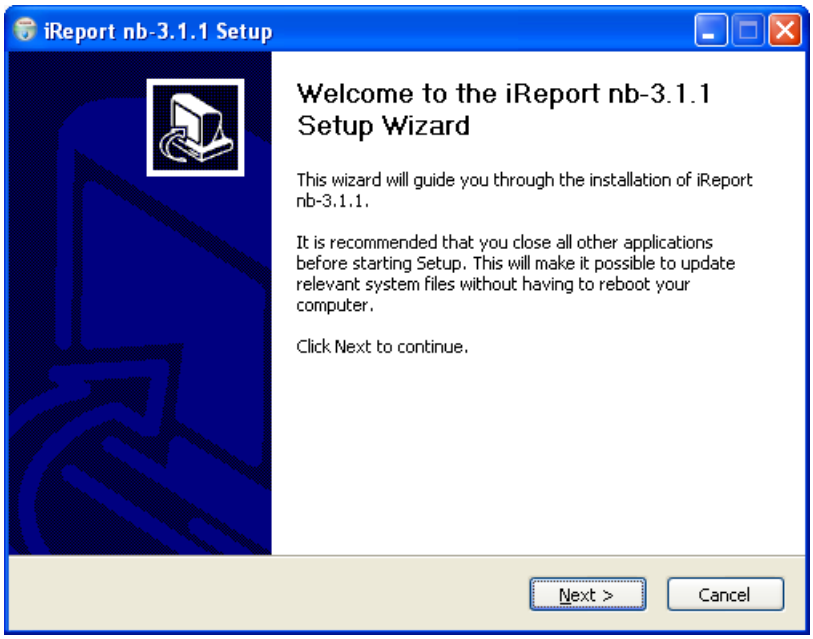



**Figure 2-3 Windows Installer – Step 1**

iReport provides a convenient Windows installer created using NSIS, the popular installer from Nullsoft (http://nsis.sourceforge.net/Main_Page). To install iReport, double-click `iReport-nb-x.x.x-windows-installer.exe` to bring up the screen shown in **Figure 2-4**:

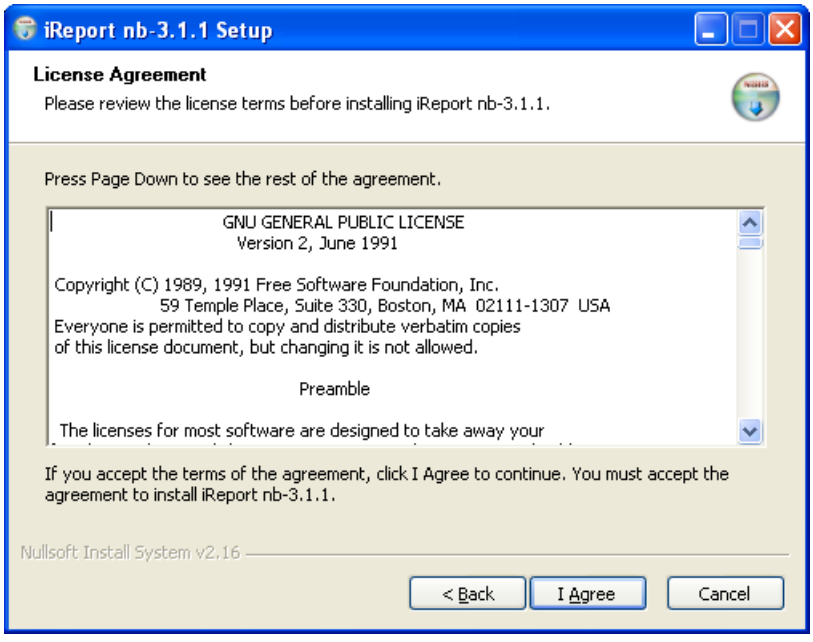



**Figure 2-4 Windows Installer — Step 2**

Click **Next** and follow the instructions on the screen:

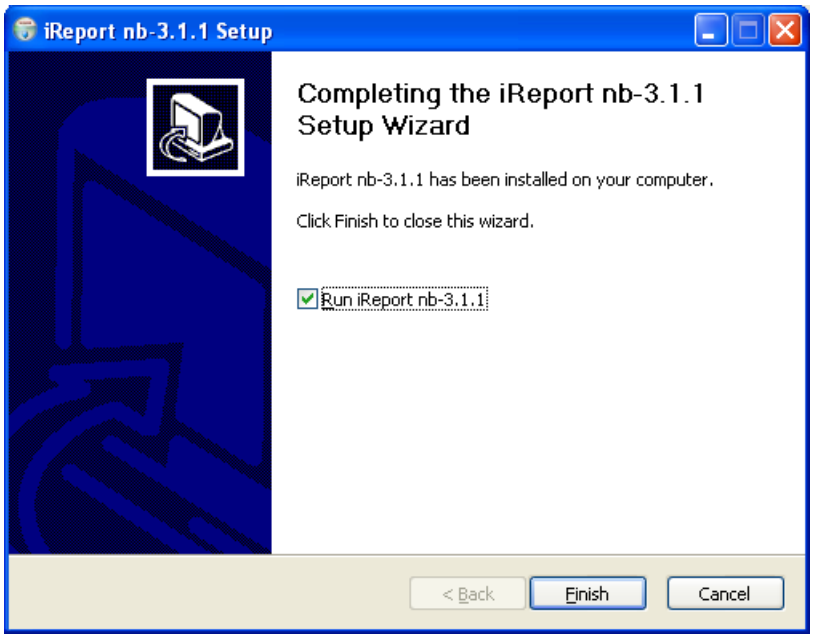



**Figure 2-5 Windows Installer - Last Step**

At the end of the installation process, you get a new menu item in the Program files menu (**Program files** → **JasperSoft** → iReport-nb-x.x.x).

You can have more than one version of iReport installed on your machine at the same time, but all these versions will share the same configuration files.

The installer creates a shortcut to launch iReport, linking the shortcut to `iReport.exe` (present in the `bin` directory under the program home directory).

## 2.7 First iReport Execution

When you run iReport for the first time, you will need to configure a couple of options in order to start designing reports, like a data source to be used with the reports and optionally the location of the external programs to preview the reports (only if you don't want simply use the internal preview).

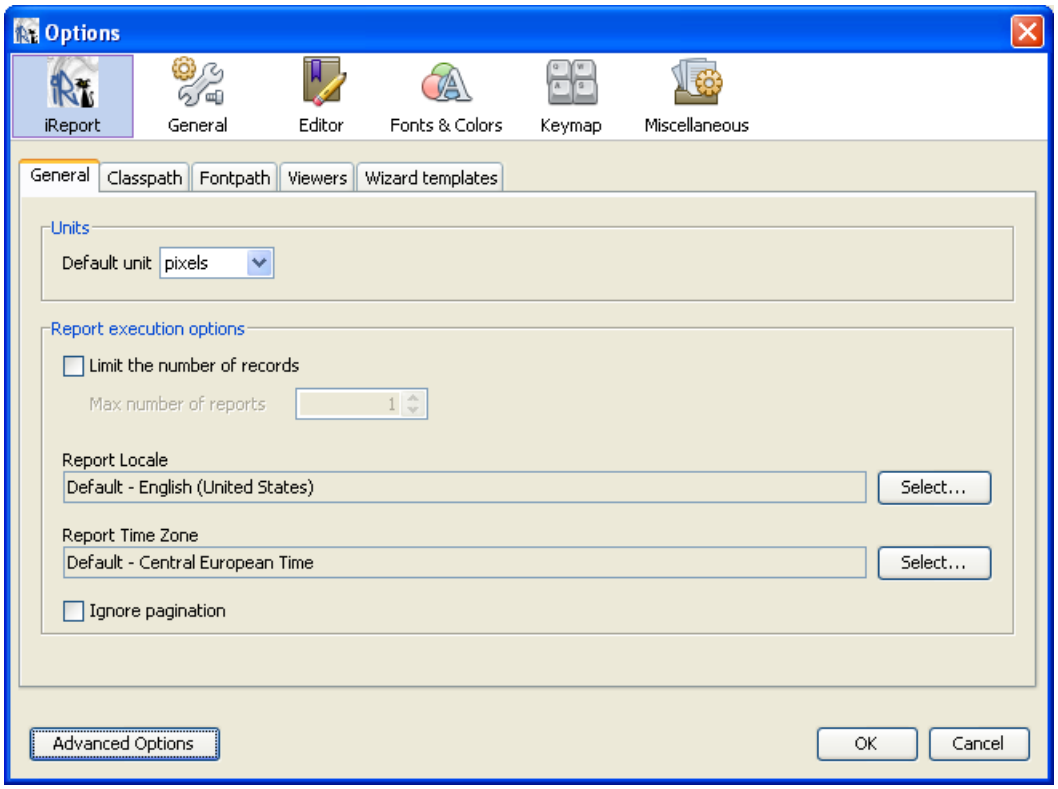



**Figure 2-6 Options Window – General Options**

Run iReport and select **Options** → **Settings** to display the window shown in **Figure 2-6**.

I will discuss all the options shown in this panel in later chapters. For now, click on the **Viewers** tab (see **Figure 2-7**) and configure the appropriate applications you will need to view your output reports.

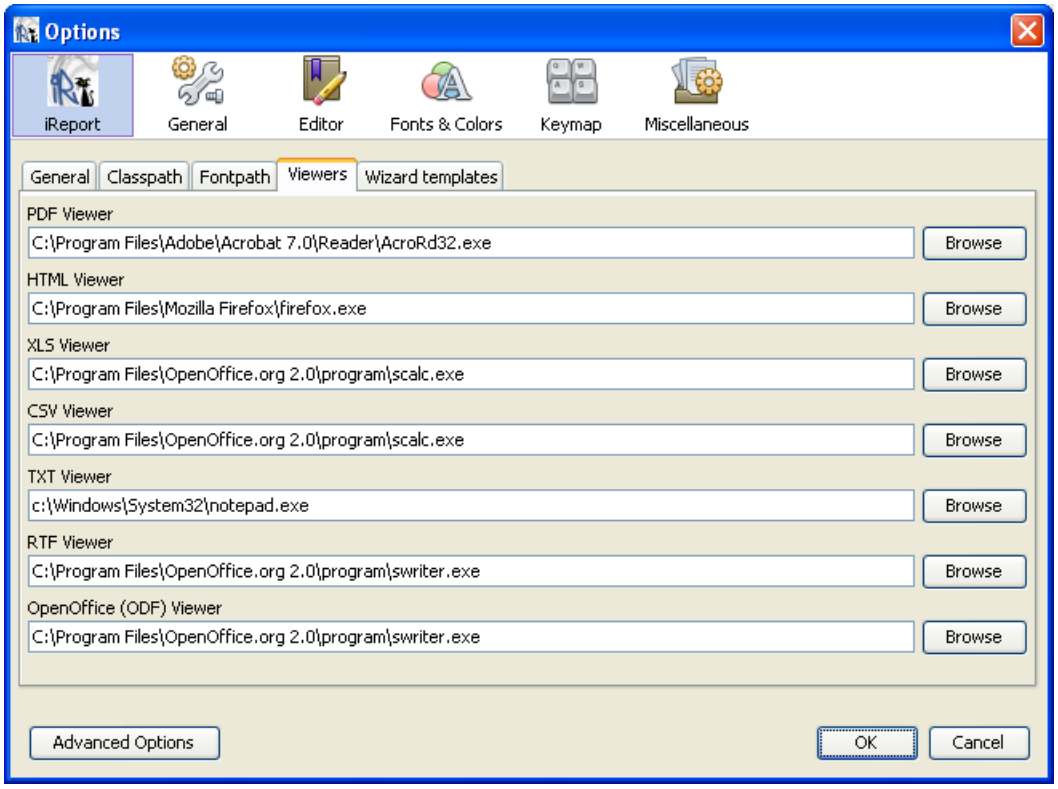



**Figure 2-7 Options Window – Viewers**

Test the configuration by creating a new blank report:

1. Select **File > New** empty report.
2. Select where to save it and confirm.
3. Then just click the **Preview** button on the tool bar.

If everything is OK, iReport generates a JASPER file and displays a preview of a blank page. This means you have installed and configured iReport correctly.

JRXML and JASPER files: iReport stores report templates as xml files, with extension of `.jrxml`. Compiled versions are stored as binary files, with the extension `.jasper`. This last file is the one actually used to generate the report.

## 2.8 Creating a JDBC Connection

The most common data source for filling a report is typically a relational database. Next, you will see how to set up a JDBC connection in iReport. **Select Tools > Report Datasources** and click the **New** button in the window with the connections list. A new window will appear for the configuration of the new connection (see **Figure 2-8**).

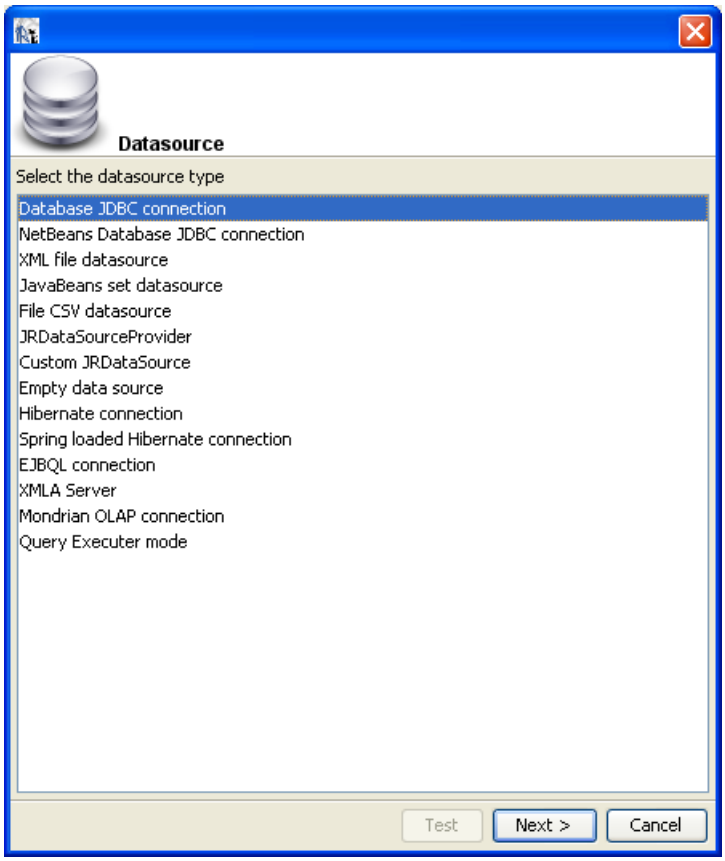



**Figure 2-8 Data source type selection**

Select **Database JDBC connection** and click **Next**. In the new frame, enter the connection name (e.g., “My new connection”) and select the right JDBC driver. iReport recognizes the URL syntax of many JDBC drivers. You can automatically create the URL by entering the server address and database name in the corresponding boxes and clicking the Wizard button. To

complete the connection configuration, enter the username and password for access to the database. If you want to save the password, select the **Save** password check box.

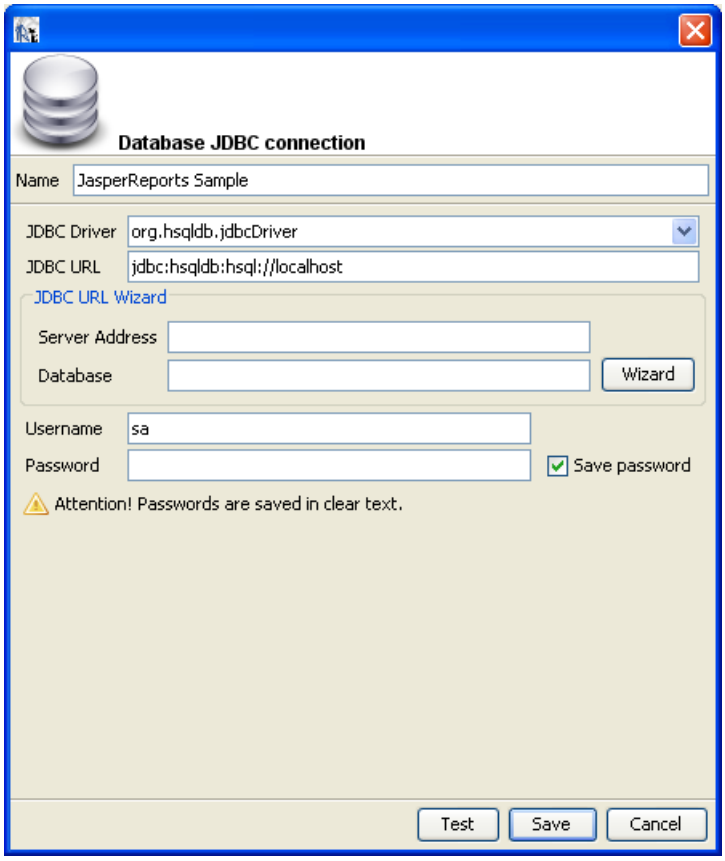



**Figure 2-9 JDBC connection using a built-in JDBC driver**

I suggest that you test the connection configuration before moving on, which you can do by clicking on the **Test** button.

iReport provides the JDBC driver for the following SQL-compliant database systems:

- HSQL
- MySQL
- PostgreSQL

If iReport returns a ClassNotFound error, it is possible that there is no JAR archive (or ZIP) in the classpath that contains the selected database driver.

In this case there are two options:

- Adding the required jar to the iReport classpath
- Registering the new driver through the service window

To extend the iReport classpath, select the menu item **Tools** → **Options**, go to the classpath tab under the iReport category and add the JAR to the list of paths.

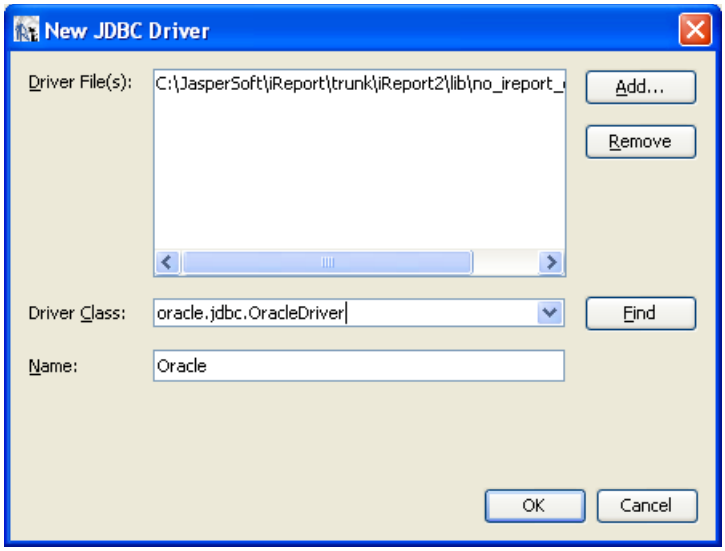



**Figure 2-10 Oracle driver loaded from an external JAR**

If you prefer the second way, open the services window (**Window** → **Services** or Ctl+5), select the Databases node, then the Drivers node, right click on it and select New Driver. The dialog shown in **Figure 2-10** will pop up.

Without closing iReport, copy the JDBC driver into the `lib` directory and retry. iReport automatically locates the required JAR file and loads the driver from it. In the chapter **Data Sources and Query Executors** I will explain the configuration methods for various data sources in greater detail.

At the end of the test, click the **Save** button to store the new connection. It will appear in the data source drop-down list in the main tool bar (figure 1-11). Select it to make it the active connection.

Another way to set the active connection is by opening the data source window (**Figure 2-11**):

1. Select the **Tools** → **Report** data sources menu item (or by clicking the button on the tool bar next to the data sources drop-down list).
2. Select the data source you want to make active.

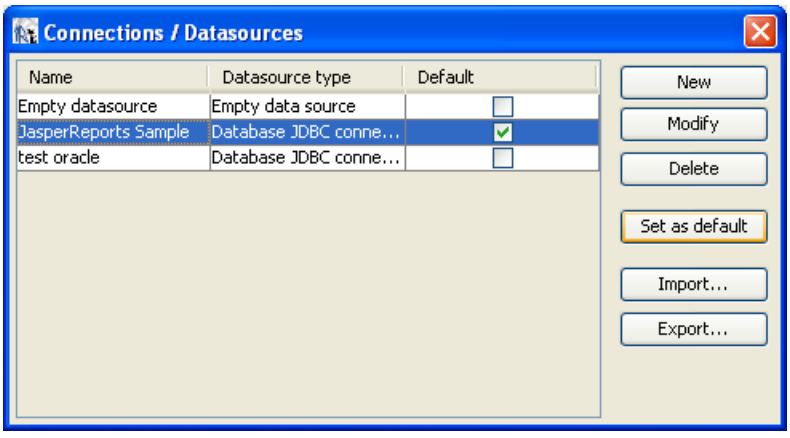



**Figure 2-11 Data Sources Window**

3. Press the button **Set as default**.

The selected data source is the one used to fill the report and perform other operations like the acquisition of the fields selected through SQL queries. There is no strict binding between a report and a data source, so you can run a report with different data sources, but only one at time (we will see how subreports can be used to create a report that uses more that a single data source).

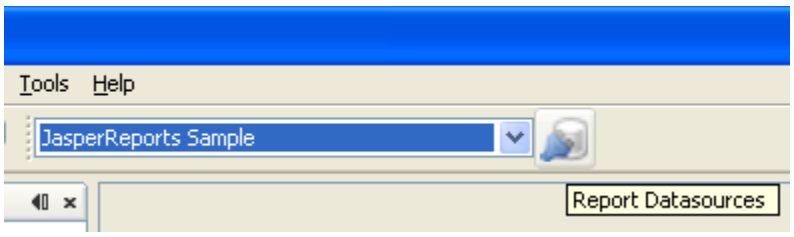



**Figure 2-12 The Data Sources Drop-Down Menu**

The Data Sources Drop-Down Menu allows to select the active data source, the button on the left will open the data sources window.

## 2.9 Creating Your First Report

Now that you have installed and configured iReport, and prepared a JDBC connection to the database, you will proceed to create a simple report using the Wizard.

For it and for many other following examples, you will use HSQLDB, a small relational database written in Java and supplied with a JDBC driver. You can learn more about this small jewel by visiting the HSQLDB project site at this address: http://hsqldb.sourceforge.net.

### 2.9.1 Using the Sample Database

For the samples we will use the sample database that comes with JasperReports. Download JasperReports (the biggest distribution) and unpack it somewhere. Open a command prompt (or a shell), move into the JasperReports folder, the `demo/hsqldb`; if you have Ant (and you know what it is), just run:

```
C:\jasperreports-3.0.1\demo\hsqldb> ant runServer
```

otherwise run this command (all in a single line):

```
C:\jasperreports-3.0.1\demo\hsqldb> java -cp ..\..\lib\hsqldb-1.7.1.jar org.hsqldb.Server
```

The database server will start and we will be ready to use it with iReport.

### 2.9.2 Using the Report Wizard

The table below lists the parameters you should use to connect to the sample database:

| Properties | Value |
|---|---|
| Name | JasperReports Sample |
| JDBC Driver | `org.hsqldb.jdbcDriver` |
| JDBC URL | `jdbc:hsqldb:hsql://localhost` |

| | |
|---|---|
| Username | sa |
| Password | |

When the password is blank, as in this case, remember to set the **Save** password check box when configuring the connection.

Select **File** → **Report Wizard**. This loads a tool for the step-by-step creation of a report, starting with the selection of the name and the location of the new report:

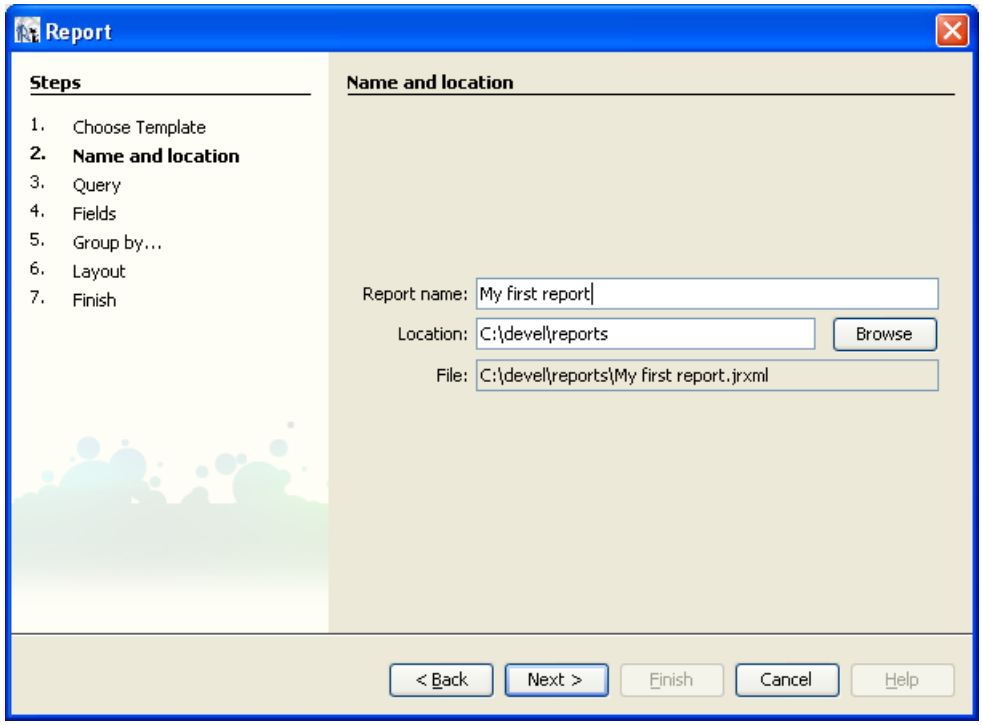



**Figure 2-13 Report Wizard – New Report Name and Location**

In the second step, select the JDBC connection we configured in the previous step (JasperReports Sample). The wizard will detect that we are working with a connection that allows the use of SQL queries and will prompt a text area to specify an SQL query (**Figure 2-14**). Optionally we can visually design the query by pressing the button **Design query**. We assume that you know at least a bit of SQL, so we will directly enter a simple query:

```
select * from address order by city
```

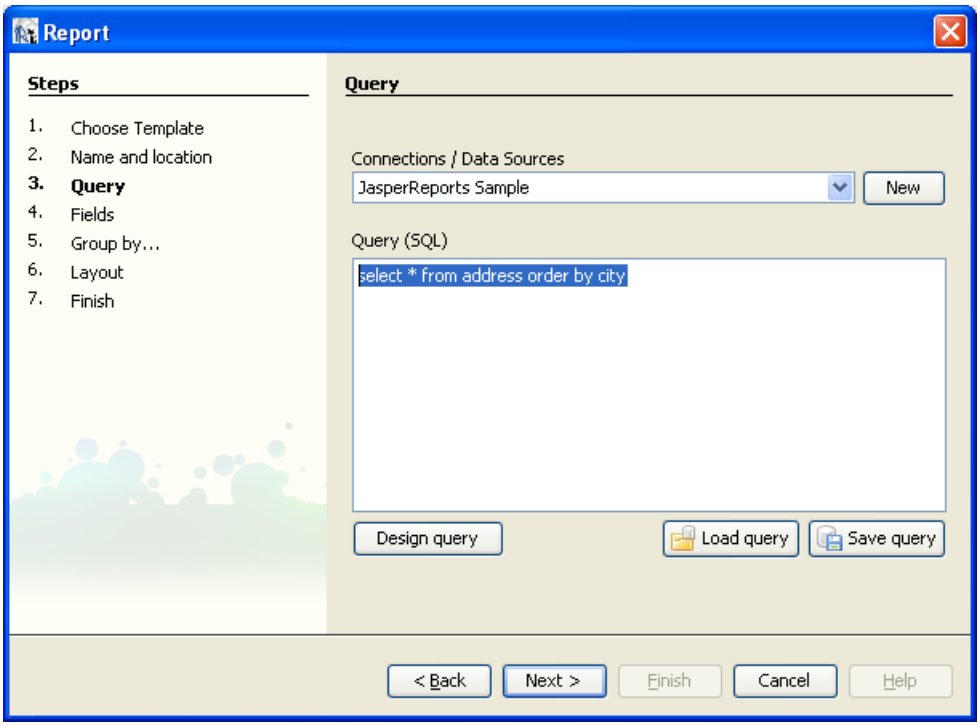



**Figure 2-14 Report Wizard – SQL query**

Click **Next >**. The clause "order by" is important to the following choice of sort order (I will discuss the details a little later). iReport reads the fields of the addresses table, and then presents them in the next screen of the Wizard, as shown in **Figure 2-15**.

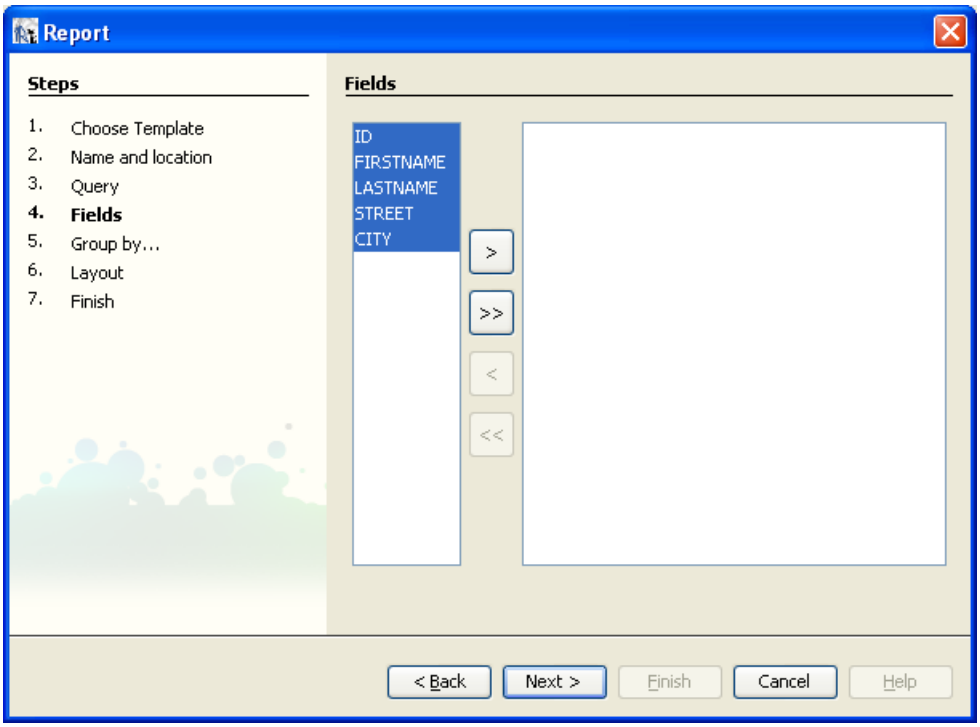



**Figure 2-15 Report Wizard – Fields Selection**

Select the fields you wish to include and click **Next**. Now that you have selected the fields to put in the report, you are prompted to choose which fields to use for sorting, if any (see **Figure 2-15**).

Using the wizard, you can create up to four groups. You can define more fields later. (In fact, it is possible to set up an arbitrary number of groupings.)

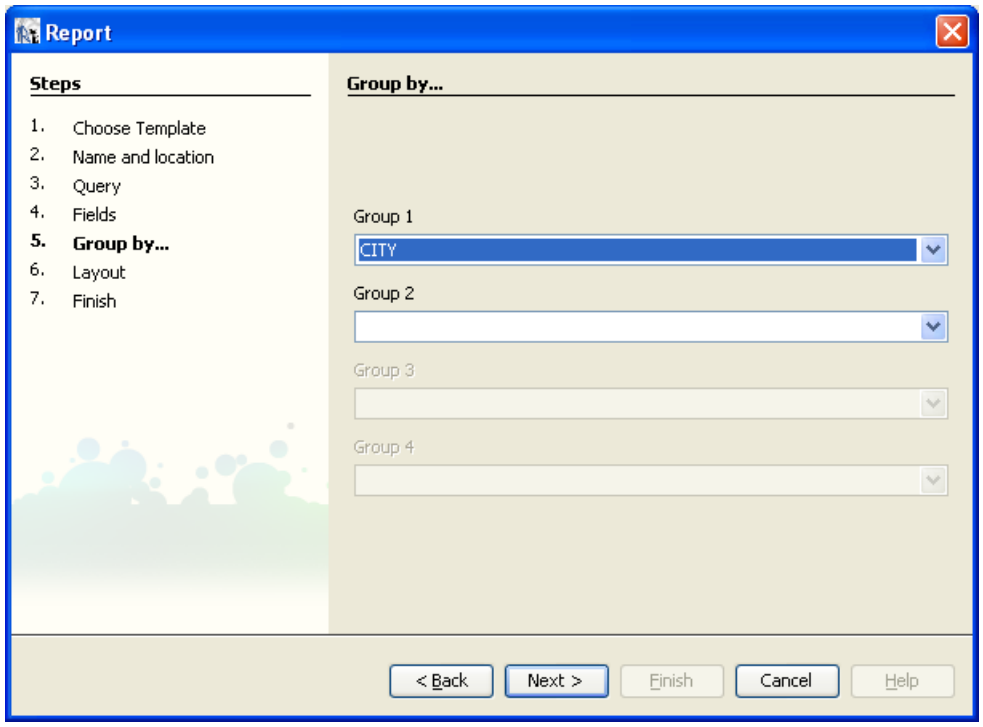



**Figure 2-16 Report Wizard – Grouping**

For this first report, define a simple grouping on the `CITY` field, as shown in **Figure 2-16**.

The next step of the wizard allows you to select the print template, which is a model that you can use as a base for the creation of the report (see **Figure 2-17**). iReport includes a number of basic templates, and later you will see how to create new ones.

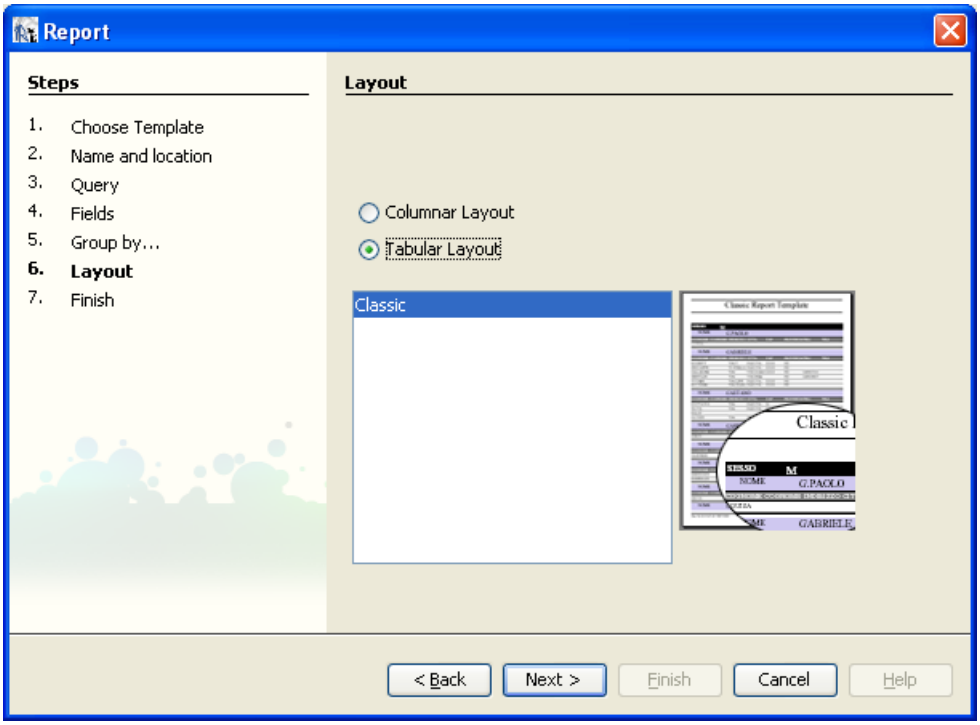



**Figure 2-17 Report Wizard – Choosing a Template**

In this chapter we are going to work with two styles of templates:

- **Tabular Layout** – Every record occupies one line, as in a table
- **Columnar Layout** – Report fields display in columns

For your first report, click on the **Tabular Layout** button and select the Classic template in the list window below.

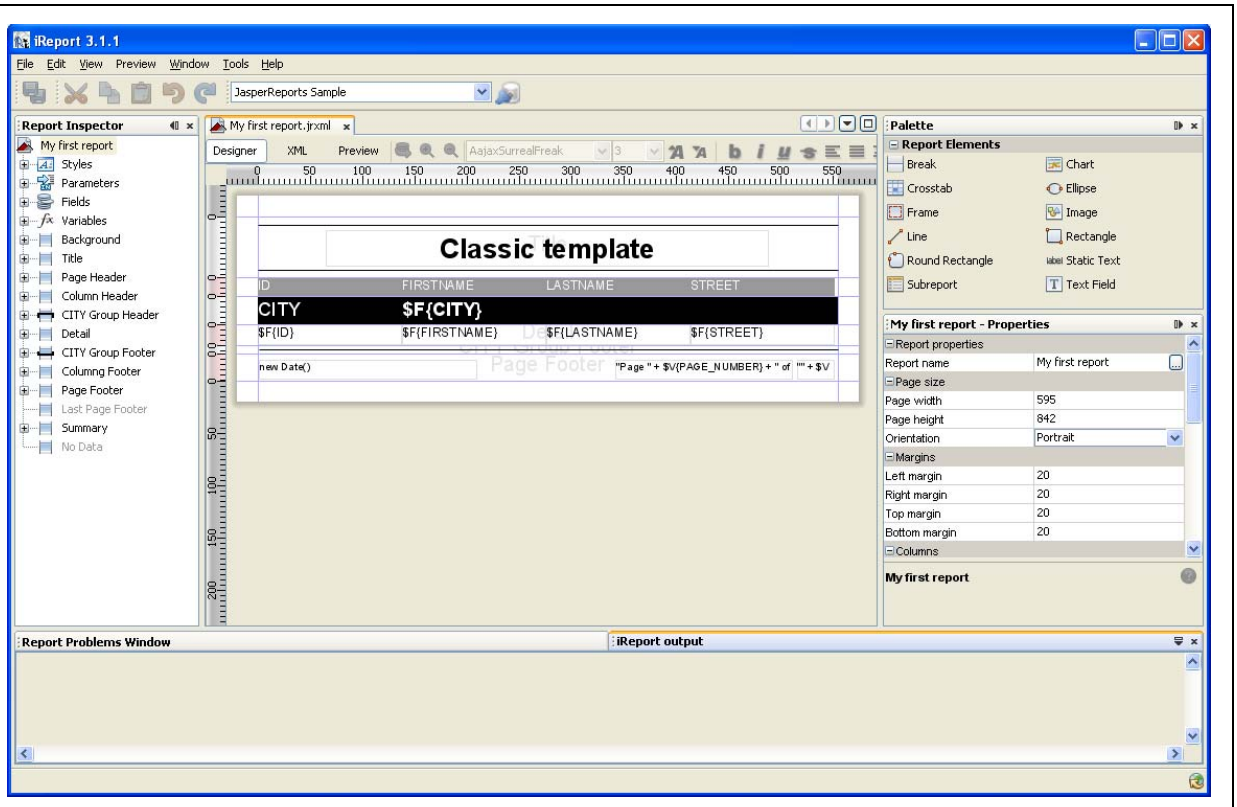



**Figure 2-18 Main Design Window**

After you have chosen the template, click **Next**. The last screen of the wizard will appear, and it will tell you the outcome of the operation. Click **Finish** to create the report, which will appear in the iReport central area, ready to be generated, as shown below. Just press the **Preview** button to see the final result.

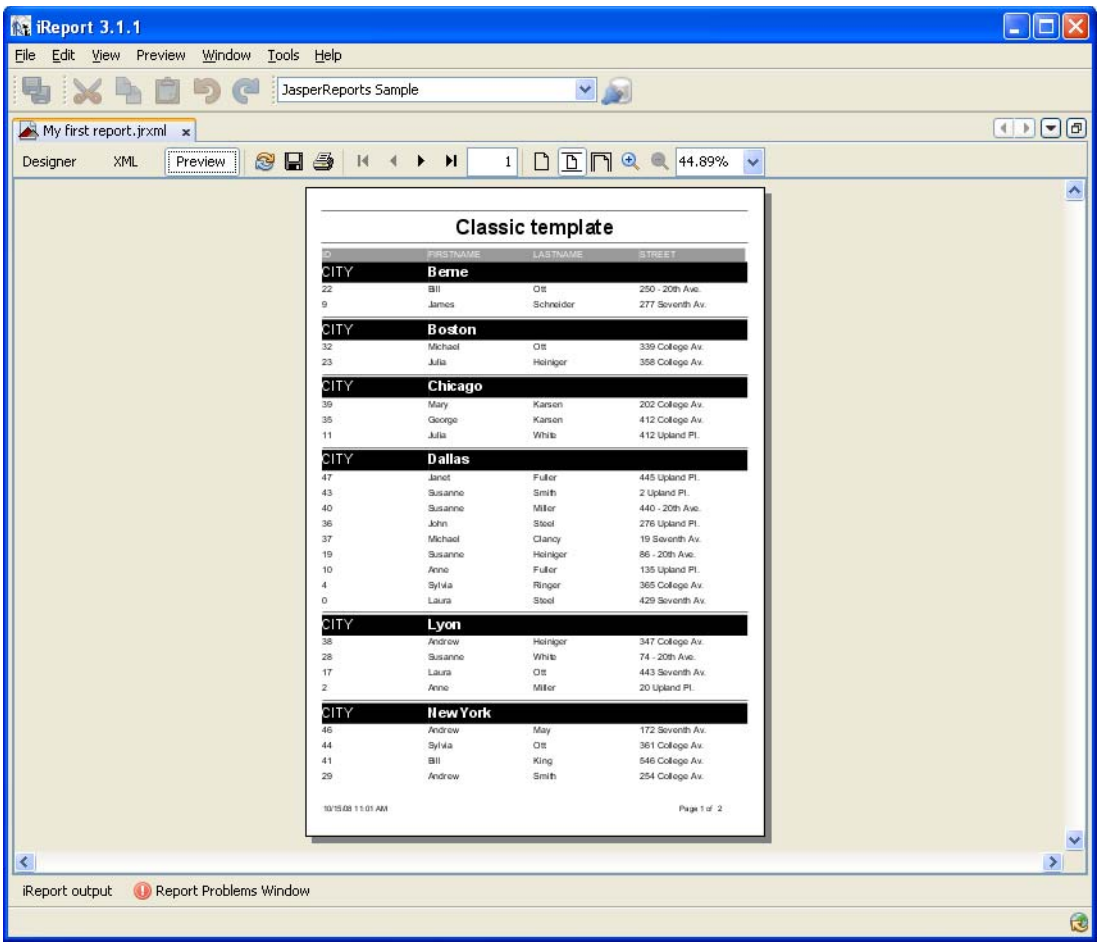



**Figure 2-19 The internal preview**

When you click **Preview**, iReport compiles the report, generating the JASPER file and executing the report against the specified data source. You can track the progress in the output window, which is in the part below the main window. When for some reason the execution fails, you can see a set of problems listed in the **Report Problems** window, and other error tracking information (e.g., a full stack trace) in the iReport output window.

In this case everything should work just fine, and you should see the report in the preview window as shown above.

You can save the report by clicking on the disk icon in the preview window tool bar. iReport can save reports in several formats, including PDF and HTML.

To automatically export the report in a particular format and run the appropriate viewer application, select a format from the menu **Preview**. You can now run the report again from the preview window by clicking the reload button in the preview tool bar, or, if you change the report design, save it and just press **Preview**.

# 3 Basic Notions of JasperReports

The heart of iReport is an open source library named JasperReports, developed and maintained by JasperSoft Corporation under the direction of Teodor Danciu and Lucian Chirita. It is the most widely distributed and powerful free software library for report creation available today.

In this chapter, I will illustrate JasperReports' base concepts for a better understanding of how iReport works.

The JasperReports API, the XML syntax for report definition, and all the details for using the library in your own programs are documented very well in The JasperReports Ultimate Guide. This guide is available from Jaspersoft. Other information and examples are directly available on the official site at http://jasperreports.sourceforge.net.

Unlike iReport, which is distributed according to the GPL license, JasperReports is published under the LGPL license, which is less restrictive. This means that JasperReports can be freely used on commercial programs without buying very expensive software licenses and without remaining trapped in the complicated net of open source licenses. This is fundamental when reports created with iReport have to be used in a commercial product: in fact, programs only need the JasperReports library to produce prints, which work as something like a runtime.

Without the appropriate commercial license (available upon request), you can only use iReport as a development tool. Only programs published under the terms of the GPL license may include iReport as a component.

## 3.1 The Report Life Cycle

When we think about a report, the final document comes in mind just the final document, like a PDF or an Excel file. But this is only the final stage of a report life cycle. It starts with the report design. Design a report means to create some sort of template, like a form where we left some blank space that can be filled with some data. Some portions of a page defined in this way are reused, others stretch to fit the content and so on.

When we are finished, we save this template as an XML file sub-type that we call JRXML. It contains all the basic information about the report layout, including complex formulas to perform calculations, an optional query to retrieve data out of a data source and other functionality we will discuss in detail in later chapters.

A JRXML can not be used as is. For performance reasons, and for the benefit of the program that will run the report, iReport compiles the JRXML and saves it as an executable binary: a JASPER file. A JASPER file is the template that JasperReports will use to generate a report melding the template and the data retrieved from the data source. The result is a “meta print”—an interim output report—that can then be exported in one or more formats, giving life to the final document.

The life cycle can be divided in two distinct action sets:

- Tasks executed during the development phase (design and planning of the report, and compilation of a JASPER file source, the JRXML)

- Tasks that must be executed in a runtime (loading of the JASPER file, filling of the report and export of the print in a final format)

The main role of iReport is to design a report and create an associated JASPER file, though it is able to preview the result and export it in all the supported formats. iReport also provides the support for a wide range of data sources and allows the user to test their own ones, becoming a complete environment for report development and testing.

## 3.2 JRXML Sources and JASPER Files

As already explained, JasperReports defines a report with an XML file. In previous versions, JasperReports defined the XML syntax with a DTD file (`jasperreport.dtd`). Starting with Version 3.0.1, JasperReports changed the definition method to allow for support of user defined report elements. The set of tags was extended and the new XML documents must be validated using an XML-Schema document (`jasperreport.xsd`).

A JRXML file is composed of a set of sections, some of them concerning the report's physical characteristics, such as the dimension of the page, the positioning of the fields, and the height of the bands; and some of them concerning the logical characteristics, such as the declaration of the parameters and variables, and the definition of a query for the data selection.

The syntax has grown more and more complicated with the maturity of JasperReports. This is why many times a tool like iReport is indispensable.

The following figure shows the source code of the report generated in the previous chapter; you can see the result is shown in **Figure 2-19 on page 25**.

**Figure 3-1 A Simple JRXML File Example**

```
<?xml version="1.0" encoding="UTF-8"?>
<jasperReport xmlns="http://jasperreports.sourceforge.net/jasperreports"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
jasperreports.sourceforge.net/jasperreports http://jasperreports.sourceforge.net/
xsd/jasperreport.xsd" name="My first report" pageWidth="595" pageHeight="842"
columnWidth="535" leftMargin="20" rightMargin="20" topMargin="20" bottomMargin="20">
 <queryString language="SQL">
  <![CDATA[select * from address order by city]]>
 </queryString>
 <field name="ID" class="java.lang.Integer">
  <fieldDescription><![CDATA[]]></fieldDescription>
 </field>
 <field name="FIRSTNAME" class="java.lang.String">
  <fieldDescription><![CDATA[]]></fieldDescription>
 </field>
 <field name="LASTNAME" class="java.lang.String">
  <fieldDescription><![CDATA[]]></fieldDescription>
 </field>
 <field name="STREET" class="java.lang.String">
  <fieldDescription><![CDATA[]]></fieldDescription>
 </field>
 <field name="CITY" class="java.lang.String">
  <fieldDescription><![CDATA[]]></fieldDescription>
 </field>
```

**Figure 3-1 A Simple JRXML File Example**

```
<group name="CITY">
 <groupExpression><![CDATA[$F{CITY}]]></groupExpression>
 <groupHeader>
  <band height="27">
   <staticText>
    <reportElement mode="Opaque" x="0" y="0" width="139" height="27"
    forecolor="#FFFFFF" backcolor="#000000"/>
    <textElement>
     <font size="18"/>
    </textElement>
    <text><![CDATA[CITY]]></text>
   </staticText>
   <textField hyperlinkType="None">
    <reportElement mode="Opaque" x="139" y="0" width="416" height="27"
    forecolor="#FFFFFF" backcolor="#000000"/>
    <textElement>
     <font size="18" isBold="true"/>
    </textElement>
    <textFieldExpression class="java.lang.String"><![CDATA[$F{CITY}]]></
    textFieldExpression>
   </textField>
  </band>
 </groupHeader>
```

**Figure 3-1 A Simple JRXML File Example**

```
  <groupFooter>
   <band height="8">
     <line direction="BottomUp">
       <reportElement key="line" x="1" y="4" width="554" height="1"/>
     </line>
   </band>
  </groupFooter>
</group>
<background>
 <band/>
</background>
<title>
 <band height="58">
  <line>
    <reportElement x="0" y="8" width="555" height="1"/>
  </line>
  <line>
    <reportElement positionType="FixRelativeToBottom" x="0" y="51" width="555"
    height="1"/>
  </line>
  <staticText>
    <reportElement x="65" y="13" width="424" height="35"/>
    <textElement textAlignment="Center">
      <font size="26" isBold="true"/>
    </textElement>
    <text><![CDATA[Classic template]]></text>
  </staticText>
 </band>
</title>
```

**Figure 3-1 A Simple JRXML File Example**

```
	<pageHeader>
		<band/>
	</pageHeader>
	<columnHeader>
		<band height="18">
			<staticText>
				<reportElement mode="Opaque" x="0" y="0" width="138"
height="18" forecolor="#FFFFFF" backcolor="#999999"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<text><![CDATA[ID]]></text>
			</staticText>
			<staticText>
				<reportElement mode="Opaque" x="138" y="0" width="138"
height="18" forecolor="#FFFFFF" backcolor="#999999"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<text><![CDATA[FIRSTNAME]]></text>
			</staticText>
			<staticText>
				<reportElement mode="Opaque" x="276" y="0" width="138"
height="18" forecolor="#FFFFFF" backcolor="#999999"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<text><![CDATA[LASTNAME]]></text>
			</staticText>
			<staticText>
				<reportElement mode="Opaque" x="414" y="0" width="138"
height="18" forecolor="#FFFFFF" backcolor="#999999"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<text><![CDATA[STREET]]></text>
			</staticText>
		</band>
	</columnHeader>
```

**Figure 3-1 A Simple JRXML File Example**

```
	<detail>
		<band height="20">
			<textField hyperlinkType="None">
				<reportElement x="0" y="0" width="138" height="20"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<textFieldExpression
class="java.lang.Integer"><![CDATA[$F{ID}]]></textFieldExpression>
			</textField>
			<textField hyperlinkType="None">
				<reportElement x="138" y="0" width="138" height="20"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<textFieldExpression
class="java.lang.String"><![CDATA[$F{FIRSTNAME}]]></textFieldExpression>
			</textField>
			<textField hyperlinkType="None">
				<reportElement x="276" y="0" width="138" height="20"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<textFieldExpression
class="java.lang.String"><![CDATA[$F{LASTNAME}]]></textFieldExpression>
			</textField>
			<textField hyperlinkType="None">
				<reportElement x="414" y="0" width="138" height="20"/>
				<textElement>
					<font size="12"/>
				</textElement>
				<textFieldExpression
class="java.lang.String"><![CDATA[$F{STREET}]]></textFieldExpression>
			</textField>
		</band>
	</detail>
	<columnFooter>
		<band/>
	</columnFooter>
```

**Figure 3-1 A Simple JRXML File Example**

```
      <pageFooter>
            <band height="26">
                  <textField evaluationTime="Report" pattern=""
isBlankWhenNull="false" hyperlinkType="None">
                        <reportElement key="textField" x="516" y="6" width="36"
height="19" forecolor="#000000" backcolor="#FFFFFF"/>
                        <box>
                              <topPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                              <leftPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                              <bottomPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                              <rightPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                        </box>
                        <textElement>
                              <font size="10"/>
                        </textElement>
                        <textFieldExpression class="java.lang.String"><![CDATA[""
+ $V{PAGE_NUMBER}]]></textFieldExpression>
                  </textField>
                  <textField pattern="" isBlankWhenNull="false"
hyperlinkType="None">
                        <reportElement key="textField" x="342" y="6" width="170"
height="19" forecolor="#000000" backcolor="#FFFFFF"/>
                        <box>
                              <topPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                              <leftPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                              <bottomPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                              <rightPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                        </box>
```

**Figure 3-1 A Simple JRXML File Example**

```
                        <textElement textAlignment="Right">
                               <font size="10"/>
                        </textElement>
                        <textFieldExpression
class="java.lang.String"><![CDATA["Page " + $V{PAGE_NUMBER} + " of "]]></
textFieldExpression>
                  </textField>
                  <textField pattern="" isBlankWhenNull="false"
hyperlinkType="None">
                        <reportElement key="textField" x="1" y="6" width="209"
height="19" forecolor="#000000" backcolor="#FFFFFF"/>
                        <box>
                               <topPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                               <leftPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                               <bottomPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                               <rightPen lineWidth="0.0" lineStyle="Solid"
lineColor="#000000"/>
                        </box>
                        <textElement>
                               <font size="10"/>
                        </textElement>
                        <textFieldExpression class="java.util.Date"><![CDATA[new
Date()]]></textFieldExpression>
                  </textField>
            </band>
      </pageFooter>
      <summary>
            <band/>
      </summary>
 </jasperReport>
```

During compilation of the JRXML file (done through some JasperReports classes), the XML is parsed and loaded in a JasperDesign object, a rich data structure that allows you to represent the exact XML contents in memory. Without going into details, regardless of which language is used for expressions inside the JRXML, JasperReports create a special Java class that represents the whole report which is then compiled, instanced and serialized in a JASPER file, ready for loading at any time.

The JasperReports speed is due to all of the report's formulas being compiled into Java-native bytecode and the report structure verified during compilation, not runtime. The JASPER file does not contain extraneous resources, like images used in the report, resource bundles to run the report in different languages or extra scriptlets or external style definitions. All these resources must be located at run time and provided by the host application.

## 3.3 Data Sources and Print Formats

Without a means of supplying content from a dynamic data source, even the most sophisticated and appealing report would be useless. JasperReports allows you to specify fill data for the output report in two different ways: through parameters and data sources, which are presented by means of a generic interface named `JRDataSource`, as shown in **Figure 3-2**.

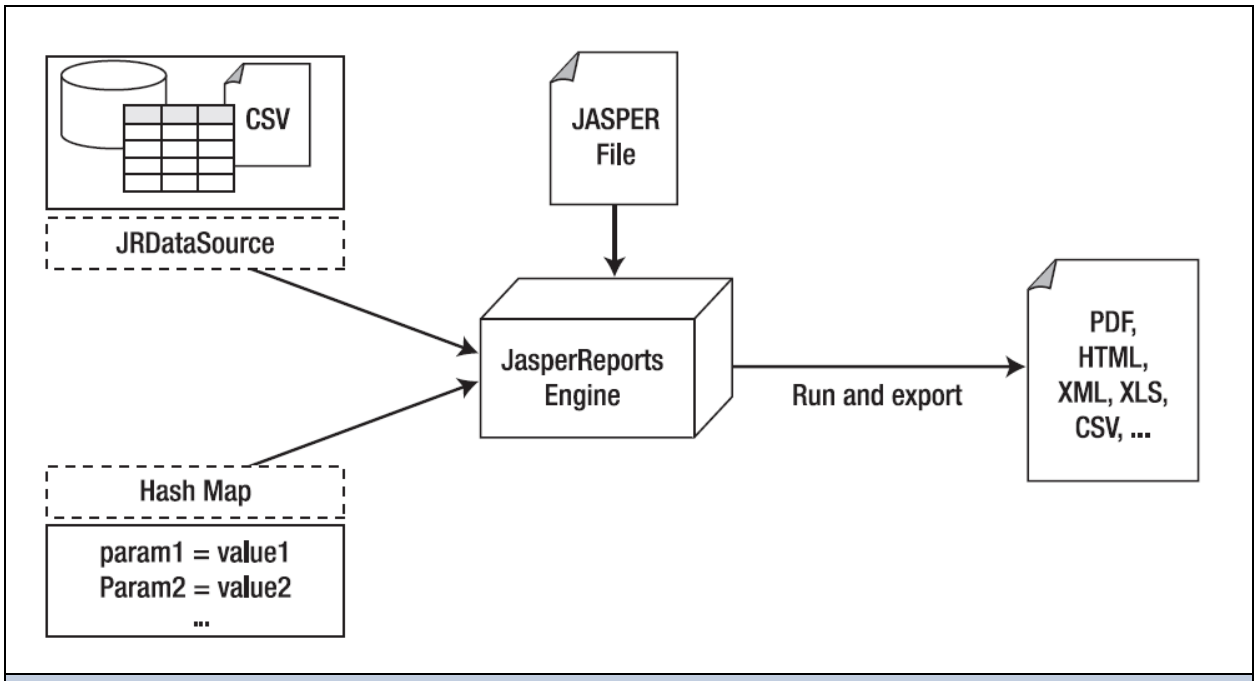



**Figure 3-2 Data source and Parameter Flows for Report Creation**

The chapter **Data Sources and Query Executors** is dedicated to data sources; it explains how they can be used in iReport and how it is possible to define custom data sources (in case those supplied with JasperReports are not right for your requirements). `JRDataSource` allows a set of records that are organized in tables (rows and columns) to be read.

Instead of using an explicit data source to fill in a report, JasperReports is able to do so through a JDBC connection (already instanced and opened) to whichever relational database you want to run a SQL query on (which is specified in the report).

If the data (passed through a data source) is not enough to meet the requirements of the user, i.e. when it is necessary to specify particular values to condition its execution, it is possible to produce some name/value pairs to "transmit" to the print engine. These pairs are named parameters, and they have to be "preventively declared" in the report. Through the `fillManager`, it is possible to join a JASPER file and a data source in a JasperPrint object. This object is a meta-print that can create a real print after you have exported it in the desired format through appropriate classes that implement the `JRExporter` interface.

JasperReports puts at your disposal different predefined exporters like those for creating files in such formats as PDF, XLS, CVS, XML, RTF, ODF, text, HTML and SWF. Through the `JRViewer` class, you can view the print directly on the screen and print it.

## 3.4 Compatibility Between Versions

When a new version of JasperReports is distributed, usually some classes change. These modified classes typically impact the XML syntax and the JASPER file structure.

Before JasperReports 1.1.0, this was a serious problem and a major upgrade deterrent, since it required recompiling all the JRXML files in order to be used with the new library version. Things changed after the release of Version 1.1.0, after which JasperReports assures backward compatibility, meaning that the library is able to understand and execute any JASPER file generated with a previous version of JasperReports.

With JasperReports 3.1, the JRXML syntax has moved from a DTD based definition to XML-schema. The XML source declaration syntax now references a schema file, rather than a DTD. Based on what we said previously, this is not a problem since JasperReports assures backward compatibility, but many people are used to designing reports with the last version of iReport and then expect to use them with a system able to compile JRXML by using a previous version of JasperReports. This was always a risky operation, but it was still legal because the user was not using a new tag in the xml. With the move to XML-schema, the JRXML output of iReport 3.1.1 and newer can only be compiled with a JasperReports 3.1.0 or later.

## 3.5 Expressions

Though I designed iReport to be useful to non-technical report designers, many settings in a report are defined using formulas (like conditions to hide an element, special calculations, text processing, etc...) that require a minimum knowledge of a scripting language.

Fortunately formulas can be written in at least three languages, two of which (JavaScript and Groovy) are pretty simple and can easily used even without knowledge of programming methods.

All of the formulas in JasperReports are defined through expressions. The default expression language is Java, but I suggest that you design your projects with JavaScript or Groovy. Both hide a lot of the Java complexity and are definitively the languages to use if you don't know Java. The language is a property of the document, so to set it, select the document root node in the outline view and choose your language in the Language property in the properties view. We will go through all the languages in the following sections, but let's concentrate for a moment on our definition of an “expression,” in particular the type you will declare for it and why that is important in JasperReports.

An expression is just a formula that operates on some values returning a result. Think of an expression as the formula you might define for a spreadsheet cell. You can have a simple value or you can use a complex formula that can refer other values (in a spreadsheet you would refer to values contained in other cells, in JasperReports you will use the report fields, parameters, and variables). The main point is that whatever you have in your expression, when it is computed it gives a value as result (which can be null; that's still a value).

### 3.5.1 The Type of an Expression

The type of an expression is the nature of the value resulting from an expression, and it is determined by the context in which the expression is used. For example, if your expression is used to evaluate a condition, the type of the expression should be Boolean (true or false); if you are creating an expression that should be displayed in a text field, it will probably be a String or a number (Integer or Double). We could simplify the declaration of types by limiting them to text, numbers, Booleans, and generic object values. Unfortunately, JasperReports is a bit more formal and in many cases you have to be very precise when setting the type of your expression.

So far, we are discussing only Java types (regardless of the language used). Some of the most important types are:

| | |
|---|---|
| `java.lang.Boolean` | Defines an Object that represents a boolean value like true and false |
| `java.lang.Byte` | Defines an Object that represents a byte |
| `java.lang.Short` | Defines an Object that represents an short integer |
| `java.lang.Integer` | Defines an Object that represents integer numbers |
| `java.lang.Long` | Defines an Object that represents long integer numbers |
| `java.lang.Float` | Defines an Object that represents floating point numbers |
| `java.lang.Double` | Defines an Object that represents real numbers |
| `java.lang.String` | Defines an Object that represents a text |
| `java.util.Date` | Defines an Object that represents a date or a timestamp |
| `java.lang.Object` | A generic java Object |

As noted, if the expression is used to determine the value of a condition that determines, for instance, whether an element should be printed, the return type will be `java.lang.Boolean`; to create it you need an expression that returns an instance of a Boolean object. Similarly, if I'm writing the expression to show a number in a text field, the return type will be:

`java.lang.Integer` or `java.lang.Double`.

Fortunately, JavaScript, and Groovy are not particularly formal about types, since they are not typed languages: the language itself treats a value in the best way by trying to guess the value type or performing implicit casts (conversion of the type).

### 3.5.2 Expression Operators and Object Methods

All the operators are similar in Java, Groovy and JavaScript, since these languages essentially share the same basic syntax. Operators can be applied to a single operand (unary operators) or on two operands (in which case they are defined as binary operators).

**Table 3-1Operators**

| Operator | Description | Example |
|---|---|---|
| + | Sum (it can be used to sum two numbers or to concatenate two strings) | A + B |
| - | Subtraction | A - B |
| / | Division | A / B |
| % | Rest, it returns the rest of an integer division | A % B |
| \|\| | Boolean operator OR | A \|\| B |
| && | Boolean operator AND | A && B |
| == | Equals* | A == B |
| != | Not equals† | A != B |
| ! | Boolean operator NOT | !A |

*.In Java the == operator can be used only to compare two primitive values, with objects you need to use the special method “equals”, in example you can not write an expression like: `“test” == “test”`, you need to write `“test”.equals(“test”)`.
†.This operator follows the same note as Equals.

**Figure 3-2** shows a number of operators: this is not a complete list; they are the ones I suggest. For instance, there is a unary operator to add 1 to a variable (++) but in my opinion it is not easy to read and can be easily replaced with x + 1. Better, no?

Within the expression, you can refer to the parameters, variables, and fields which are defined in the report using the syntax summarized in**Table 3-2**

**Table 3-2Syntax for Referring to Report Objects**

| Syntax | Description |
|---|---|
| `$F{name_field}` | Specifies the `name_field` field ("F" means field) |
| `$V{name_variable}` | Specifies the `name_ variable` variable |
| `$P{name_parameter}` | Specifies the `name_parameter` parameter |
| `$P!{name_parameter}` | Special syntax used in the report SQL query to indicate that the parameter does not have to be dealt as a value to transfer to a prepared statement, but that it represents a little piece of the query |
| `$R{resource_key}` | Special syntax for localization of strings |

We will describe the nature of fields, variables, and parameters in the next chapter. For now we just have to keep in mind that they always represent objects (i.e., they can have a null value) and that you specify their type when you declare them within a report. Version 0.6.2 of JasperReports introduced a new syntax: `$R{resource_key}`. This is used to localize strings. I will discuss this at greater lengths in the **Internationalization** chapter.

In spite of the possible complexity of an expression, usually it is a simple operation that returns a value. It is not a snippet of code, or a set of many instructions, and you can not use complex constructs or flow control keywords, such as switches, loops, for and while cycles, if and else.

Anyway, there is a simple if-else construct very useful in many situations. An expression is just an arbitrary operation (however complicated) that returns a value. You can use all the mathematical operators or call object methods, but at any stage the expression must represent a value.

In Java, all these operators can be applied only to primitive values, except for the sum operator (+). The sum operator can be applied to a String expression with the special meaning of "concatenate". So for example:

```
$F{city} + ", " + $F{state}
```

will result in a string like: San Francisco, California.

All the objects may include methods. A method can accept zero or more arguments, and it can return or not a value; in an expression you can use only methods that return a value (otherwise you would have nothing to return from your expression...). The syntax of a method call is:

`Object.`*method*`(`*argument 1*`,` *argument 2*`,` etc... )

Some examples:

| Expression | Result |
|---|---|
| "test".length() | 4 |
| "test".substring(0, 3) | "tes" |
| "test".startsWith("A") | false |
| "test".substring(1, 2).startsWith("e") | true |

All the methods of each object are usually explained in a set of documents called *Javadocs* freely available on Internet.

You can use parentheses to isolate expressions and make the overall expression more readable.

### 3.5.3 Using an If-Else Construct in an Expression

A way to create an if-else like expression is by using the special question mark operator. Here is a sample:

```
(($F{name}.length() > 50) ? $F{name}.substring(0,50) : $F{name})
```

The syntax is *(<condition>) ? <value on true> : <value on false>*. It is extremely useful, and the good news is that it can be recursive, meaning that the value on true and on false can be represented by another expression which can be a new condition:

```
(($F{name}.length() > 50) ?
(($F{name}.startsWidth("A")) ? "AAAA" : "BBB")
    :
      $F{name})
```

This expression returns the String "AAAA" when the value of the field name is longer than 50 characters and starts with A, returns BBB if it is longer than 50 characters but it does not start with A, and finally return the original field value if none of the previous conditions are true.

Despite the possible complexity of an expression (having multiple if-else instructions and so on), it can be insufficient to define a needed value. For example, if you want to print a number in Roman numerals or give back the name of the weekday of a date, it is possible to transfer the elaborations to an external Java class method, which must be declared as static, as shown in the following:

```
MyFormatter.toRomanNumber( $F{MyInteger}.intValue() )
```

The function operand `toRomanNumber` is a static method of the `MyFormatter` class, which takes an `int` as argument (the conversion from Integer to `int` is done by means of the `intValue()` method and it is required only when using Java as language) and gives back the Roman version of a number in a lace.

This technique can be used for many aims, for example, to read the text from a CLOB field or to add a value into a HashMap (a convenient Java object to represent a set of key/values pairs).

## 3.6 Using Java As a Language for Expressions

First of all, there is no reason to prefer Java over other languages when working with iReport. It is the first language supported by JasperReports and this is the only reason for which it is still the commonly used language (and the default one).

Following are some examples of Java expressions:

- “This is an expression”
- new Boolean(true)
- new Integer(3)
- (($P{MyParam}.equals("S")) ? "Yes" : "No")

The first thing to note is that each of these expressions represent a Java Object, meaning that the result of each expression is a non-primitive value. The difference between an object and a primitive value makes sense only in Java, but it is very important: a primitive value is a pure value like the number 5 or the boolean value true. Operations between primitive values have as a result a new primitive value, so the expression:

```
5+5
```

results in the primitive value 10. Object are complex types that can have methods, can be null and must be “instanced” with the keyword “new” most of the time. In the second example, for instance, we must wrap the primitive value `true` in an object that represents it.

In a scripting language (like Groovy and JavaScript), primitive values are automatically wrapped into objects, so the distinction between primitive values and objects wanes. When using Java, the result of our expression must be an object, that's why the expression 5+5 is not legal as is but must be fixed with something like:

```
new Integer( 5 + 5 )
```

that creates a new object of type Integer representing the primitive value 10.

So, if you use Java as default language for your expressions, remember that expressions like:

- 3 + 2 * 5
- true
- (($P{MyParam} == 1) ? "Yes" : "No")

are not valid because they don't make the correct use objects. In particular, the first and the second expressions are not valid because they are of a primitive type (integer in the first case and boolean in the second case) which does not produce an object as result.

The third expression is not valid because it assumes that the `MyParam` parameter is a primitive type and that it can be compared through the == operator with an `int`, but this is not true, in fact we said that parameters, variables and fields are always objects and primitive values can not be compared or used directly in a mathematical expression with an object.

Since JasperReports is compiled to work with Java 1.4, the auto-boxing functionality of Java 1.5, that would in some cases solve the use of objects as primitive values and vice versa, is not leveraged.

## 3.7 Using Groovy As a Language for Expressions

The modular architecture of JasperReports provides a way to plug the support for languages other than Java to be used in the expressions; by default the library supports other two different languages: Groovy and JavaScript (this last one from the version 3.1.3).

Groovy is a full language for the Java 2 Platform: this means that inside the Groovy language you can use all classes and JARs available for Java.

Table 3-3 compares some typical JasperReports expressions written in Java and in Groovy.

**Table 3-3Groovy and Java Samples**

| Expression | Java | Groovy |
|---|---|---|
| Field | `$F{field_name}` | `$F{field_name}` |
| Sum of two double fields | `new Double($F{f1}.doubleValue() + $F{f2}.doubleValue())` | `$F{f1} + $F{f2}` |
| Comparison of numbers | `new Boolean($F{f}.intValue() == 1)` | `$F{f} == 1` |
| Comparison of strings | `new Boolean($F{f} != null && $F{f}.equals("test"))` | `$F{f} == "test"` |

The following is a correct Groovy expression:

```
new JREmptyDataSource($F{num_of_void_records})
```

`JREmptyDataSource` is a class of JasperReports that creates an empty record set (meaning with the all fields set to null). You can see how you can instance this class (a pure Java class) in Groovy without any problem. At the same time, Groovy allows you to use a simple expression like this one:

```
5+5
```

The language automatically encapsulates the primitive value 10 (the result of that expression) in a proper object. Actually, you can do more: you can treat this value as an object of type `String` and create an expression such as:

```
5 + 5+ ”my value”
```

Whether such an expression resolves to a rational value, it is still a legal expression and the result will be an object of type `String` with the value:

```
10 my value
```

Hiding the difference between objects and primitive values, Groovy allows the comparison of different types of objects and primitive values, such as the legal expression:

```
$F{Name} == “John”
```

that returns true or false, or again:

| | |
|---|---|
| `$F{Age} > 18` | Returns true if the Age object interpreted as number is greater than 18 |
| `“340” < 100` | Always returns false |
| `“340”.substring(0,2) < 100` | Always returns true (since the substring method call will produce the string “34” that is less than 100. |

Groovy provides a way to simplify a lot the expressions and never complains about null objects (that can crash a Java expression throwing a `NullPointerException`). It really does open the doors of JasperReports to those people who don't know Java.

## 3.8 Using JavaScript As a Language for Expressions

JavaScript is a popular scripting language with a syntax very similar to Java and Groovy. The support for JavaScript has been requested for a long time from the community and has been finally introduced starting from JasperReports 3.1.2 using the open source Rhino JavaScript implementation.

JavaScript has his set of functions and object methods that in some case differs from Java and Groovy, in example in JavaScript does not exists the method *String.startsWith(...)*. The good news is that you can still use Java objects in JavaScript. A simple example is as follow:

```
(new java.lang.String("test")).startsWith("t")
```

This is a valid JavaScript expression: as you can see, we are able to create a Java object (in this case a java.lang.String), and use its methods.

JavaScript is the best choice for people that have absolute no knowledge of other languages since it is easy to learn and there are plenty of JavaScript manuals and references on the net.

The only significant advantage of using Groovy is that it is not interpreted at run time, but it generates pure java byte-code, reaching almost the same performance of using Java.

## 3.9 A Simple Program

I finish this introduction to JasperReports by presenting an example of a simple program (**Figure 3-5**) that shows how to produce a PDF file from a jasper file using a special data source named `JREmptyDataSource`, a utility data source that provides zero or more records without fields. The file `test.jasper`, referenced in the example, is the compiled version of Listing ?-?.

**Figure 3-4** JasperTest.java

```
import net.sf.jasperreports.engine.*;
import net.sf.jasperreports.engine.export.*;
import java.util.*;

public class JasperTest
{
 public static void main(String[] args)
 {
  String fileName = "/devel/examples/test.jasper";
  String outFileName = "/devel/examples/test.pdf";
  HashMap hm = new HashMap();

  try
  {
   JasperPrint print = JasperFillManager.fillReport(
     xfileName,
     hm,
     new JREmptyDataSource());

   JRExporter exporter =
     new net.sf.jasperreports.engine.export.JRPdfExporter();

   exporter.setParameter(
     JRExporterParameter.OUTPUT_FILE_NAME,
     outFileName);
   exporter.setParameter(
   JRExporterParameter.JASPER_PRINT,print);

   exporter.exportReport();
   System.out.println("Created file: " + outFileName);
  }
  catch (JRException e)
  {
   e.printStackTrace();
   System.exit(1);
  }
  catch (Exception e)
  {
   e.printStackTrace();
   System.exit(1);
  }
 }
}
```

# 4 Report Structure

In this chapter we will analyze the report structure, the underlying template that determines the style and organization of a report. We will see which parts compose it and how they behave in relation to input data as iReport creates an output report.

## 4.1 Bands

A report is defined by means of a type page. This is divided into different horizontal portions named bands. When the report is joined with data to run the print, these sections are printed many times according to their function (and according to the rules

that the report author has set up). For instance, the page header is repeated at the beginning of every page, while the detail band is repeated for every single elaborated record.


**Figure 4-1 Pre-defined bands of a document**

The type page is divided into nine predefined bands to which new groups are added. In fact, iReport manages a heading band (Group header) and a recapitulation band (Group footer) for every group.

A band is always as wide as the usable page width (right and left margins excluded). However, it's height, even if it is established during the design phase, can vary during the print creation according to the contained elements; it can “lengthen” towards the bottom of page in an arbitrary way. This typically occurs when bands contain sub-reports or text fields that have to adapt to the content vertically. Generally, the height specified by the user should be considered “the minimal height” of the band. Not all bands can be reorganized dynamically according to the content, in particular the *Column Footer*, *Page Footer* and *Last Page Footer* bands.

The sum of all band heights (except for the background) has to always be less than or equal to the page height minus the top and bottom margins.

The following sections briefly outline each of the predefined bands.

### 4.1.1 Title

The *title* band is the first visible band. It is created only once and can be printed on a separate page. Regarding the defined dimensions, it is not possible during design time to exceed the report page height (top and bottom margins are included). If the title is printed on a separate page, this band height is not included in the calculation of the total sum of all band heights, which has to be less than or equal to the page height, as mentioned previously

### 4.1.2 Page Header

The *page header* band allows you to define a page header. The height specified during the design phase usually does not change during the creation process (except for the insertion of vertically re-sizable components, such as text fields that contain

long text and sub-reports). The page header appears on all printed pages in the same position defined during the design phase. Title and Summary bands do not include the page header when printed on a separate page.

### 4.1.3 Column Header

The *column header* band is printed at the beginning of each detail column. (The column concept will be explained in the "Columns" section later in this chapter.) Usually, labels containing the column names of the tabular report are inserted in this band.

### 4.1.4 Group Header

A report can contain zero or more group bands, which permit the collection of detail records in real groups. A *group header* is always accompanied by a *group footer* (both can independently be visible or not). Different properties are associated with a group. They determine its behavior from the graphic point of view. It is possible to always force a group header on a new page or in a new column and to print this band on all pages if the bands below it overflow the single page (as a page header, but at group level). It is possible to fix a minimum height required to print a group header: if it exceeds this height, the group header band will be printed on a new page (please note that a value too large for this property can create an infinite loop during the print). (I will discuss groups in greater detail later on in this chapter.)

### 4.1.5 Detail

A *detail* band corresponds to every record that is read by the data source that feeds the print. In all probability, most of the print elements will be put here.

### 4.1.6 Group Footer

The *group footer* band completes a group. Usually it contains fields to view subtotals or separation graphic elements, such as lines.

### 4.1.7 Column Footer

The *column footer* band appears at the end of every column. Its dimensions are not adjustable at run time (not even if it contained re-sizable elements such as sub-reports or text fields with a variable number of text lines).

### 4.1.8 Page Footer

The *page footer* band appears on every page where there is a page header. Like the column footer, it is not re-sizable at run time.

### 4.1.9 Last Page Footer

If you want to make the last page footer different from the other footers, it is possible to use the special *last page footer* band. If the band height is 0, it is completely ignored and the layout established for the common page will be used also for the last page. This band first appeared in JasperReports version 0.6.2.

### 4.1.10 Summary

The *summary* band allows you to insert fields concerning total calculations, means, or whatever you want to insert at the end of the report. In other systems, this band is often named "report footer".

### 4.1.11 Background

The *background* band appeared for the first time in JasperReports version 0.4.6. It was introduced after insistent requests from many users who wanted to be able to create watermarks and similar effects (such as a frame around the whole page). It can have a maximum height equal to the page height and his content will appear in all the pages without be influences by the page content defined in the other bands.

### 4.1.12 No Data

The *No Data* band is an optional report section that is printed only if the data source does not return any record and the report property *When no data type* must be set to “No Data section”. Since this band will be printed instead of all the other bands, his height can have the same size of the report page (excluded margins).

### 4.1.13 Report Properties

Now that you have seen the individual parts that comprise a report, you will proceed to creating a new one. Select *New Empty Report* from the *File* menu, choose a name for the document and press the button *Finish*. A new empty report will appear in the main design window.

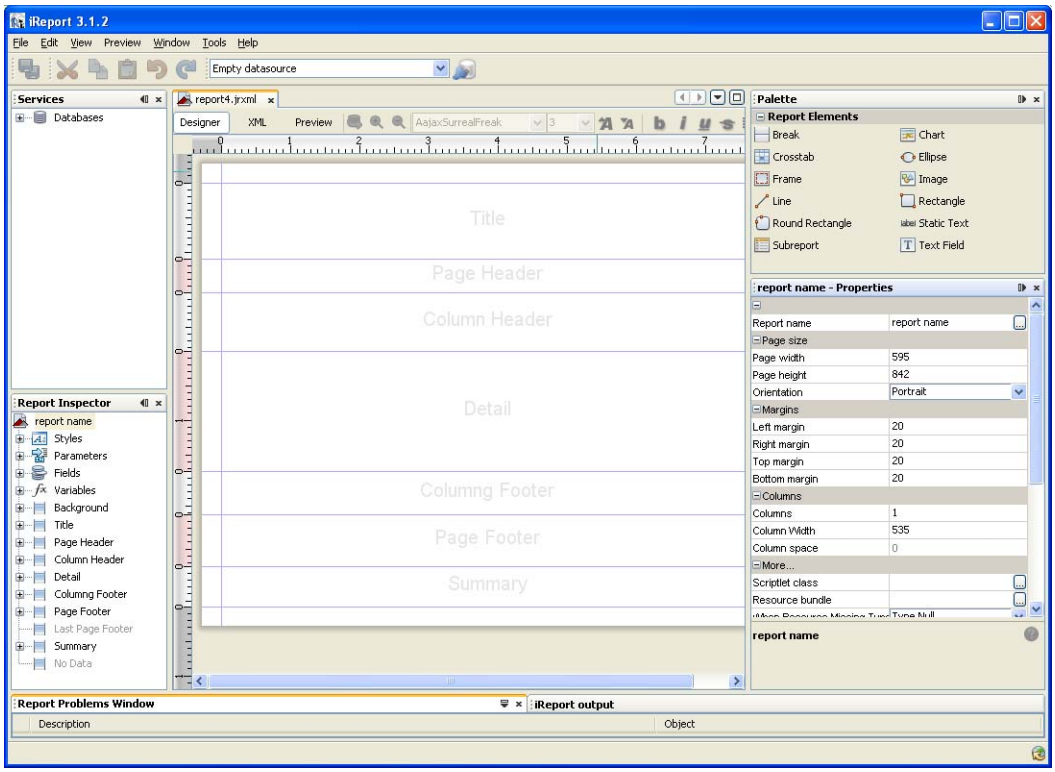



**Figure 4-2 A new empty report**

The properties view (on the right side of the main window) shows the properties of the object that is currently selected in the Report Inspector view (on the left side of the main window) or in the design area (like a band or an element). When a new report is created, the property sheet displays the report properties. You can recall the report properties at any time by selecting the root node in the Report Inspector (showing the report name) or by clicking with the mouse any area outside the document in the design window.

The first property is the report name. It is a logical name, independent from the source file’s name, and is used only by the JasperReports library (e.g., as base name for the temporary Java file produced when a report is compiled).

The page dimensions are probably the report’s most important properties. The unit of measurement used by iReport and JasperReports is the pixel (which has a resolution of 75 dpi, or dots per inch).

Table 4-3 lists some standard page formats and their dimensions in pixels.

**Table 4-3 Standard Print Formats**

| Page type | Dimensions in pixels | | Page type | Dimensions in pixels | |
|---|---|---|---|---|---|
| | Width | Height | | Width | Height |
| LETTER | 612 | 792 | B1 | 2004 | 2836 |
| NOTE | 720 | 540 | B2 | 1418 | 2004 |
| LEGAL | 1008 | 612 | B3 | 1002 | 1418 |
| A0 | 3368 | 2380 | B4 | 709 | 1002 |
| A1 | 2380 | 1684 | B5 | 501 | 709 |
| A2 | 1684 | 1190 | ARCH_E | 2592 | 3456 |
| A3 | 1190 | 842 | ARCH_D | 1728 | 2592 |
| A4 | 842 | 595 | ARCH_C | 1296 | 1728 |
| A5 | 595 | 421 | ARCH_B | 864 | 1296 |
| A6 | 421 | 297 | ARCH_A | 648 | 864 |
| A7 | 297 | 210 | FLSA | 612 | 936 |
| A8 | 210 | 148 | FLSE | 612 | 936 |
| A9 | 148 | 105 | HALFLETTER | 396 | 612 |
| A10 | 105 | 74 | _11X17 | 792 | 1224 |
| B0 | 4008 | 2836 | LEDGER | 1224 | 792 |

By modifying width and height, it is possible to create a report of whatever size you like. The page orientation option, *landscape* or *portrait*, in reality is not meaningful, because the page dimensions are characterized by width and height, independently from the sheet orientation. However, this property can be used by certain report exporters to decide how to print the report using a printer.

The page margin dimensions are set by means of the four entries on the *Margins* section.

### 4.1.14 Columns

As we have seen, a report is divided into horizontal sections: bands.

The page, which composes the report, presents portions which are independent from the records that coming from the data source (such as the title section, or the page footers), and other sections that are driven by that records (such as the group headers/footers and the detail). These last portions can be divided into vertical columns in order to optimize the available space on the page.

In this context the concept of “column” can be easily confused with that of “field”. A column is not connected to a record field, we are just defining here the layout of the page, not a table or something tied to the format of the data to print. This means that if you want to print records having for instance ten fields, and you want to create a report that looks like a table, you don't need ten report columns, but you'll have to place the report elements (labels and text fields) in a single column report in order to get a table effect

You use columns when you need a layout similar to the one of the newspapers, where the text rows are presented on several columns to improve readability and better use the space on the page.

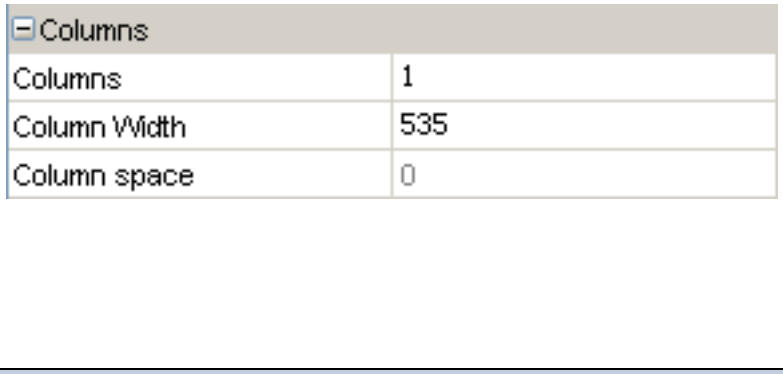



**Figure 4-4 Report objects in the outline view**

In the following figures we present two examples. The first shows how to set up the report to use a single column (actually the default and most common configuration; in this particular case the size of the page is a regular A4).

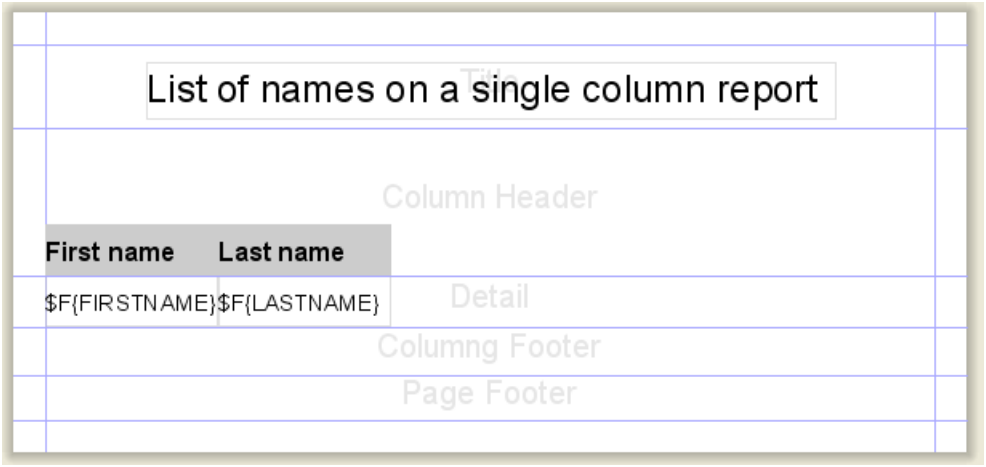



**Figure 4-5 Layout of a single column report showing a set of names**

The values are set in the report properties view. The number of columns is 1 and the width is equal to the entire page with, except for the margins (that's 535 pixels). Since there is just a single column, the space between columns is not meaningful and it is set to zero (that property is actually disabled when the column number is 1).

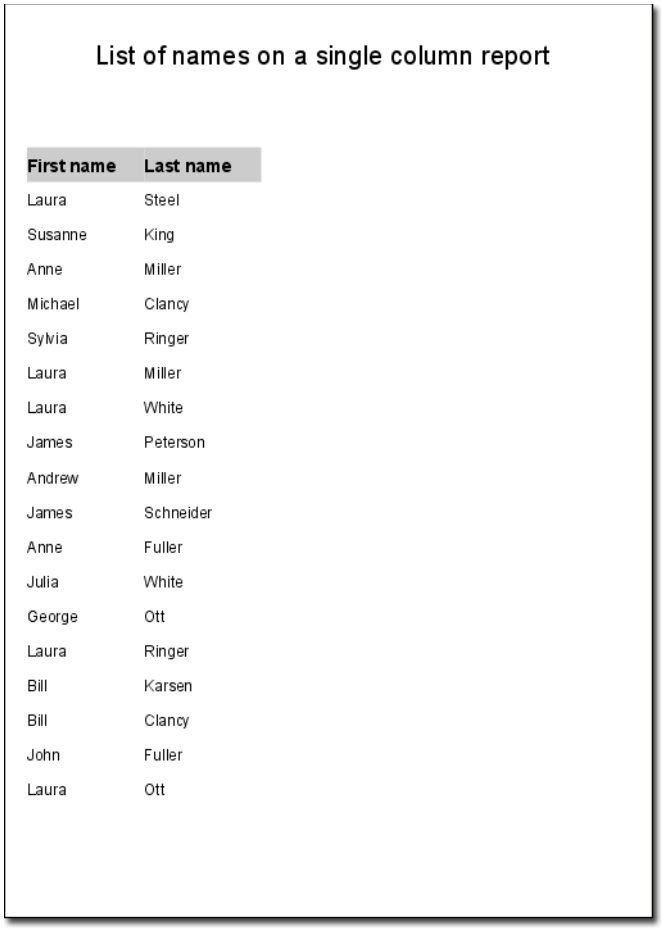



**Figure 4-6 Result of a report using the single column layout**

As you can see in **Figure 4-6**, most of the page is not used (the figure shows only the first page, but the report is composed of other pages that look very similar); in facts each record takes the whole horizontal width of the page. So the idea here is try to split the pages in two columns, so that when the first column reach the end of the page, we can start to print in this page again in the second column. **Figure 4-7** shows the dimensions used for a two columns report.

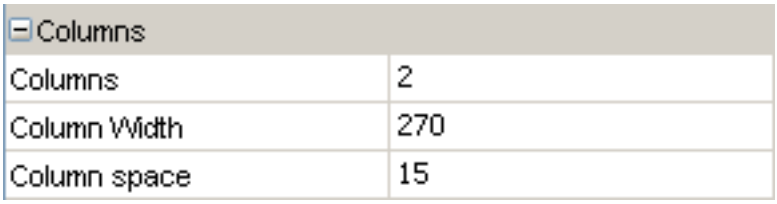



**Figure 4-7 Settings for a two columns report**

In this case the columns number property is set to 2. iReport will automatically calculate the maximum column width according to the margins and to the page width. If you want to increase the space between the columns, just increase the value of the Column space property.

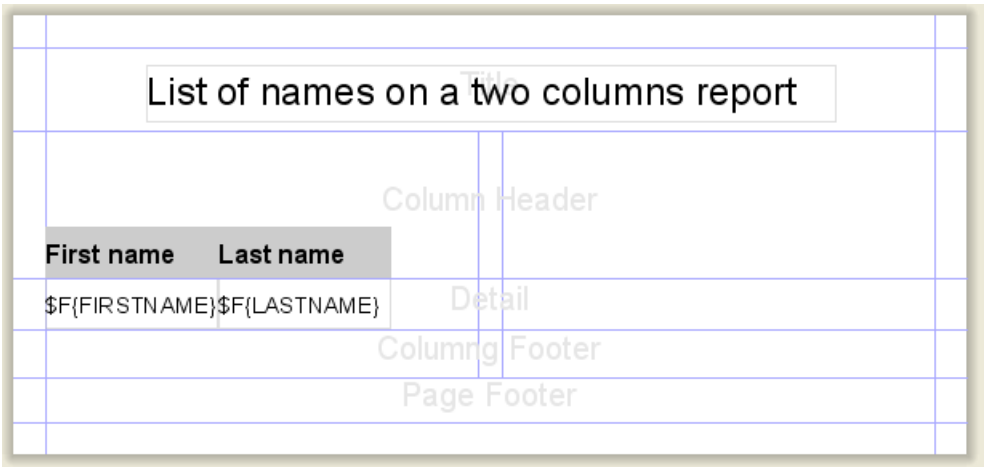
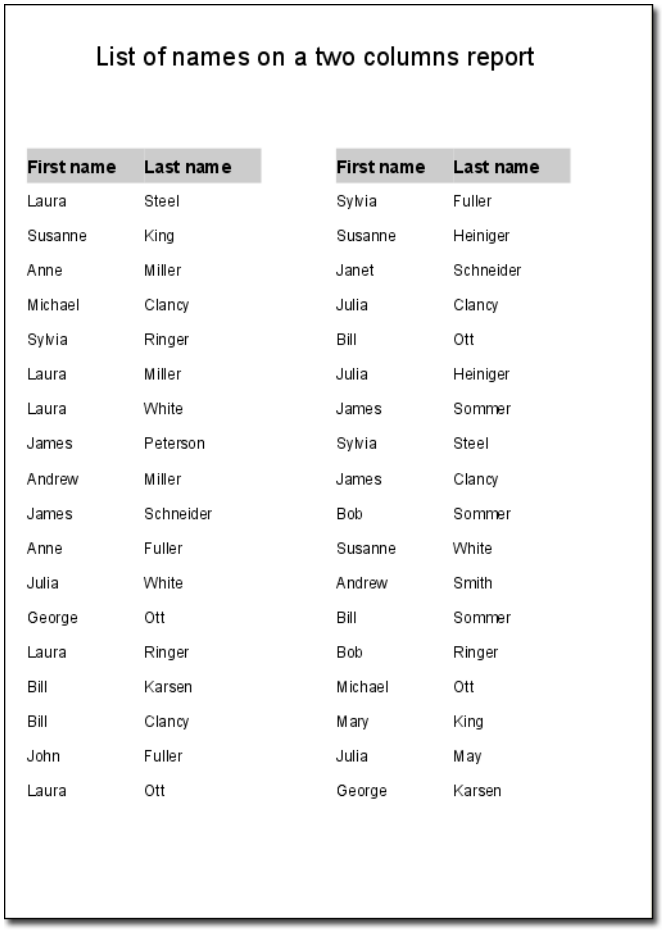



**Figure 4-8 Layout of a two columns report showing a set of names**

The designer will show the column bounds and the space between the columns..



List of names on a two columns report

| First name | Last name | First name | Last name |
|---|---|---|---|
| Laura | Steel | Sylvia | Fuller |
| Susanne | King | Susanne | Heiniger |
| Anne | Miller | Janet | Schneider |
| Michael | Clancy | Julia | Clancy |
| Sylvia | Ringer | Bill | Ott |
| Laura | Miller | Julia | Heiniger |
| Laura | White | James | Sommer |
| James | Peterson | Sylvia | Steel |
| Andrew | Miller | James | Clancy |
| James | Schneider | Bob | Sommer |
| Anne | Fuller | Susanne | White |
| Julia | White | Andrew | Smith |
| George | Ott | Bill | Sommer |
| Laura | Ringer | Bob | Ringer |
| Bill | Karsen | Michael | Ott |
| Bill | Clancy | Mary | King |
| John | Fuller | Julia | May |
| Laura | Ott | George | Karsen |

**Figure 4-9 Result of a report using the two column layout**

As we see in **Figure 4-9**, the page space is now better used.

Multiple columns are commonly used for prints of very long lists (for example the phone book). The sum of the margins, column widths and every space between columns, has to be less than or equal to the page width. If this condition is not verified the compilation can result in error.

When working with more that a column, you should put elements (fields, images, etc…) just inside the first column. The other columns are displayed in the designer just for reference, but any element placed here at design time would be treated as part of the first column (in facts you are just defining a detail template, there are no restrictions about placing elements outside the horizontal band's bounds, but it would be like putting elements outside the page).

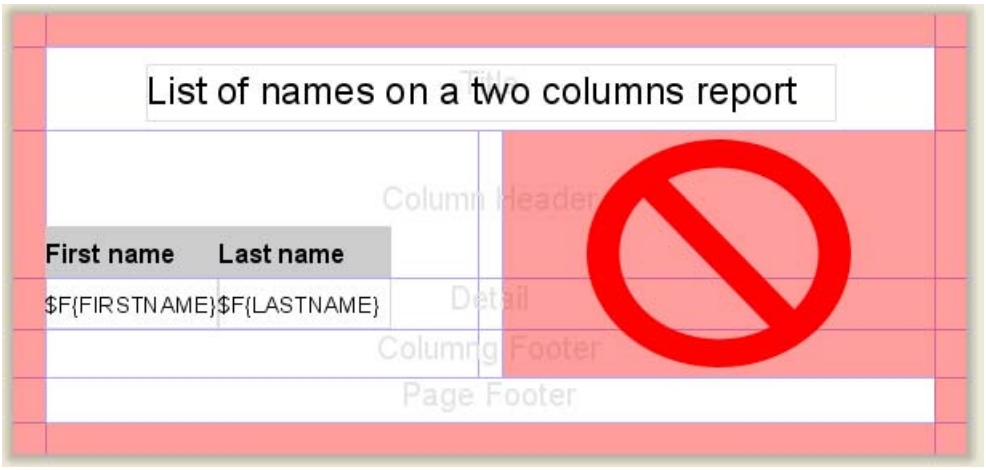



**Figure 4-10 Safe area to place report elements (text fields, images, etc...)**

The following picture shows the “unsafe” areas: they are essentially the margins and all what stays on the right of the first column.

Of course, the same rules about where to place elements are applied to the report even if there is a single column.

## 4.1.15 Advanced Report Options

Up to now we have seen only basic characteristics concerning the layout. Now we will see some advanced options. Some of them will be examined thoroughly and explained in every detail in the following chapters, and some of them will be fully understood and used in a useful way only after having acquired familiarity with the use of JasperReports.

### 4.1.15.1 Scriptlet

A *scriptlet* is a Java class whose methods are executed according to specific events during report creation, such as the beginning of a new page or the end of a group. For those who are familiar with visual tools such as MS Access or MS Excel, a scriptlet can be compared with a *module*, in which some procedures associated with particular events or functions recallable in other report contexts (for example, the expression of a text field) are inserted. I discuss scriptlets at length in **Subdatasets**.

### 4.1.15.2 Resource Bundle

The *Resource Bundle* is a property used when you want to internationalize a report. A Resource Bundle is the set of files that contain the translated text of the labels, sentences, and expressions used within a report in each defined language. Each language corresponds to a specific file. What you set in the resource bundle property is the *resource bundle base name* that's the prefix through which you can find the file with the correct translation. In order to reconstruct the file name required for a particular language, some language/country initials (e.g., _it_IT for Italian-Italy) and the `.properties` extension are added to this prefix. (I will explain internationalization in greater detail in Chapter **14, “Internationalization,” on page 231**.)

If a resource is not available, you can specify what to do by choosing an option from property labeled *When resource missing type*. The available options are listed in Table 2:

| Option | Description |
|---|---|
| `Null` | It prints the “Null” string (it’s the default option) |
| `Empty` | It prints nothing |
| `AllSectionsNoDetails` | It prints the missing key name |
| `Error` | It throws an exception stopping the fill process |

**Figure 4-11 Table 2: Options for the When resource missing type**

#### 4.1.15.3 Query

The *Query* property is used to set a query to select data. The language of the query is set through the property labeled The language for the dataset query. Although the query and his language are presented in the property sheet, it is much more convenient edit them using the query editor accessible trough the dedicated tool bar button  .

#### 4.1.15.4 Filter Expression

The *filter expression* is another property that can be edited from the query editor. It is a boolean expression that can use as usual all the objects of the report (parameters, variables and fields) to decide if the current records read from the data source should be used or not.

Here are some examples of filter expressions.

Filter only records where the field FIRSTNAME starts with the letter “L”

- *JavaScript*`$F{FIRSTNAME}.substr(0,1) == "L"`
- *Groovy*`$F{FIRSTNAME}.startsWith("L")`

Filter only records where the field FIRSTNAME length is less than 5

- *JavaScript*`$F{FIRSTNAME}.length < 5`
- *Groovy*`$F{FIRSTNAME}.length() < 5`

Filter only records where the field `FIRSTNAME` is the one provided by the parameter `NAME`

- *JavaScript*`$F{FIRSTNAME} == $P{NAME}`
- *Groovy*`$F{FIRSTNAME} == $P{NAME}`

#### 4.1.15.5 Properties

It is possible to define a set of name/value pairs to a report. These pairs is what we call report properties. The name and the value of the properties are just simple strings and they are used for a lot of purposes, including driving special exporter

features, override JasperReports default values and so on. We will see that the same kind of properties can be set for report elements too..

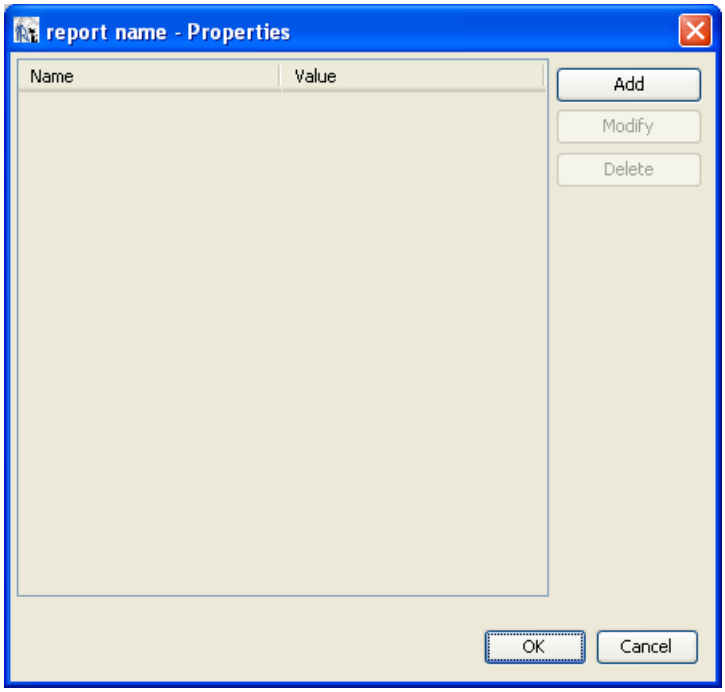



**Figure 4-12 Properties Dialog**

When editing the properties, the dialog in **Figure 4-12** pops up..

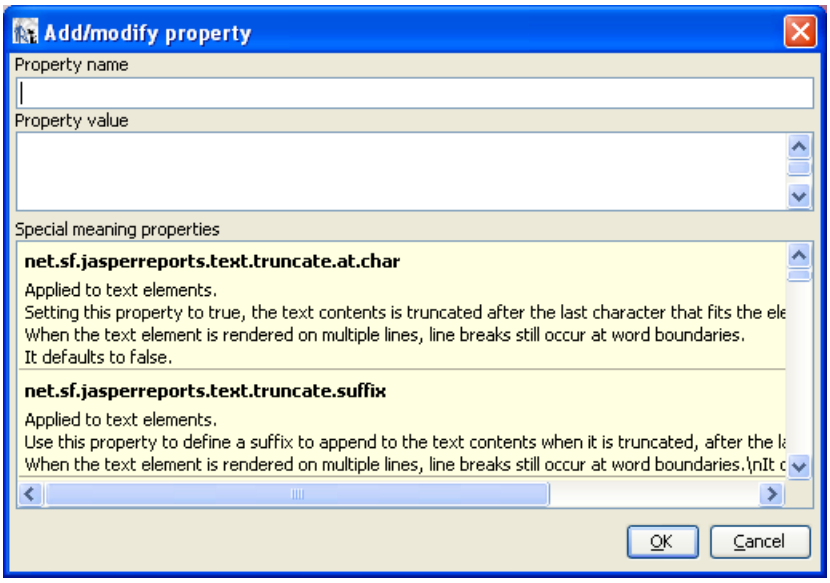



**Figure 4-13 Report Properties – Add Property**

Press the button “Add” to create a new property. A new window will open (**Figure 4-13**).

The dialog allows you to specify a property name and the value.

In the lower part of the window there is a list of special meaning properties. You can double click an item to set the property name field with the one specified by the item.

The list is not  exhaustive, but it contains the most important property names with a special meaning understood by JasperReports. If you scroll the list, you'll notice that these special properties can be used for a lot of different tasks like specifying particular attributes when the report is exported in a specific format (i.e. avoid pagination when exporting in XSL), activate special exporter directives (i.e. to encrypt the file when exported in PDF), or even specify a particular theme to be used with the charts in the document.

#### 4.1.15.6 Title and Summary on a new page option.

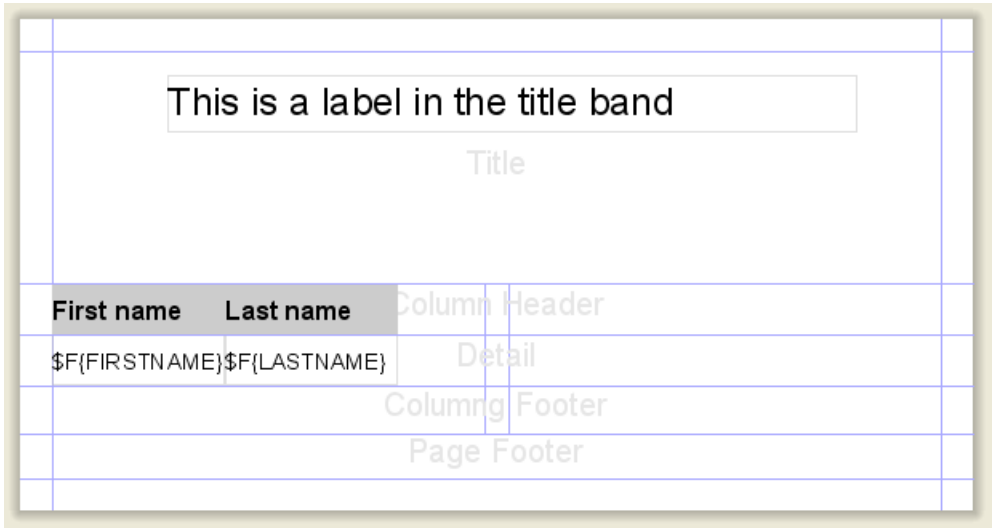



**Figure 4-14 Title Band**

The *Title on a new page* option specifies that the title band is to be printed on a new page, which forces a page break at the end of the title band. By default this option is not activated. As an example, take a look at **Figure 4-14**, which shows a simple report.

Changing the option does not affect the design window: in the editor the title band is always drawn on top of the others (except, when present, the background).

When the report is run the title band will go on a separate page based to this option value..

**Figure 4-15** and **Figure 4-16** show the same report, the first printed without setting the title on a new page option, the second setting it to `true`.

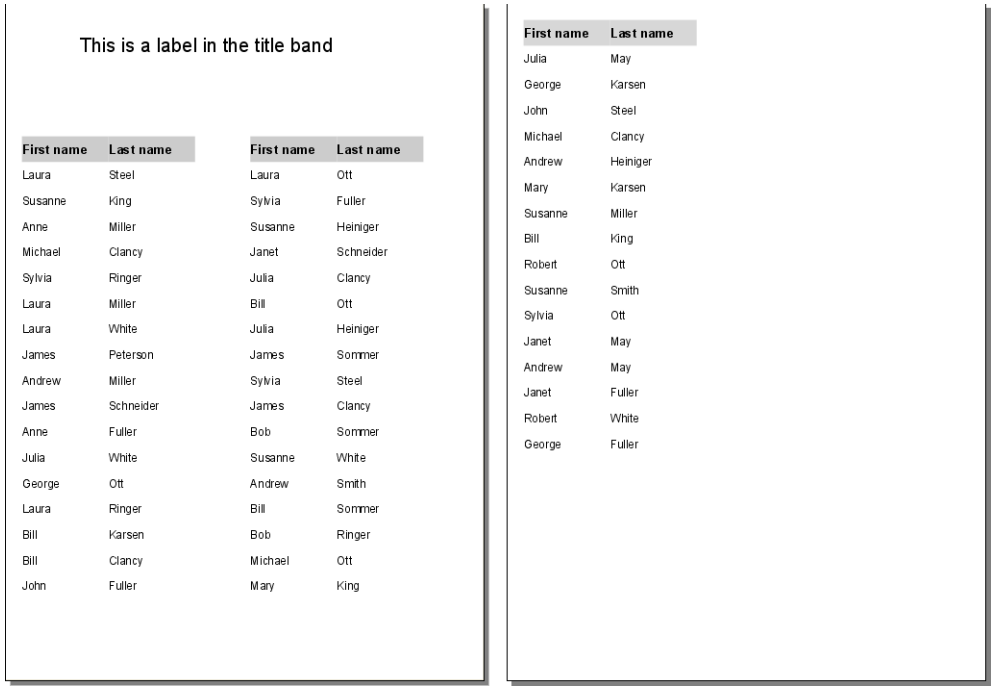



**Figure 4-15 Default printing of the title band**

As you can see in **Figure 4-16**, when the title on a new page option is activated, no one other band is printed on the title page, not even the *page header* or *page footer*. However, this page is still counted in the total pages numeration..

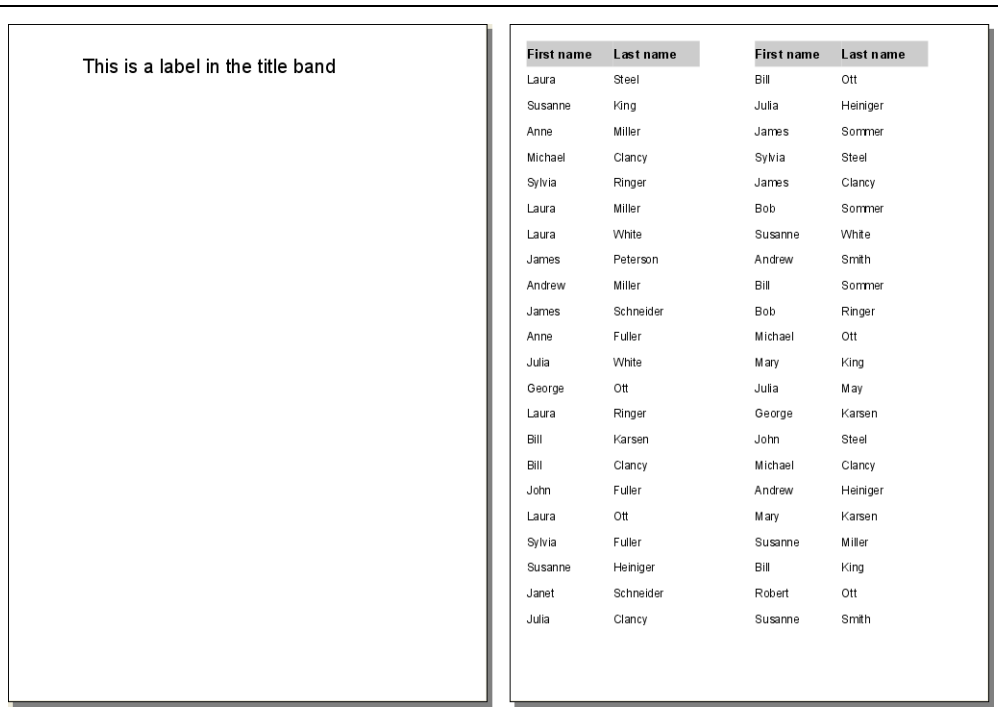



**Figure 4-16 Title band on a new page**

This option is available for the *Summary* band too, the difference is that the summary band is printed as the last page. Now, if you need to print this band on a new page, the new page will only contain the summary band.

#### 4.1.15.7 Floating Column Footer Option

This option allows you to force the printing of the column footer band immediately after the last detail band (or group footer) and not at the end of the column. This option is used, for example, when you want to create tables using the report elements (see the JasperReports `tables.jrxml` example for more details).

#### 4.1.15.8 Print Order

The *Print order* option determines how the print data is organized on the page when using multiple columns. The default setting is Vertical, that is, the records are printed one after the other, passing to a new column only when the previous column has reached the end of the page (like what happens in a newspaper or a phone book)..

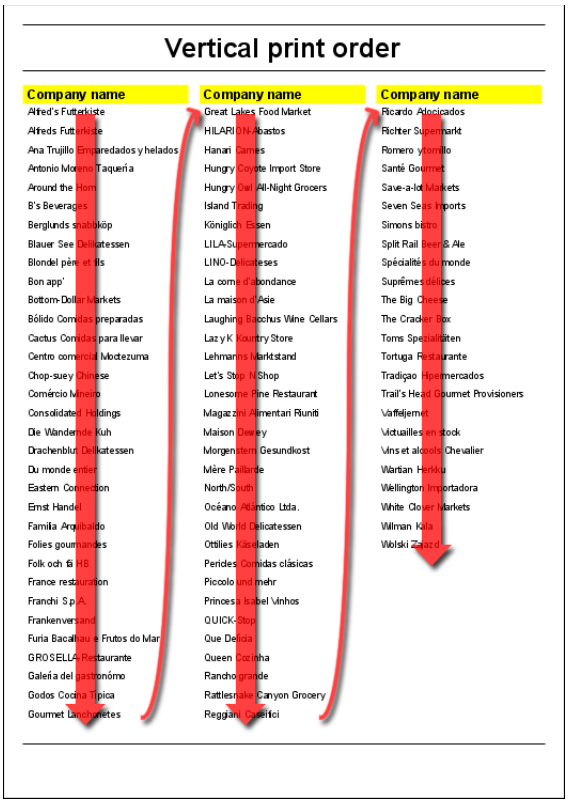



**Figure 4-17 Vertical Print Order**


**Figure 4-18 Horizontal Print Order**

Horizontal print order prints the different records horizontally across the page, occupying all of the available columns before passing to a new line. Refer to **Figure 4-17** and **Figure 4-18** for examples of vertical and horizontal print order.

The prints in these two figures should clarify the concept of print order. As you can see, the names are printed in alphabetical order. In **Figure 4-17**, they are printed in vertical order (filling in the first column and then passing to the following column), and in **Figure 4-18**, they are printed in horizontal order (filling all columns horizontally before passing to the following line).

#### 4.1.15.9 Print without data (when no data)

When an empty data set is supplied as the print number (or the SQL query associated to the report gives back no records), an empty file is created (or a stream of zero byte length is given back). This default behavior can be modified by specifying what to do in the case of absence of data (that is, when no data). Table 3 summarizes the possible values and their meaning.

| Option | Description |
|---|---|
| *NoPages* | This is the default; the final result is an empty buffer |
| *BlankPage* | This gives back an empty page |
| *AllSectionsNoDetails* | This gives back a page composed of all the bands except for the detail band |
| *No Data section* | Print the No Data band |

**Figure 4-19 Options for the When no data property**

#### 4.1.15.10 Format Factory Class

A `format factory` class is a class that implements the interface `net.sf.jasperreports.engine.util.FormatFactory`. You can set a custom implementation of that class, which will be used to define the default format template for numbers and dates.

## 4.2 Working with Bands.

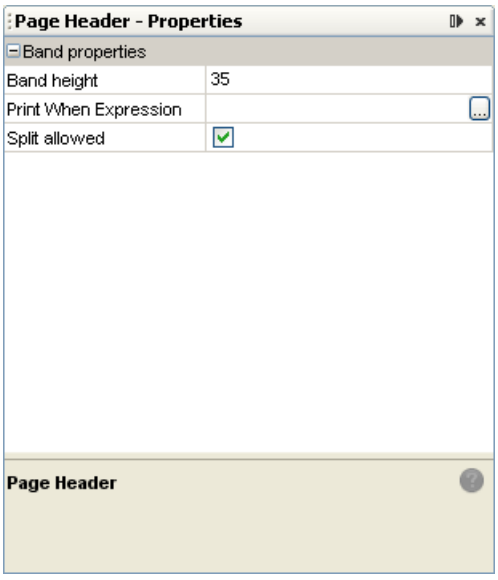



**Figure 4-20 Band Properties**

When creating a new empty report, the default template puts at your disposal a set of predefined bands (background, title, page header, column header, detail, column footer, page footer and summary). In particular the height of the background band in the template is zero, so you actually don't see it in the designer. In the Report Inspector you can see all the available bands, the ones displayed in gray (Last Page Footer and No Data) are not present in the report, this means that if you want to use them, you need to explicitly add them to the report clicking the band node, right click and select the menu item *Add Band.* In the same way you add a band not present in the report, you can remove one. Another options is to set the height of an unwanted band to 0. There is only a case where this would not work: with the *Last Page Footer* band. In effects this band automatically replaces the *Page Footer* band in the last page of the report when it is defined, so to avoid that you need to remove it if you don't want it and this is why it is not present in the template.

The properties of a band can be modified selecting the band node or just clicking with the mouse in a free area of the band in the main designer (so not over an element or outside the band bounds).

## Band height

The *Band height* is the design height of the band. As explained earlier in this chapter, the band height can change during the filling process. The height of a band in general does not get lower of the specified value, even if this is possible when the *Remove Line When Blank* option is set in one or more elements in that band and all the conditions to remove the horizontal space taken by these elements at filling time are verified (the *Remove Line When Blank* option is explained in the next chapter).When this property is modified, iReport checks whether the value set is acceptable (calculating the available space in the page and taking in consideration options like *Title on a new page* and *Summary on a new page*. If the value set does not fit the requirements, iReport suggests the possible value range.

## Print When Expression

The *Print When Expression* is a Boolean expression (so it must return true or false) that can be used to hide the band and prevent it from being included in the output report. The expression is evaluated every time the band referenced in a report. So, for example, in a report page the title band is evaluated only once, for the page header it is evaluated every time a new page is produced, for the detail band it is evaluated all the times a new record is processed.

Like in all the expression, you are free to use all the report objects available (fields, parameters and variables).

## Split allowed

The *Split allowed* option is used to specify what to do if the band does not fit the remaining available space in the page. Keeping in mind that the bands can grown dynamically during the filling process, it's easy to reach the situation for which there is not enough space to print the current band. Usually JasperReports splits the band printing in the current page only what fits there, and the rest in the following pages. However it can be there cases where this approach is not good for us, and we would like to keep together the whole content of the band. To do that, just disable the *Split allowed* condition. JasperReports will check if there is enough space in the page to print the whole band otherwise it will print it on a new page. At this point the split allowed condition is no longer considered.  This to avoid an infinite loop condition, where JasperReports keeps skipping pages in order to find a break point where the band can fit, a condition that could be never satisfied.

In the next chapter we will see how to use the group header and the group footer bands, and what other options can be set to place band groups in a new column or on a new page. At this point you should understand the structure of a page and how it is divided into several bands (or sections). You should also understand the conditional nature of bands, and how iReport evaluates whether and how to include a band in a report page. In the bands we will put the content to print.

# 5 REPORT ELEMENTS

In this chapter, I will explain the main objects that can be inserted into a report and discuss their characteristics.

By “element,” I mean graphical object, such as a text string or a rectangle. Unlike what happens in a word processing program, in iReport the concept of line or paragraph does not exist; everything is created by means of elements, which can contain text, create tables when they are opportunely aligned, and so on. This approach follows the model used by the majority of report authoring tools.

Nine basic elements are offered by the JasperReports library:

- Line
- Rectangle
- Ellipse
- Static text
- Text field (or simply Field)
- Image
- Frame
- Sub-report
- Crosstab
- Chart
- Break

Through a combination of these elements, it is possible to produce every kind of report.

In addition to the nine basic elements, there is a special element to create manual breaks (column or page break). Finally, JasperReports allows developers to implement their own *generic elements* and *custom components* for which is possible add the support in iReport to create a proper plug-in.

Each kind of element has some common properties, such as height, width, position, and the band to which it belongs. Other properties are specific to the type of element (for example, font or, in the case of a rectangle, thickness of the border). There are several types of elements; the graphic elements are used to create shapes and display images (they are line, rectangle, ellipse, image), the text elements are used to print text strings such as labels or fields (they are static text and textfield), the frame element is used to group a set of elements and optionally draw a border around them. Sub-reports, Charts and Crosstabs are more complex elements, so I will touch briefly on them later in the this chapter, and discuss them in more detail in separate chapters. Finally there is a special element used to insert a page or column break.

The elements are inserted into bands. In particular, every element is associated indissolubly to a band. If an element is not completely contained within the band that it is part of, the report compiler will return a message that informs you about the wrong position of the element; the report will be compiled despite such an error, and in the worst case, the “out-of-band” element will not be printed.

## 5.1 Working with Elements

The elements are presented in a palette, usually located in the top right portion of the main window (see Figure 5-1).

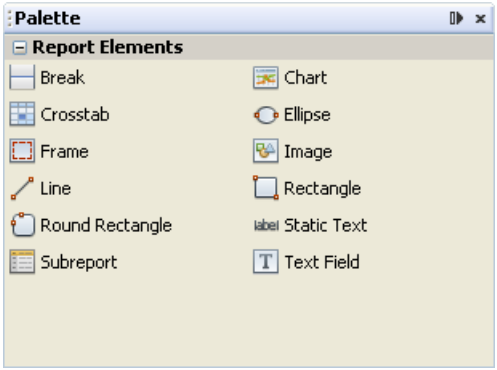



**Figure 5-1 Tools Palette**

To insert an element into the report, drag it from the palette into a report band. The new element will be created with a standard size and will appear in the report inspector. To select the element just click on it in the designer, or click the relative node in the report inspector. You can adjust the element position by selecting and dragging it; to modify his size drag a corner of the orange selection frame.

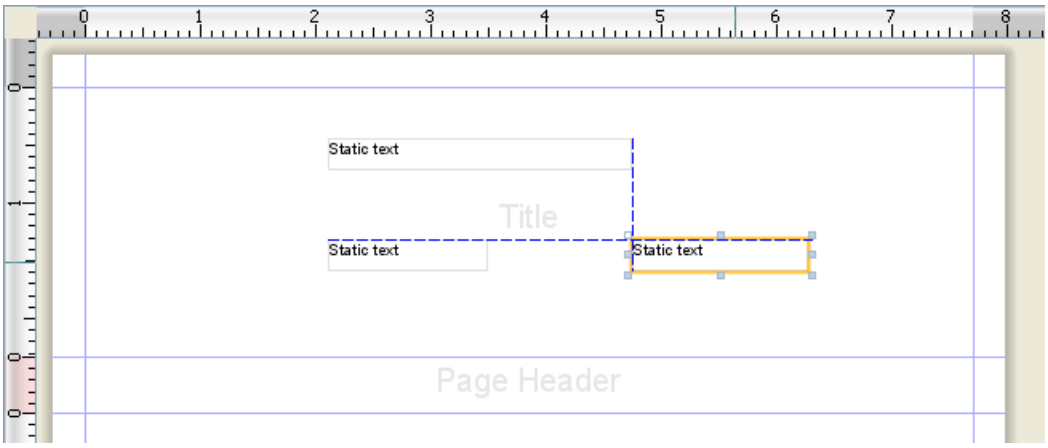



**Figure 5-2 Suggested alignment with other elements**

When dragging or resizing an element, iReport suggests places to align it based on the elements already in the design pane, band bounds and (if present) guide lines.

To obtain greater precision in the movement, use the arrows keys to move the element 1 pixel at a time; similarly, using the arrow keys while pressing the Shift key will move the element 10 pixels.

If you need a reference to position elements in the page, you can turn on the grid in the design pane by selecting the menu items **View → Report Designer → Show Grid**. To force the elements to snap to the grid, select **View → Report Designer → Snap to Grid**.

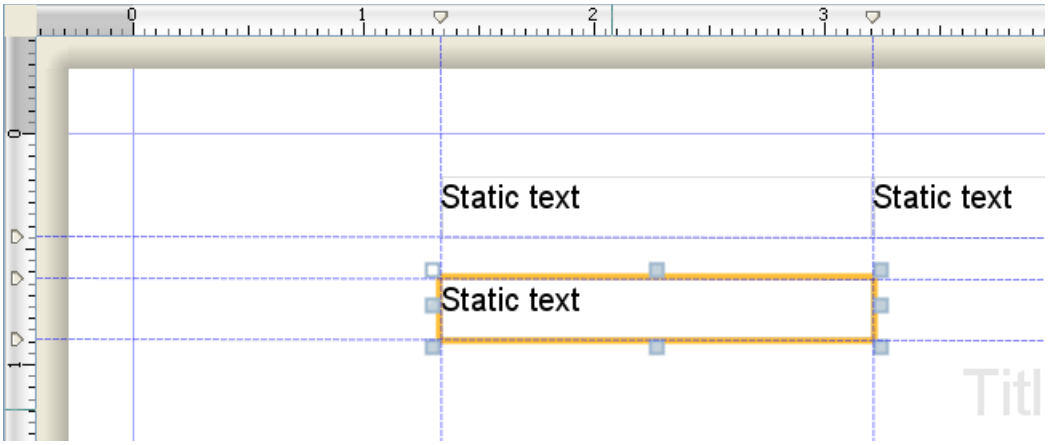



**Figure 5-3 Guide Lines**

Guide lines are also useful to position your elements in the report. With the guide line's magnetic effect, it is easy to place the elements in the right position. To create a guide line, just click on a ruler (vertical or horizontal) and drag the guide line in the wanted position (see **Figure 5-3**). By default rulers use inches as unit. In the options panel (**Tools > Options**) you can set the a different unit (like pixels, centimeters and millimeters).

You can drag and change the position of a guide line at any time, this will not have any effect on the elements position.

To remove a guide line, just drag it into to the top/left corner of the design pane.

The *top* and *left* values that define the element's position are always relative to the parent band, or better to the parent *container*, that's usually a band, but it could be a *frame* element.

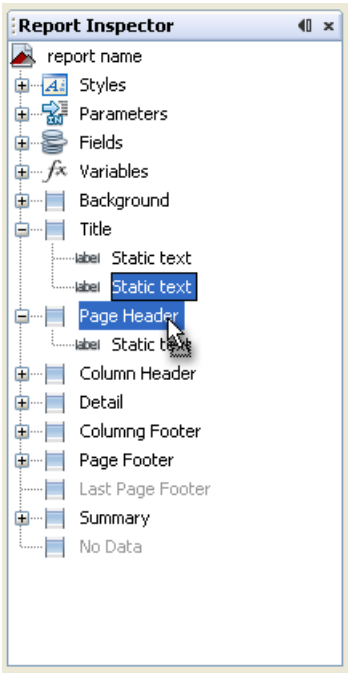



**Figure 5-4 Moving an element from a band to another**

If you want to move an element from its initial band to another one, or into a frame and vice versa, you need to use the report inspector and drag the element node in the new band (or frame) node as shown in **Figure 5-4**.

In the designer window, you can drag an element from a band to another one, but the element parent band will not change: we said that an element must be contained in his band, well, this is not always true.There are several exceptions to this rule and this is why iReport allows you to move an element anywhere in the report without preventing you from doing that and without changing or updating the parent band according to the new element position.

As general rule, you cannot position an element under the bottom of its parent band (even partially). If this happens, a design error will be displayed in the report problems view and the report will not work. In **Figure 5-5** we have a text element which has the Title as parent band. Since the element height spans over the Page Header band (that follows the Title band), a warning about the invalid element position appears in the report problems view.

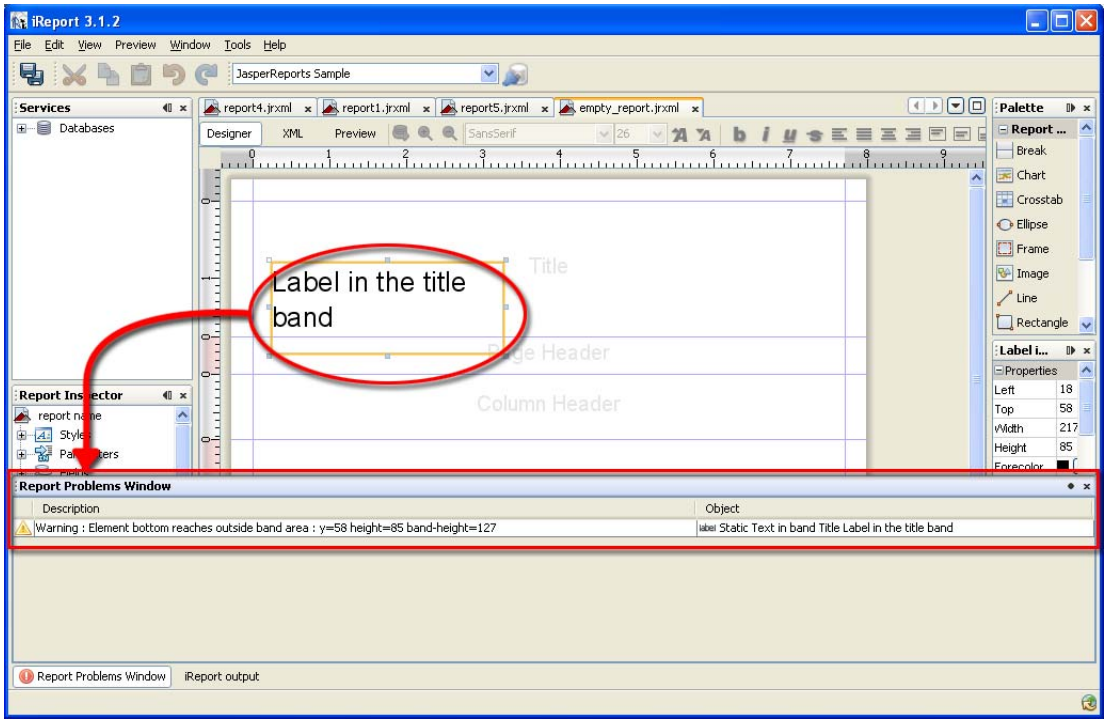



**Figure 5-5 Element not correctly positioned**

To edit the element properties, you can use the property sheet usually located on the right side of the designer window. The property sheet is not used only for elements, it can be used to edit the properties of all the components that make up the report,

like the page format, the band options, parameters, variables and fields options, etc. When something is selected in the designer or in the Report Inspector view, the property sheet shows the proper options for the selected object.

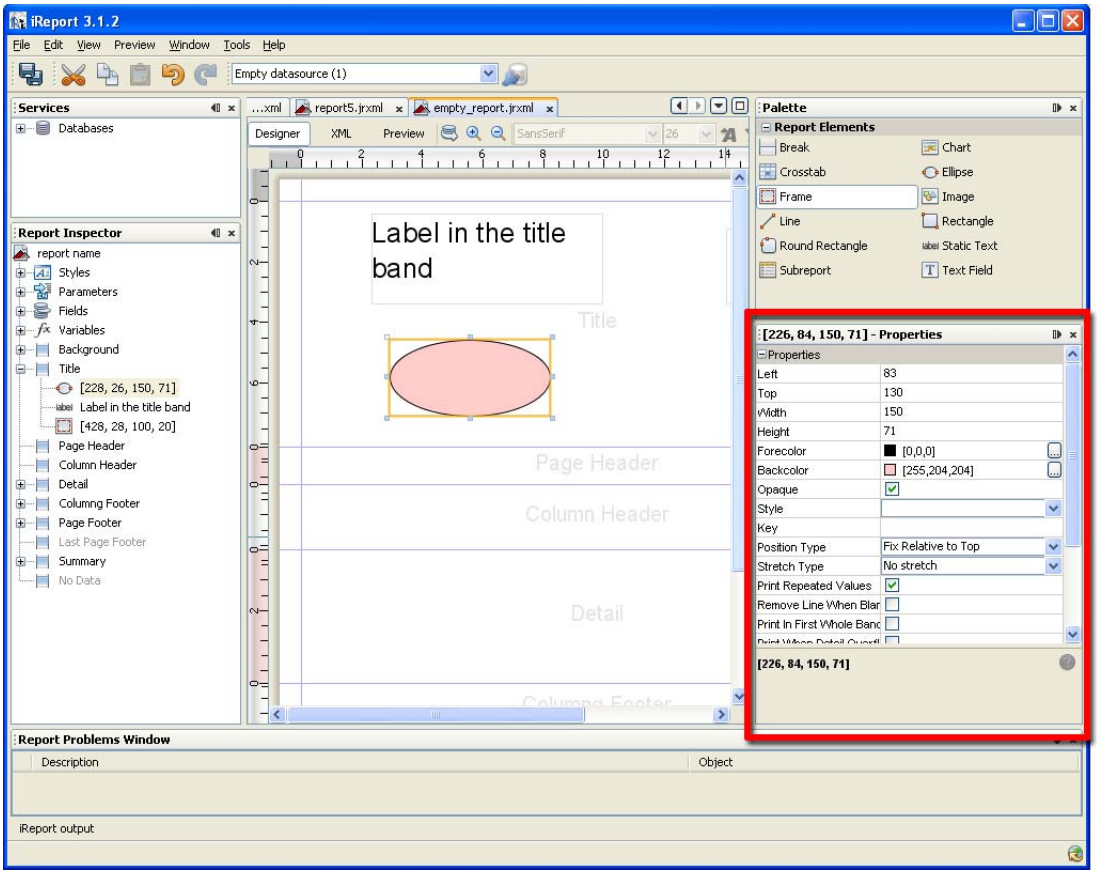



**Figure 5-6 Element not correctly positioned**

It is possible to select more elements at the same time by using the arrow tool and drawing a rectangle which will contain the elements to select. Depending on the direction on the rectangle is drawn, elements can be selected only if fully contained in the selected area, or even if partially selected.

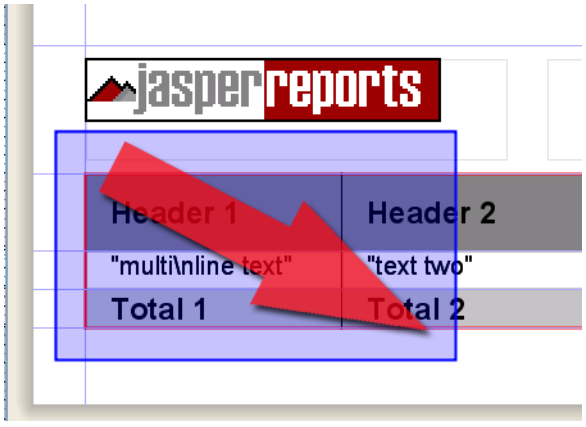



**Figure 5-7 Selection right to left**

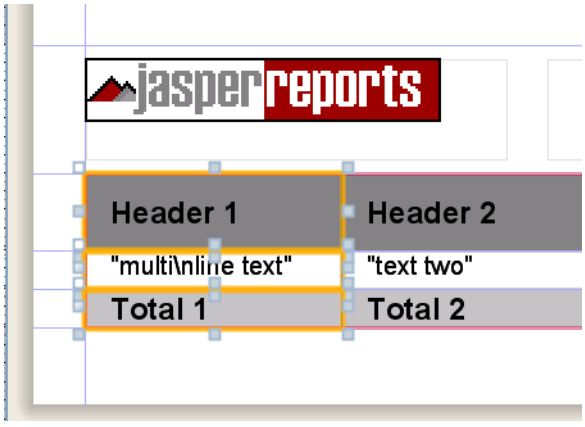



**Figure 5-8 Only elements fully contained in the selected area are selected**

Alternatively, it is possible select more than one element at the same time keeping pressed the "*Shift*" key and clicking with the mouse over all interested elements.

If two or more elements are selected, only the common properties are visualized. If the values of these properties are different, they will appear blank (usually the field is shown empty). Specifying a value for a particular property applies that value to all selected elements.

## 5.2 Formatting Tools

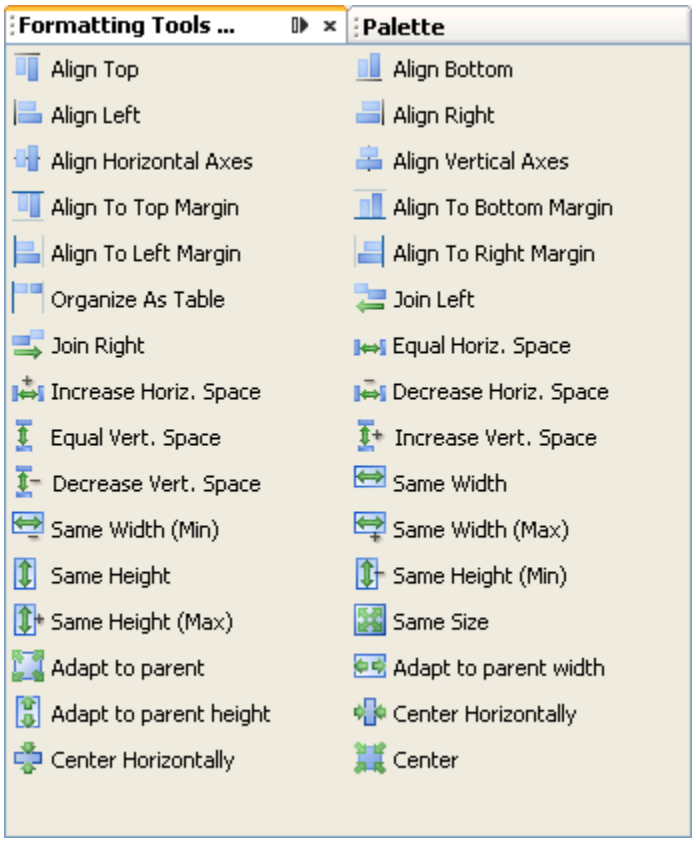



**Figure 5-9 Formatting Tools View**

To better organize the elements in the designer window, a comprehensive set of tools is provided. To access the formatting tools view, select the menu **Window > Formatting tools**. The tools view will appear. Each tool is enabled only when the selection match its minimum requirements (single or multiple selection).

Figure 5-10 lists all the available tools, specifying what kind of selection each tool requires (single or multiple selection) and briefly explaining what each tool does.

| | Operation | Description | Sel. mult. |
|---|---|---|---|
| | Align left | It aligns the left sides to that of the primary element. | √ |
| | Align right | It aligns the right sides to that of the primary element. | √ |
| | Align top | It aligns the top sides (or the upper part) to that of the primary element. | √ |
| | Align bottom | It aligns the bottom sides to that of the primary element. | √ |

| | | | |
|---|---|---|---|
|  | Align vertical axis | It centers horizontally the selected elements according to the primary element. | √ |
|  | Align horizontal axis | It centers vertically the selected elements according to the primary element. | √ |
|  | Align to band top | It sets the top value at 0. | |
|  | Align to band bottom | It puts the elements in the position at the bottom as much as possible according to the band to which it belongs. | |
|  | Same width | It sets the selected elements width equal to that of the primary element. | √ |
|  | Same width (max) | It sets the selected elements width equal to that of the widest element. | √ |
|  | Same width (min) | It sets the selected elements width equal to that of the most narrow element. | √ |
|  | Same height | It sets the selected elements height equal to that of the primary element. | √ |
|  | Same height (max) | It sets the selected elements height equal to that of the highest element. | √ |
|  | Same height (min) | It sets the selected elements height equal to that of the lowest element. | √ |
|  | Same size | It sets the selected elements dimension to that of the primary element | √ |
|  | Center horizontally (band based) | It puts horizontally the selected elements in the center of the band | |
|  | Center vertically (band based) | It puts vertically the selected elements in the center of the band | |
|  | Center in band | It puts the elements in the center of the band | |
| | Center in background | It puts the elements in the center of the page in the background | |
|  | Join sides left | It joins horizontally the elements by moving them towards left | √ |
|  | Join sides right | It joins horizontally the elements by moving them towards right | √ |
|  | Horiz. Space →Make equal | It distributes equally the horizontal space among elements | √ |
|  | Horiz. Space →Increase | It increases of 5 pixel the horizontal space among elements (by moving them towards right) | √ |
|  | Horiz. Space →Decrease | It decreases of 5 pixel the horizontal space among elements (by moving them towards left) | √ |
|  | Horiz. Space →Remove | It removes the horizontal space among elements by moving them towards left | √ |
|  | Vert. Space →Make equal | It distributes equally the horizontal space among elements | √ |
|  | Vert. Space →Increase | It increases of 5 pixel the horizontal space among elements (by moving them towards right) | √ |
|  | Vert. Space →Decrease | It decreases of 5 pixel the horizontal space among elements (by moving them towards left) | √ |

| | | | |
|---|---|---|---|
|  | Vert. Space →Remove | It removes the horizontal space among elements by moving them towards left | √ |
|  | Adapt to parent | Increase the size of the element to fit the size of his container (a band, a cell or a frame) | |
|  | Adapt to parent width | Increase the width of the element to fit the width of his container (a band, a cell or a frame) | |
|  | Adapt to parent height | Increase the height of the element to fit the height of his container (a band, a cell or a frame) | |
|  | Organize as a table | Align on top the selected elements and make equal the horizontal space between them | |

**Figure 5-10 Formatting Tools - Definitions**

## 5.3 Managing Elements with the Report Inspector

The Report Inspector shows the complete report structure. The root node represents the page and it can be selected to modify all the general report properties as we have seen in the previous chapter. The following nodes are used for the style, the parameters, the fields and the variables and other report objects if present (like sub datasets).

After these nodes there are the bands. Each band contains the elements. Container elements (like frames) can have other elements represented as sub nodes. The order of the elements in the Inspector is important because it is the what is usually called the z-order (the position from the depth point of view), in other words, if an element precedes other elements in the inspector view, it will be printed before them. If an element overlaps some predecessors, it will cover them. Please note that some exporters (like the HTML exporter) does not support overlapping elements so they are skipped during the rendering, other times you can have two or more overlapped elements and print only one of them using the “print when condition”: this is a simple trick to print different content based on a condition.

To change the z-order, you can move the elements dragging them with the mouse in the inspector, or you can use the **Move Down** and **Move Up** menu items. Remember that elements on top of the list are printed first, so to bring an element to front, you need to move it down in the list.

All the elements can be copied and pasted, except for charts and crosstabs. When an element is pasted, it keeps the top/left coordinates used in its previous container (a band, a cell or a frame). If the new container is smaller than the previous one, you may need to adjust the element position since it could be outside the new container bounds.

The report inspector allows you to select elements inside the report even if those elements are not visible in the designer or even if they are hard to select due to the complexity of the report.

## 5.4 Basic Element Attributes

All the elements have a set of common attributes presented in the element properties view (as shown earlier in Figure 5-1). These attributes concern information about element positioning on the page: the following list describes the different attributes available.

 Coordinates and dimensions are always expressed in pixels in relation to a 72-pixel-per-inch resolution.

*Top* This is the distance of the top-left corner of the element from the top of the container the element belongs.

*Left* This is the distance of the top-right corner of the element from the left margin of the container.

*Width* This is the element width.

*Height* This is the element height; in reality, this indicates a minimum value that can increase during the print creation according to the value of the other attributes.

**Figure 5-11** shows how iReport positions an element relative to the band (or, more broadly, to its container) to which the element belongs. The band width is always equals to the document page width (minus the left and right margin); alternatively, its height can change depending on the type of band and by the contained elements.

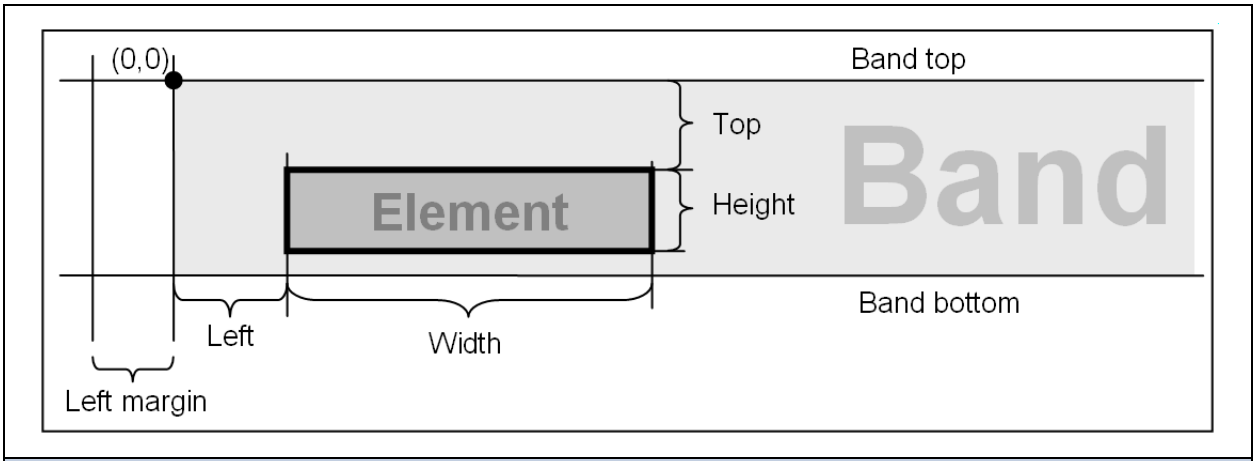



**Figure 5-11 Element Positioning**

*Foreground* This is the color with which the text elements are printed and the lines and the element corners are drawn.

*Background* This is the color with which the element background is filled. Since by default some elements are transparent, remember to make the element opaque.

*Opaque* This option controls if the element background has to be transparent or not; the transparency involves only the parts that should be filled with the background.

Not all export formats support the transparency attribute.

*Style* If the user has defined one or more styles in the report, it is possible to apply a style to the element selecting it from the list.

*Key* This is the element name, which has to be unique in the report (iReport proposes it automatically), and it is used by the programs that need to modify the field properties at runtime

*Position type* This option determines how the top coordinates have to be considered in the case that the band changes its height during the filling process. The three possible values are as follows:

- *FixRelativeToTop* This is the predefined position type; the coordinate values never change.
- *Float* The element is progressively pushed toward the bottom by the previous elements that increase their height.

| | |
|---|---|
| *FixRelativeToBottom* | The distance of the element from the bottom of the band remains constant; usually this is used for lines that separate records. |
| *Stretch type* | This attribute defines how to calculate the element height during the print elaboration; the three possible values are as follows: |
| *NoStretch* | This is the predefined stretch type, and it dictates that the element height should be kept equal. |
| *RelativeToBandHeight* | The element height is increased proportionally to the increasing size of the band; this is useful for vertical lines that simulate table borders |
| *RelativeToTallestObject* | The element modifies its height according to the deformation of the nearest element: this option is also used with the element group, which is an element group mechanism not managed by iReport |
| *Print repeated values* | This option determines whether to print the element when its value is equal to that which is used in the previous record. |
| *Remove line when blank* | This option takes away the vertical space occupied by an object, if it is not visible; the element visibility is determined by the value of the expression contained in the *Print when expression* attribute or in case of text fields by the *Blank when null* attribute too. Think of the page as a grid where the elements are placed, with a line being the space the element occupies. **Figure 5-12** highlights the *element A* line; in order to really remove this line, all the elements that share a portion of the line have to be null (that is, they will not be printed) |

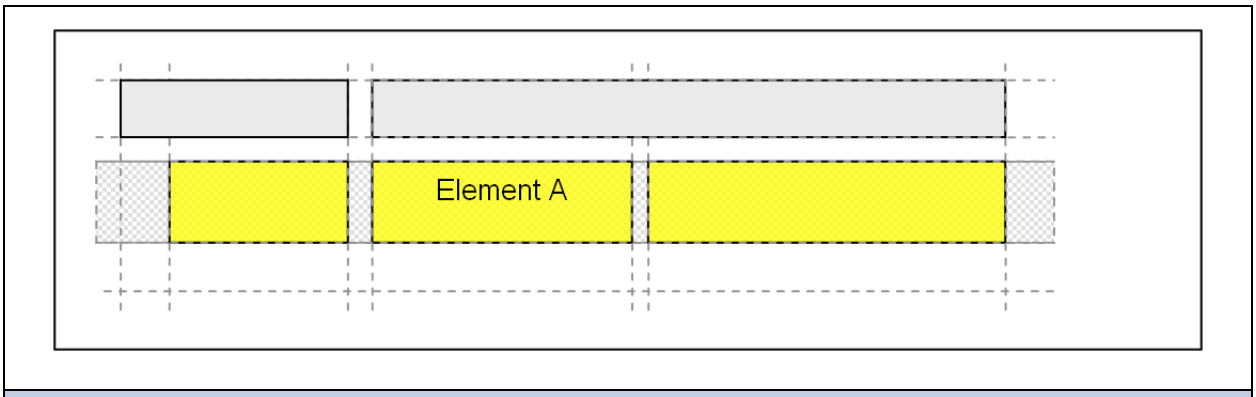



**Figure 5-12 Element A line**

| | |
|---|---|
| *Print in first whole band* | This option ensures that an element is printed in the next page (or column) if the band overflows the page (or the column); this type of guarantee is useful when the *Print repeated values* attribute. |
| *Print when detail overflows* | This option prints the element in the following page or column, if the band is not all printable in the present page or column. |
| *Print when group changes* | In this combo box, all report groups are presented. If one of them is selected, the element will be printed only when the expression associated to the group changes—that is, when a new break of the selected group is created. |

| | |
|---|---|
| *Print when expression* | This is an expression like those described in **Chapter 3**, and it must return a Boolean object; besides being associated to elements, this expression is associated to the bands, too. If the expression returns true, the element is hidden. An empty expression or a null value implicitly identifies an expression like `new Boolean(true)`, which will print the element unconditionally. |
| *Properties expressions* | They are a set of key/value pairs that can be defined for each element. |

## 5.5 Element Custom Properties

For each element it is possible to define a set of simple custom properties: each property is a pair key/value where both key and value are simple text strings. The value can be dynamic and generated using an expression (that clearly will have to return a String).

Element custom properties are set by modifying the **Properties expressions** attribute in the property sheet displayed when the element is selected (see **Figure 5-13**):

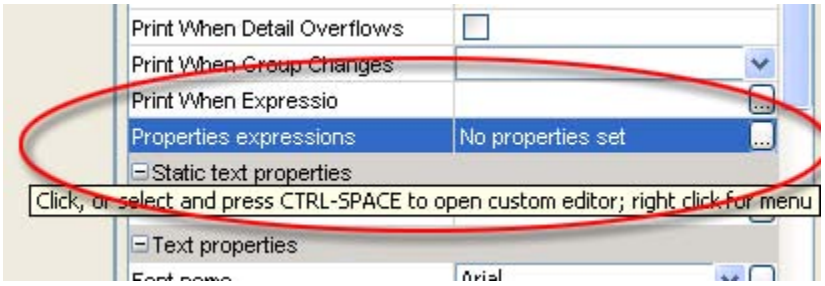



**Figure 5-13 Custom element properties**

The custom element properties can be used for many purposes, like to specify special behavior for the element when it is exported in a particular format or to set how characters have to be treated in a text field or again to set special tags like those required Standard 508 to define the structure of a document.

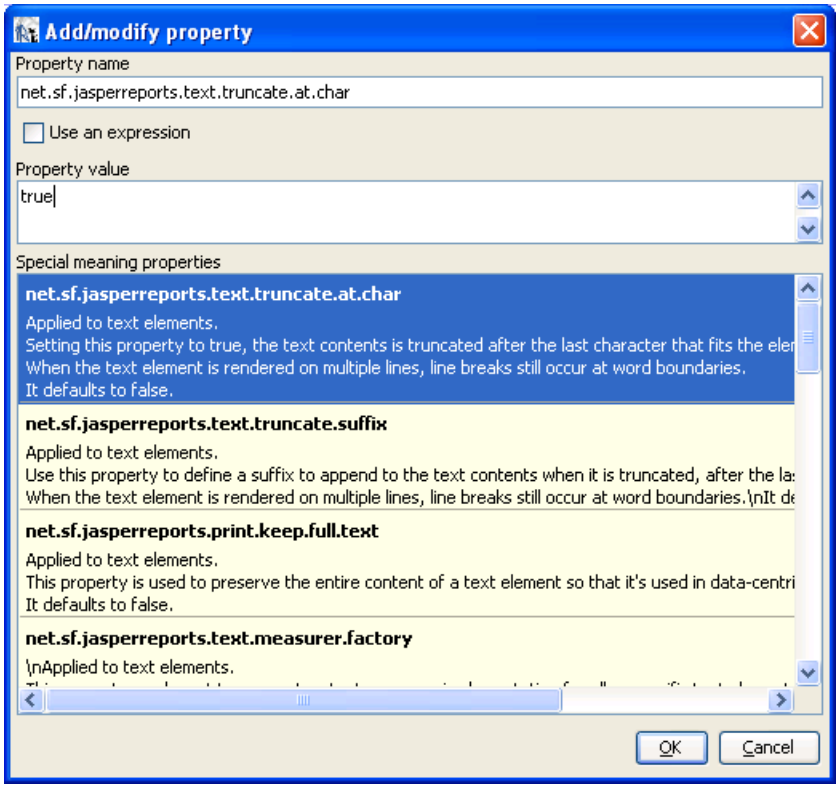



**Figure 5-14 Custom Property Dialog**

When a property is created, the property dialog suggests some of the most important common property keys with a little description of the property meaning.

To use an expression, check the **Use an expression** check box: the text area will become an expression area and the button to open the expression editor will appear.

## 5.6 Graphics Elements

The graphic elements are drawing objects such as the line and the rectangle; they do not show data generally, but they are used to make prints more readable and agreeable from an aesthetic point of view. All kinds of elements have the *pen* and the *fill* properties.

The *pen* is used to draw a shape (or just the borders of the element in case of images). This property is edited with the Pen dialog (see **Figure 5-15**).

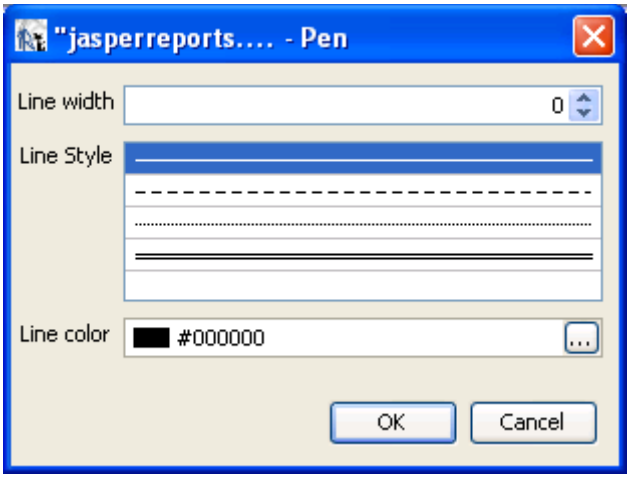



**Figure 5-15 Pen Definition**

It is possible to set a particular line width (a zero line width means that no lines will be painted) and choose between 4 different styles: normal, dashed, dotted and double.

By default, the color used to paint the lines is the element foreground color, but it is possible to override that value by specifying a different value. To reset the color the default value you need to reset the whole pen right clicking the *Pen* item in the property sheet and selecting *Restore Default Value*.

The default values for the pen (like for many other common element properties) depend on the specific element. Lines, rectangles and ellipses have a default width of 1 pixel, while for images the default line width is zero.

The *fill* property has a single possible value: *Solid*.

### 5.6.1 Line

In JasperReports, a line is defined by a rectangle for which the line represents the diagonal (see **Figure 5-16**).

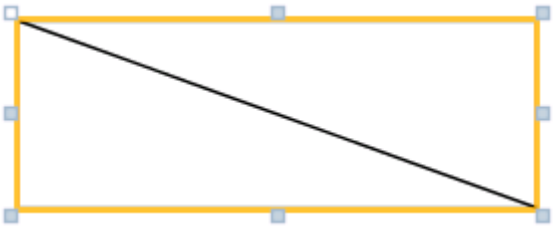


**Figure 5-16 Line element of type top-down**

The line is drawn using the pen settings. If they are not set, the foreground color is used as the default color and a normal 1 pixel width line is used as the line style.

The only specific property of a line is the *Line direction*, used to indicate which of the two rectangle diagonals represents the line; the possible values are *Top Down* and *Bottom Up*.

### 5.6.2 Rectangle

The rectangle element is usually used to draw frames around other elements (even if it is preferable use a frame element for this specific purpose in order to avoid overlapping elements). Similarly to the line element, the rectangle border is drawn using the pen settings. If they are not set, the Foreground is used as color (which is black by default) and a normal 1 pixel width line is used as line style. The background is filled with the color specified with the Background setting if the element has not been defined as transparent.

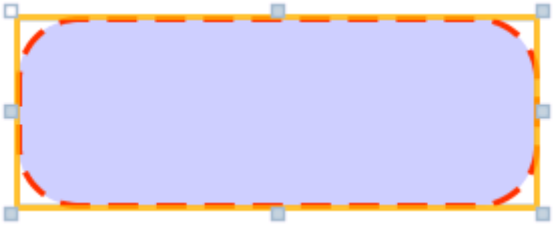


**Figure 5-17 Rectangle element with radius set to 20**

In JasperReports, it is possible to have a rectangle with rounded corners (see **Figure 5-17**). The rounded corners are defined by means of the *Radius* attribute, which represents the curvature radius of the corners, expressed in pixels.

### 5.6.3 Ellipse

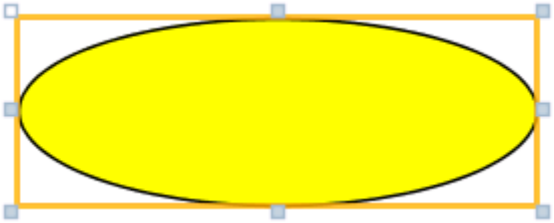


**Figure 5-18 Rectangle element with radius set to 20**

The ellipse is the only element that has no attributes specific to it. The ellipse is drawn into a rectangle that is built up on the four sides that are tangent to it (see **Figure 5-18**). The border is drawn using the pen settings. If they are not set, the Foreground is used as color (which is black by default) and a normal 1 pixel width line is used as line style. The background is filled with the Background color setting if the element has not been defined as transparent.

## 5.7 Working with Images


**Figure 5-19 Image Element**

An image is the most complex of the graphic elements (see **Figure 5-19**). It can be used to insert raster images (such as GIF, PNG and JPEG images) into the report, but it can be also used as a *canvas* object to render for example a Swing component or to leverage some custom rendering code.

Dragging an image element into the designer, iReport pops up a file chooser dialog. This is the most convenient way to specify an image to use in the report. iReport will not save or store the selected image anywhere, it will just use the file location, translating the absolute path of the selected image into an expression to locate the file when the report will be executed. The expression is then set as value for the *Image Expression* property. Here is a sample expression:

```
"C:\\Documents and Settings\\gtoffoli\\Desktop\\splashscreen.png"
```

As you can see, this is a real Java (or Groovy or Javascript) expression, not just the value of a file path, it starts and ends with double quotes and the back slash character (\) is escaped with another back slash (\\).

The *Image Expression Class* defines what kind of object is returned by the *Image Expression*. In this case it is of the type `java.lang.String`, but there are several other options.

**Figure 5-20** summarizes the values that the *Image Expression Class* can adopt and describes how the *Image Expression* result is interpreted and used.

| Type | Interpretation |
|---|---|
| `java.lang.String` | A string is interpreted like a file name; JasperReports will try to interpret the string like an absolute path; if no file is found, it will try to load a resource from the classpath with the specified name.Correct expressions are:<br>`“c:\\devel\\ireport\\myImage.jpg”`<br>`“com/mycompany/resources/icons/logo.gif”` |
| `java.io.File` | It specifies a File object to load as an image.<br>A correct expression could be:<br>`new java.io.File(“c:\\myImage.jpg”)` |

| | |
|---|---|
| `java.net.URL` | It specifies the java.net.URL object. It is useful when you have to export the report in HTML format. A correct expression could be:<br>`new java.net.URL("http://127.0.0.1/test.jpg")` |
| `java.io.InputStream` | It specifies a java.io.InputStream object which is ready for reading; in this case we do not consider that the image exists and that it is in a file: in particular we could read the image from a database and return the inputStream for reading. A correct expression could be:<br>`MyUtil.getInputStream(${MyField})` |
| `java.awt.Image` | It specifies a `java.awt.Image` object; it is probably the simplest object to return when an image has to be created dynamically. A correct expression could be:<br>`MyUtil.createImage()` |
| `JRRenderable` | It specifies an object that uses the `net.sf.jasperreports.engine.JRRenderable` interface. |

**Figure 5-20 Image Expression classes**

You are free to add an image by explicitly defining the full absolute path of a file in your expression. In effect this is a really easy way to add an image to the report, but overall it has a big impact on the report portability, since the file may not be found on another machine (i.e., after deploying the report on a web server or running the report on a different computer). There are two best practices here: the first is to parametrize the image expression using a parameter (possibly with a default value) containing the folder where your images resides, and then compose the expression with something like:

```
$P{MY_IMAGES_DIRECTORY} + "myImage.png"
```

At run time in a hypothetical application, the value for the parameter `MY_IMAGES_DIRECTORY` can be set by the application itself. If a value for the parameter is not provided, we can still return a default value (we'll see how to create a parameter and set a default value in the next chapter). The advantage of this solution is that the location of the directory where the images reside is not defined discretely within the report, but can be provided dynamically.

The second option is to use the *classpath*. The *classpath* is a set of directories and JAR file locations in which a Java application (like JasperReports) looks for classes and resources. It is usually easy to add directories to the classpath of an application that uses a Java Virtual Machine. In iReport the classpath can be extended from the options dialog (**Window > Options > iReport > Classpath**). When an image is in the classpath, the only required information needed by JasperReports to find and render the image is the resource name (that's again some kind of path but relative to a classpath directory). By default, when executing a report, iReport adds the directory in which the report resides to the classpath. Suppose you have a report in a certain directory, let's say "`c:\test\myReport.jrxml`", and in the same directory you have an image called "`myImage.png`". To use it in the report you can set x-value of Image Expression to be "`myImage.png`". Since the report's directory is added to the classpath, the image will be found automatically.

This process is still valid if the image resides in a subdirectory of a directory included in the classpath. In that case you'll need to specify the complete resource path using a Unix-style path notation.For example if your image resides in the directory `c:\test\images`, the resource is found with the expression "`/images/myImage.png`". What happens here is that JasperReports will check in the directory `c:\test` (which is in the classpath) if the specified resource path (in this case `/images/myImages.png`) exists.

This method to resolve resources location is applied in many other parts of JasperReports, (e.g., to locate a sub-report JASPER file, a resource bundle, a scriptlet class and so on).

Let’s take a look at the remaining options:

*Scale Image* — Defines how the image has to adapt to the element dimension; the possible values are three:

- *Clip* — The image dimension is not changed

- *FillFrame* — The image is adapted to the element dimension (becoming deformed)

- *RetainShape* — The image is adapted to the element dimension by keeping the original proportions

*On error type* — Defines what to do if the image loading fails:

- *Error* — (default) thrown a java exception stopping the filling process
- *Blank* — the image is not printed, and a blank space will be placed in the report instead
- *Icon* — an icon is printed instead of the original image

*Is Lazy* — avoid the loading of the image at fill time, the image will be loaded when the report will exported: it's useful when an image is loaded from an URL

*Using cache* — Allows to keep the image into the memory in order to use it again if the element is printed newly; the image is kept in cache only if the *Image Expression Class* is set to `java.lang.String`

*Vertical alignment* — Defines the image vertical alignment according to the element area; the possible values are:

- *Top* — the image is aligned at the top
- *Middle* — the image is put in the middle vertically according to the element area
- *Bottom* — the image is aligned at the bottom

*Horizontal alignment* — this attribute defines the image horizontal alignment according to the element area; the possible values are:

- *Left* — the image is aligned to the left
- *Center* — the image is put in the center horizontally according to the element area
- *Right* — the image is aligned to the right

*Evaluation time* — it defines at which time of report creation the *Image Expression* has to be processed; in fact the evaluation of an expression can be done when the report engine "encounters" the element during the creation of the report (evaluation time "now") or it can be also postponed for example because the image depends by some calculations that have not been yet completed. The evaluation time is applied to the evaluation of many other expressions (like text fields and variables). An in depth explanation of the evaluation time is available in the next chapter. The possible values for the evaluation time are:

- *Now* — Evaluate the expression immediately
- *Report* — Evaluate the expression at the end of the report
- *Page* — evaluate the expression at the end of the page
- *Column* — evaluate the expression at the end of this column
- *Group* — evaluate the expression of the group which is specified in *Evaluation group*
- *Band* — evaluate this expression after the evaluation of the current band (used to evaluate expressions that deal with sub-report return values)

*Evaluation group* — See the preceding *Group* value description for the Evaluation time setting.

## 5.8 Padding and Borders

For the image element (and for the text elements) it is possible to visualize a frame or to define a particular *padding* for the four sides. It is a space between the element border and its content. Border and padding are specified by right-clicking the element (or the element node in the inspector view) and selecting the menu item **Padding And Borders**. This will open the dialog box shown in **Figure 5-21**.

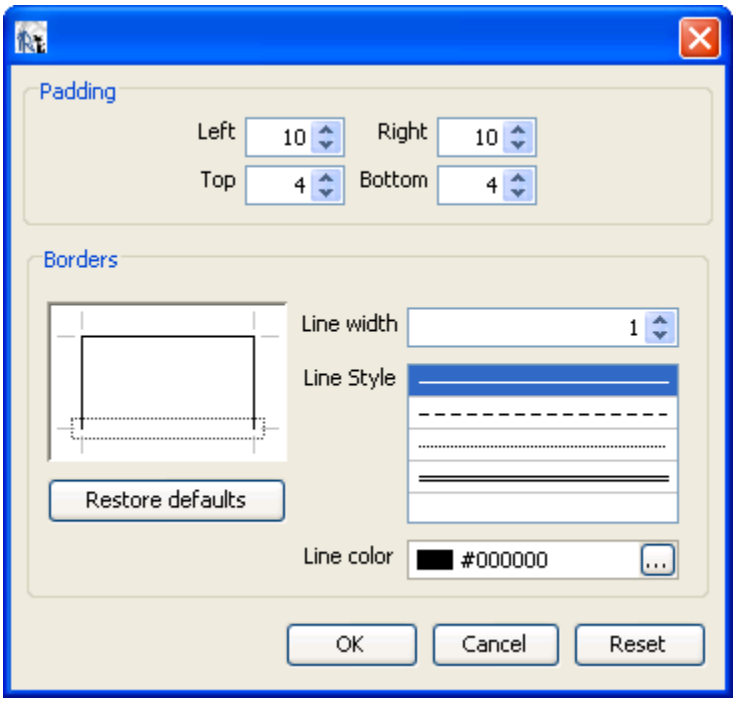



**Figure 5-21 Padding and Borders**

As always, all the measures must be set in pixels.

The four sides of the border can be edited individually by selecting them in the preview frame. When no sides are selected, changes are applied to all of them.

Image elements can have a hyperlink. Not all the export formats support them, but they work for sure in HTML, PDF and XLS. To define the hyperlink, right-click the image element and select the *Hyperlink* menu item. The hyperlink definition dialog will appear. We will explain in depth how to define an hyperlink using this dialog later in the chapter.

## 5.9 Loading an Image from the Database (BLOB Field)

If you need to print images that are stored in a database (i.e. using a BLOB column) what you need to do is assign the field that will get the BLOB value the type `java.awt.Image` (report fields will be explained in the next chapter). Create an image element by dragging the image element tool from the palette into the designer (i.e. into the detail band), click *cancel* when the file chooser prompts; in the image element properties sheet change the *Image Expression Class* to `java.awt.Image` and set as *Image Expression* the field object (i.e. `$F{MyImageField}` ).

## 5.10 Creating an Image Dynamically

To create an image dynamically requires some Java knowledge. Here we will show the best solution to modify or create an image to be printed in a report.

There are several ways to create an image dynamically in JasperReports. The first option is to write a class that produces a `java.awt.Image` object and call a method of this class in the *Image Expression* of the image element. The expression would look like:

```
MyImageGenerator.generateImage()
```

where `MyImageGenerator` is a class with the static method `generateImage()` that returns the `java.awt.Image` object. The problem with this solution is that, since the image created would be a raster image with a specific with and height, in the final result there could be there a loss of quality, especially when the document is zoomed in, or when the final output is a PDF file.

Generally speaking, the best format of an image that must be rendered in a document is an SVG, which provides high image quality regardless of original capture resolution. In order to ease the generation of a custom image, JasperReports provides an interface called `JRRenderable` that a developer can implement to get the best rendering result. A convenient class to initial use of this interface is `JRAbstractSVGRenderable`. The only method to implement here is:

```
public void render(Graphics2D g2d, Rectangle2D rect)
```

which is where you should put your code to render the image. **Figure 5-22** shows a simple implementation of a `JRAbstractSVGRenderable` to paint the outline text “JasperReports!!” inside an image element using a gradient background.

**Figure 5-22 Dynamic image generation**

```
package com.jaspersoft.ireport.samples;

import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.Shape;
import java.awt.font.FontRenderContext;
import java.awt.font.TextLayout;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import net.sf.jasperreports.engine.JRAbstractSvgRenderer;
import net.sf.jasperreports.engine.JRException;

/**
 *
 * @author gtoffoli
 */
public class CustomImageRenderer  extends JRAbstractSvgRenderer {
```

**Figure 5-22 Dynamic image generation**

```
  public void render(Graphics2D g2d, Rectangle2D rect) throws JRException {

    // Save the Graphics2D affine transform
    AffineTransform savedTrans = g2d.getTransform();
    Font savedFont = g2d.getFont();

    // Paint a nice background...
    g2d.setPaint(new GradientPaint(0,0, Color.ORANGE,
     0,(int)rect.getHeight(), Color.PINK));

    g2d.fillRect(0,0 , (int)rect.getWidth(), (int)rect.getHeight());
    Font myfont = new Font("Arial Black", Font.PLAIN, 50);
    g2d.setFont(myfont);

    FontRenderContext frc = g2d.getFontRenderContext();
    String text = new String("JasperReports!!!");

    TextLayout textLayout = new TextLayout(text, myfont, frc);
    Shape outline = textLayout.getOutline(null);
    Rectangle r = outline.getBounds();

    // Translate the graphic to center the text
    g2d.translate(
     (rect.getWidth()/2)-(r.width/2),
     rect.getHeight()/2+(r.height/2));

    g2d.setColor(Color.BLACK);
    g2d.draw(outline);

    // Restore the Graphics2D affine transform
     g2d.setFont(savedFont);
     g2d.setTransform( savedTrans );
    }

}
```

The final result is shown in **Figure 5-23**. The `CustomImageRenderer` class implements the interface `JRAbstractSvgRenderer`. The render just fills the background with the `fillRect` method using a Gradient Paint, creates a shape out of the "JasperReports!!!" text and render the shape centering it with a translation.

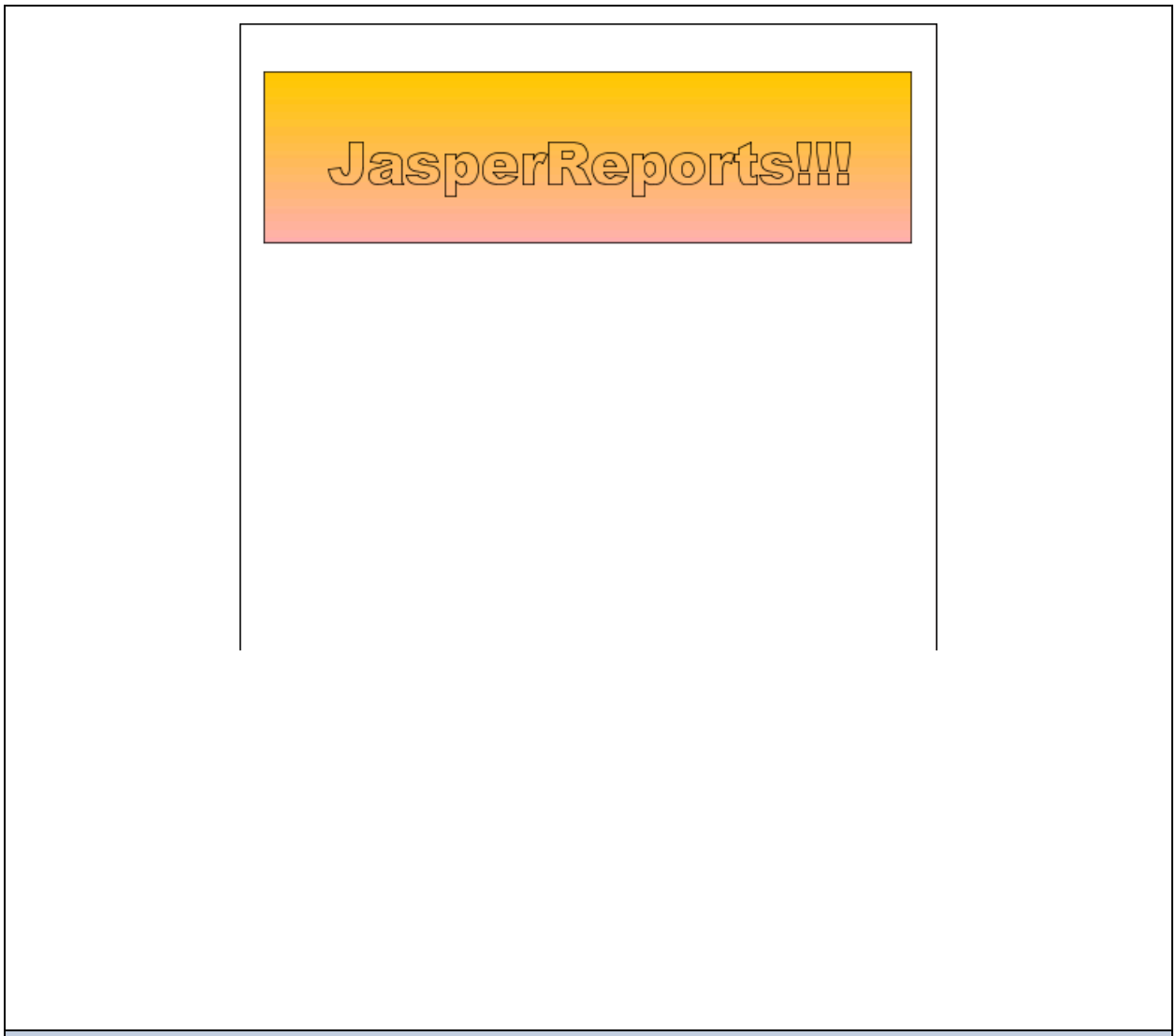



**Figure 5-23 An image element rendered using a custom `JRRenderable` object**

What we did is to set the *Image Element Expression* to:

```
new  com.jaspersoft.ireport.samples.CustomImageRenderer()
```

and the *Image Expression Class* to `net.sf.jasperreports.engine.JRRenderable`. We have not passed any argument to our implementation class, but this is possible, allowing the final user to pass to the renderer extra information to produce the image.

With a similar approach it is possible to modify an image (rotate, transform and so on), add a watermark to an image or even insert into the report a Swing component.

Figure 5-24 shows how to print a `JTable`. The code is not much different from what we have seen in the previous sample, the idea is to force the component to paint itself into the provided Graphics2D. The result is incredibly good and there is no loss of quality when using the internal JasperReports preview component (see Figure 5-25) or when exporting to PDF.

**Figure 5-24 Printing a `JTable`**

```
package com.jaspersoft.ireport.samples;

import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.JTableHeader;
import net.sf.jasperreports.engine.JRAbstractSvgRenderer;
import net.sf.jasperreports.engine.JRException;

/**
 *
 * @author gtoffoli
 */
public class SwingComponentRenderer extends JRAbstractSvgRenderer {

 public void render(Graphics2D g2d, Rectangle2D rect) throws JRException {

  / Save the Graphics2D affine transform
  AffineTransform trans = g2d.getTransform();

  // Create a simple table model
  DefaultTableModel model = new DefaultTableModel(
   new Object[][] {
    {"Mercury","NO"},
    {"Venus","NO"},
    {"Earth","YES"},
    {"Mars","YES"},
    {"Jupiter","YES"},
    {"Saturn","YES"},
    {"Uranus","YES"},
    {"Neptune","YES"},
    {"Pluto","YES"}},
   new String[]{"Planet","Has satellites"});
```

**Figure 5-24 Printing a `JTable`**

```
   // Create the table using the model
   JTable table = new JTable(model);

   // Set the column size
   table.getColumn("Planet").setWidth(100);
   table.getColumn("Has satellites").setWidth(100);

   // Resize the table to accomodate the new size
   table.setSize(table.getPreferredSize());

   // Get the header and set the size to the preferred size
   JTableHeader tableHeader = table.getTableHeader();
   tableHeader.setSize(tableHeader.getPreferredSize());

   // Paint the header
   tableHeader.paint(g2d);

   // Paint the table
   g2d.translate(0, tableHeader.getHeight());
   table.paint(g2d);

   // Restore the Graphics2D affine transform
   g2d.setTransform( trans );
  }

}
```

## 5.11 Working with Text

There are two elements specifically designed to display text in a report: *Static Text* and *Text Field*. The first is used for creating labels or more in general to print static text set at design time and not supposed to change when the report is generated. That said, in some cases you will still use a *Text Field* to print labels too, since the nature of the static text elements prevents the

ability to display text dynamically translated in different languages when the report is executed with a specific locale and it is configured to use a resource bundle leveraging the JasperReports internationalization capabilities.

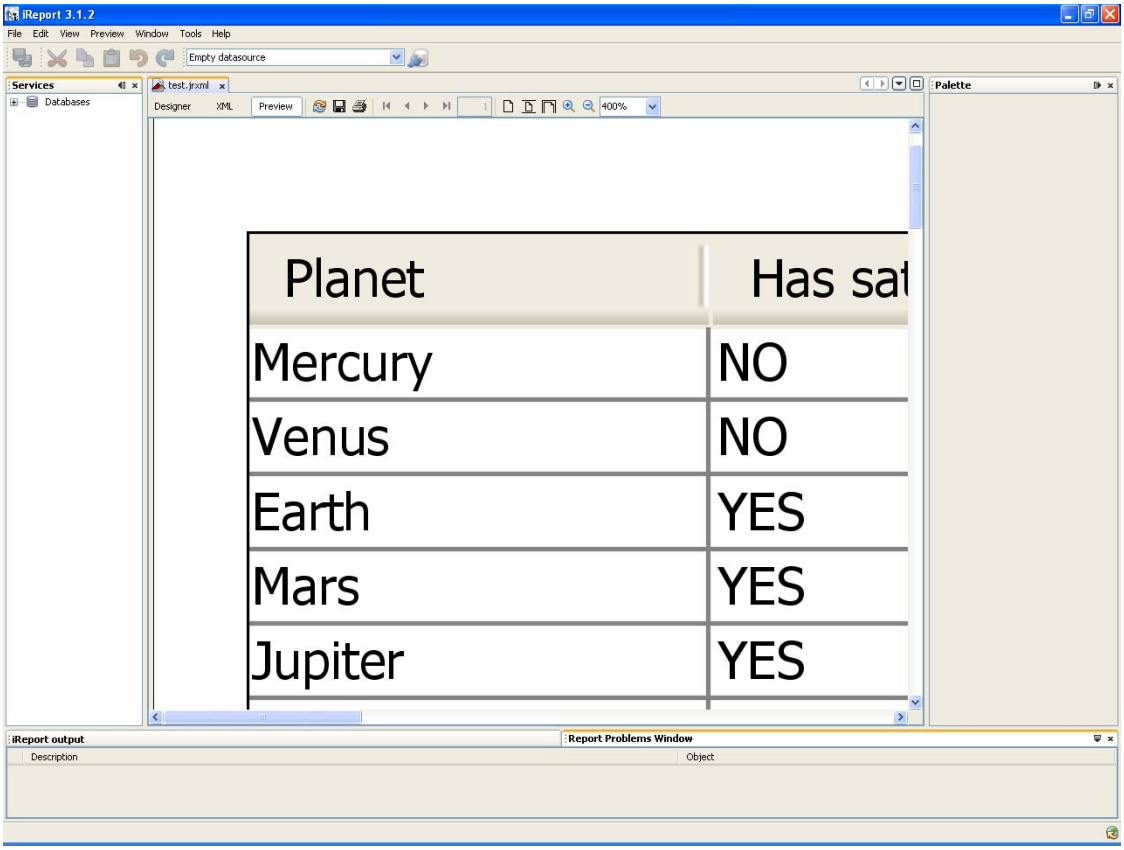



**Figure 5-25 A `JTable` printed inside an image element**

A text field acts in a way similar to a static text string, but the content (the text to print) is provided using an expression (that actually could be a simple static text string itself). That expression can return several kinds of value types, not just `String`, allowing the user to specify a pattern to format that value (this can be applied in example on numeric values and dates). Since the text specified dynamically can have an arbitrary length, a text field provides several options about how the text must be treated regarding alignment, position, line breaks and so on. Optionally the text field is able to grow vertically to fit the content when required. By default text elements are transparent with no border and with a black foreground color. All these properties can be modified using the common portion of the element property sheet and using the pop-up menu *Padding And Borders*. Text fields support hyperlinks too, refer the section about how to define them later in this chapter for more information.

Static texts and Text fields share a set of common properties to define text alignment and fonts. Let's take a look at these options:

| | |
|---|---|
| *Font name* | This is the name of the font, the list presents all the fonts found in the system. Like often happens with text documents, you may see fonts that are could not be found on other machines, so choose your font name carefully to keep the maximum compatibility. When exporting in PDF, this property is ignored, since PDF requires and the PDF font name is used instead. More information about the correct use of the fonts are provided in the "Fonts" chapter. |
| Font size | This is the size of the font. Only integer numbers are allowed, meaning that you can not define i.e. a size of 10.5. |
| *Bold*<br>*Italic* | Set the text style to bold and italic. These two properties does not work when the report is exported in PDF. In that case you need to specify a proper PDF font. |

*Underline*
*Strikethrough*
Set the text style to underline and strikethrough.

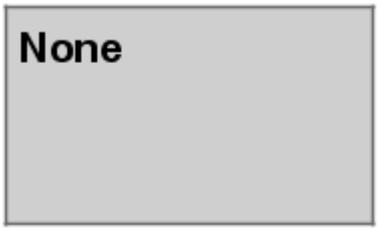
*PDF font name*
*PDF encoding*
*PDF embedded*
These flags are explained in the “Fonts” chapter.

*Horizontal align*
This is the horizontal alignment of the text according to the element.

*Vertical align*
This is the vertical alignment of the text according to the element.

*Rotation*
This specifies how the text has to be printed. The possible values are as follows:

*None*
The text is printed normally from left to right and from top to bottom.




*Left*
The text is rotated of 90 degrees counterclockwise; it is printed from bottom to top, and the horizontal and vertical alignments follow the text rotation (for example, the bottom vertical alignment will print the text along the right side of the rectangle that delimits the element area)


*Right*
The text is rotated of 90 degrees clockwise from top to bottom, and the horizontal and vertical alignments are set according to the text rotation.


*UpsideDown*
The text is rotated 180 degrees clockwise.


*Line spacing*
This is the interline (spacing between lines) value. The possible values are as follows:

*Single* — Single interline (predefined value)

*1.5* — Interline of one line and a half

*Double* — Double interline

*Markup* This attribute allows you to format the text using a specific markup language. This is extremely useful when you have to print some text that is pre-formatted i.e. in HTML or RTF. Simple HTML style tags (like <b> for bold and <i> for Italic) can be used in example to highlight a particular chunk of the text. The possible values are as follows:

- *None* — No processing on the text is performed, and the text is printed exactly like it is provided.
- *Styled* — This markup is capable to format the text using a set of HTML-like tags and it is pretty popular in the Java environments. It allows to set a specific font for chunks of text, color, background, style and so on. It's often good enough to format the text programmatically.
- HTML — If you want to print some HTML text into your report, this is what you need, but it's primary use is to format text, so don't expect to be able to print tables or add images.
- RTF — Setting the markup to this value, the content will be interpreted as RTF code. RTF is a popular document format stored in pure text. The little piece of text saying “this is a text formatted in RTF” in Illustration 19 has been generated using the string:

```
{\rtf1\ansi\ansicpg1252\deff0\deflang1033{\fonttbl{\f0\fswiss\fcharset0
Arial;}{\f1\fnil\fprq2\fcharset0 Swift;}}
{\*\generator Msftedit 5.41.15.1507;}\viewkind4\uc1\pard\f0\fs20 This is a
text \f1\fs52 formatted \f0\fs20 in RTF\par
}
```

The string is actually an RTF file created using a simple word processor.

*Report font* This is the name of a preset font, from which will be taken all the character properties. This attribute is deprecated and it is there only for compatibility reason (that's why it the label is strikedthrough. In order to define a particular style of text to use all over your document, use a style (explained in **Chapter 8**)

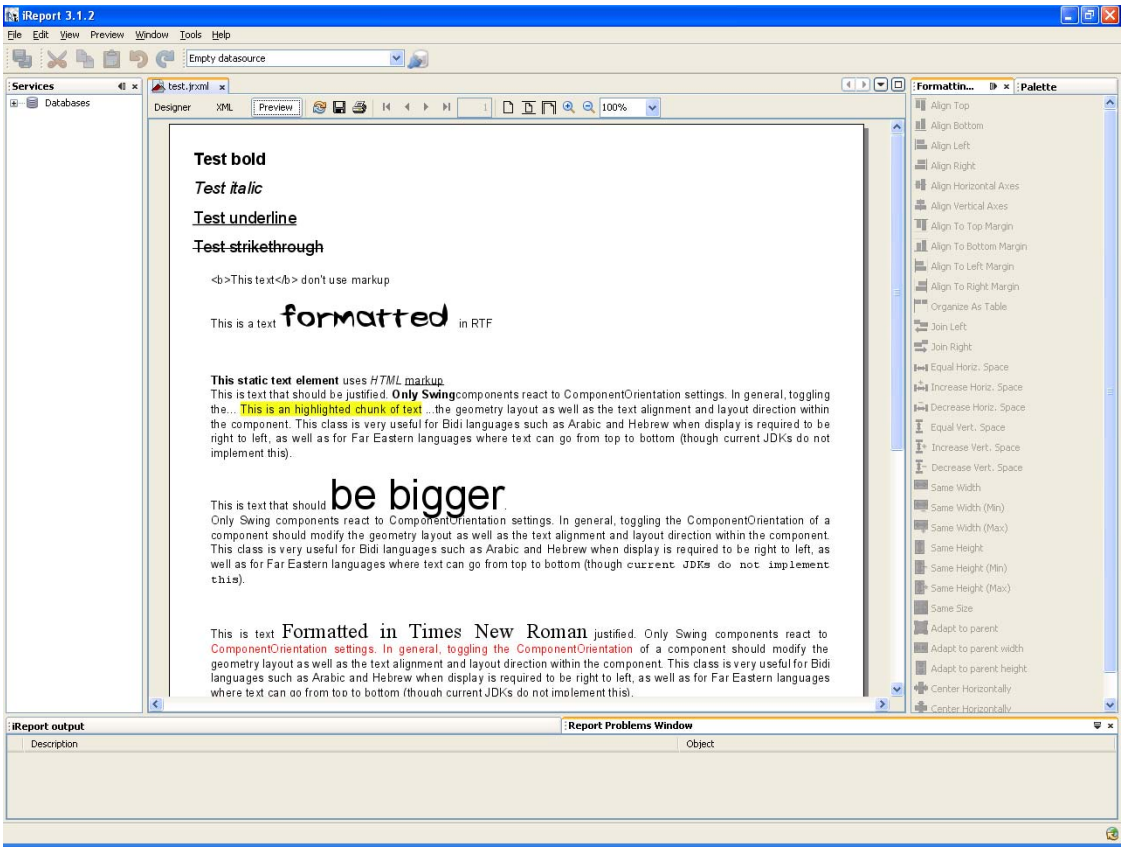



**Figure 5-26 Text formatted with several markups and styles**

For your convenience, the most used text properties can be modified using the text tool bar (see **Figure 5-27**) that is displayed when a text element is selected.


**Figure 5-27 Text Tool Bar**

The first two buttons on the left of the font size selector can be used to increase and decrease the font size.

## 5.12 Static Text

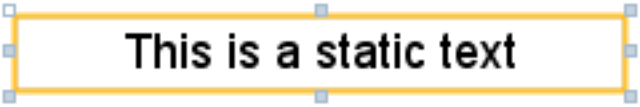



**Figure 5-28 A static text element**

The static text element is used to show non-dynamic text in reports (see **Figure 5-28**). The only parameter that distinguishes this element from a generic text element is the Text property, where the text to view is specified: it is normal text, not an expression, and so it is not necessary to enclose it in double quotes in order to respect the conventions of Java, Groovy, or JavaScript syntax.

## 5.13 Text Fields

A text field allows you to print an arbitrary section of text (or a number or a date) created using an expression. The simplest case of use of a textfield is to print a constant string (`java.lang.String`) created using an expression like this:

```
"This is a text"
```

A textfield that prints a constant value like the one returned by this expression can be easily replaced by a static field; actually, the use of an expression to define the content of a textfield provides a high level of control on the generated text (even if it's just constant text). A common case is when labels have to be internationalized and loaded from a resource bundle.

In general, an expression can contain fields, variables and parameters, so you can print in a textfield the value of a field and i.e. set the format of the value to present. For this purpose a textfield expression does not have to return necessarily a String (that's a text value): the *textfield expression class name* property specifies what type of value will be returned by the expression. It can be one of the following:

| Valid Expression Types | | |
|---|---|---|
| `java.lang.Object` | `java.sql.Time` | `java.lang.Long` |
| `java.lang.Boolean` | `java.lang.Double` | `java.lang.Short` |
| `java.lang.Byte` | `java.lang.Float` | `java.math.BigDecimal` |
| `java.util.Date` | `java.lang.Integer` | `java.lang.String` |
| `java.sql.Timestamp` | `java.io.InputStream` | |

**Figure 5-29 Expression Types for the Textfield**

An incorrect expression class is frequently the cause of compilation errors. If you use Groovy or JavaScript you can just choose `String` as expression type without causing an error when the report is compiled. The side effect is that without specifying the right expression class, the pattern (if set) is not applied to the value.

Let's see what properties can be set for a textfield:

*Blank when null* — If set to true, this option will avoid to print the textfield content if the expression result is a null object that would be produce the text “null” when converted in a string.

*Evaluation time* — it determines in which phase of the report creation the *Textfield Expression* has to be elaborated (an in depth explanation of the evaluation time is available in the next chapter)

| | |
|---|---|
| *Evaluation group* | it is the group to which the evaluation time is referred if it is set to *Group* |
| *Stretch with overflow* | when it is selected, this option allows the text field to adapt vertically to the content, if the element is not sufficient to contain all the text lines |
| *Pattern* | The pattern property allows to set a mask to format a value. It is used only when the expression class is congruent with the pattern to apply, meaning you need a numeric value to apply a mask to format a number, or a date to use a date pattern. |

The following tables provide some parameters and examples of patterns of data and numbers.

| Letter | Date components | Examples |
|---|---|---|
| G | Era designator | AD |
| y | Year | 1996; 96 |
| M | Month in year | July; Jul; 07 |
| w | Week in year | 27 |
| W | Week in month | 2 |
| D | Day in year | 189 |
| d | Day in month | 10 |
| F | Day of week in month | 2 |
| E | Day in week | Tuesday; Tue |
| a | Am/pm marker | PM |
| H | Hour in day (0-23) | 0 |
| k | Hour in day (1-24) | 24 |
| K | Hour in am/pm (0-11) | 0 |
| h | Hour in am/pm (1-12) | 12 |
| m | Minute in hour | 30 |
| s | Second in minute | 55 |
| S | Millisecond | 978 |
| z | Time zone | Pacific Standard Time; PST; GMT-08:00 |
| Z | Time zone | -0800 |

**Figure 5-30 Letters to Create Patterns for Dates**

Here there are some examples of date and time formats:

| Dates and Time Pattern | Result |
|---|---|
| "yyyy.MM.dd G 'at' HH:mm:ss z" | 2001.07.04 AD at 12:08:56 PDT |
| "EEE, MMM d, "yy" | Wed, Jul 4, '01 |
| "h:mm a" | 12:08 PM |

| | |
|---|---|
| "hh 'o"clock' a, zzzz" | 12 o'clock PM, Pacific Daylight Time |
| "K:mm a, z" | 0:08 PM, PDT |
| "yyyyy.MMMMM.dd GGG hh:mm aaa" | 02001.July.04 AD 12:08 PM |
| "EEE, d MMM yyyy HH:mm:ss Z" | Wed, 4 Jul 2001 12:08:56 -0700 |
| "yyMMddHHmmssZ" | 010704120856-0700 |

**Figure 5-31 Letters to Create Patterns for the Dates**

In the next table are examples of how certain special characters are parsed as symbols in numeric strings:

| Symbol | Location | Localized? | Meaning |
|---|---|---|---|
| 0 | Number | Yes | Digit |
| # | Number | Yes | Digit, zero shows as absent |
| . | Number | Yes | Decimal separator or monetary decimal separator |
| - | Number | Yes | Minus sign |
| , | Number | Yes | Grouping separator |
| E | Number | Yes | Separates mantissa and exponent in scientific notation. *Need not be quoted in prefix or suffix.* |
| ; | Subpattern boundary | Yes | Separates positive and negative subpatterns |
| % | Prefix or suffix | Yes | Multiply by 100 and show as percentage |
| \u2030 | Prefix or suffix | Yes | Multiply by 1000 and show as per mille |
| ¤ (\u00A4) | Prefix or suffix | No | Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator. |
| ' | Prefix or suffix | No | Used to quote special characters in a prefix or suffix, for example, "'#'#" formats 123 to "#123". To create a single quote itself, use two in a row: "# o"clock". |

**Figure 5-32 Symbols to Create Patterns for the Numbers**

Here there are some examples of formatting of numbers:

| Dates and Time Pattern | Result |
|---|---|
| "#,##0.00" | 1.234,56 |
| "#,##0.00;(#,##0.00)" | 1.234,56 (-1.234.56) |

**Figure 5-33 Example Patterns for Numbers**

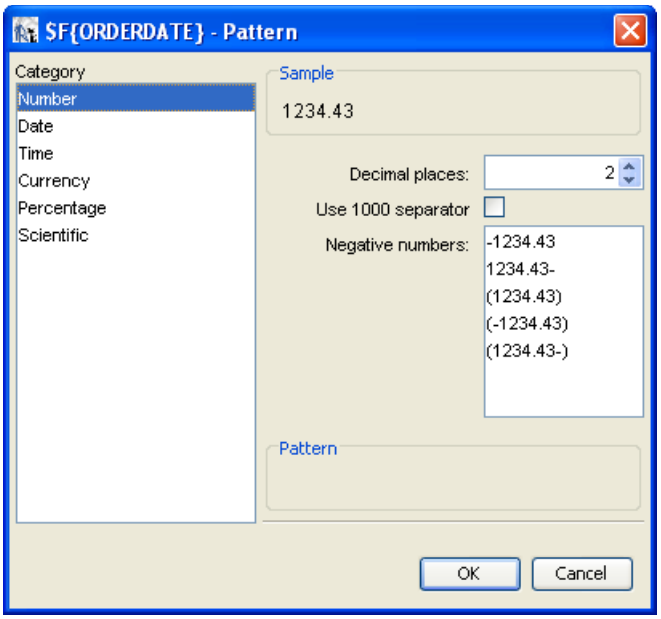



**Figure 5-34 Date Pattern**

To provide a convenient way to define string patterns, iReport provides a simple pattern editor. To open it, click the button labeled "..." for the pattern property in the property sheet.

## 5.14 Subreports

The subreport element is used to include inside a report another report represented by an external jasper file and feed using the same database connection used by the parent report or thought a data source that is specified in the subreport properties.

**Figure 5-35 Subreport Element**

This section briefly describes the characteristics of subreports.

| | |
|---|---|
| *Subreport Expression* | This identifies the expression that will return at runtime a subreport expression class object. According to the return type, the expression is valuated in order to recover a jasper object to be used to produce the subreport. In case the expression class is set to `java.lang.String`, JasperReports will look for a file following the same approach explained for the *Image Expression* of the Image element. |
| *Subreport Expression Class* | This is the class type of the expression; there are several options, each of one subtends to a different way to load the `JasperReport` object used to fill the subreport. |
| *Using cache* | This specifies whether to keep in memory the report object used to create the subreport in order to avoid to reload it all the times it will be used inside the report. It is common in facts that a subreport element placed for instance into the Detail band is printed more than once (or once for each record in the main dataset). The cache works only if the subreport expression is of type String since that string is used as key for the cache. |
| *Connection/Datasource Expression* | This identifies the expression that will return at runtime a JDBC connection or a JRDataSource used to fill in the subreport. Alternatively the user can choose to avoid to pass any data. This last option is possible and many times it is very useful, but requires some expedient in order to make the subreport to work. Since a subreport (like a common report) will return an empty document if no data are provided, the subreport document should have the document property *When No Data Type* set to something like *All Sections, No Detail*. |
| *Parameters Map Expression* | This optional expression can be used to produce at run time an object of type *java.util.Map*. The map must be contain a set of coupled names/objects that will be passed to the subreport in order to set a value for its parameters. Nothing disallows to use this expression in order to pass as parameters map to the subreport a map previously passed as parameter to the parent report. |
| *Subreport parameters* | This table allows you to define some coupled names/expressions that are useful for dynamically set a value for the subreport parameters by using calculated expressions. |
| *Subreport return values* | This table allows you to define how to store in local variables values calculated or processed in the subreport (such as totals and record count). |

## 5.15 Frame

A frame is an element that can contain other elements and optionally draw a border around them, as shown in **Figure 5-36**.


**Figure 5-36 Frame Element**

Since a frame is a container of other elements, in the document outline view the frame is represented as a node containing other elements (**Figure 5-37**).

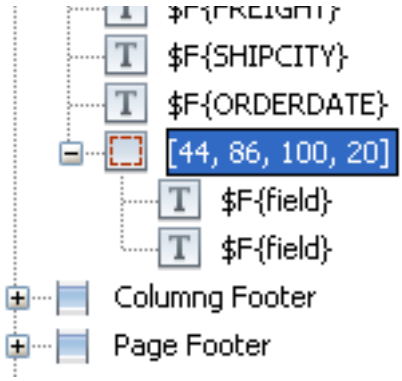



**Figure 5-37 A frame in the outline view**

A frame can contain other frames, and so on recursively. To add an element to a frame, just drag the new element from the palette inside the frame. Alternatively you can use the outline view and drag elements from a band into the frame and so on. The position of an element is always relative to the container position. If the container is a band, the element position will be relative to the top of the band and the left margin. If the container (or element parent) is a frame, the element coordinates will be relative to the top left corner of the frame. Since an element dragged from a container to another does not change his top/left properties, when moving an element from a container to another his position is recalculated based on the new container location.

The advantage of using a frame to draw a border around a set of elements, with respect to using a simple rectangle element, are:

- When you move a frame, all the elements contained in the frame will move in concert
- While using the rectangle you need to overlap some elements, with the frame the elements inside it will not treated as overlapped (respect to the frame), so you will not have problems when exporting in HTML (that does not support overlapped elements).
- Finally the frame will automatically stretch accordingly to its content, and the element *position type* property of its elements will refer to the frame itself, and not to the band, making the design a bit easier.

## 5.16 Chart

For all the details regarding charts, see Chapter **11, “Charts,” on page 191**.

## 5.17 Crosstab

For all the details regarding crosstabs, see Chapter **13, “Crosstabs,” on page 215**.

## 5.18 Page/Column Break

Page and column breaks are used to force the report engine to make a jump to the next page or column. A column break in a single column report has the same effect as a page break.

In the design view they are represented as a small line. If you try to resize them, the size will be reset to the default, this because they are used just to set a particular vertical position in the page (or better, in the band) at which iReport forces a page or column break.

The type of break can be changed in the property sheet.

## 5.19 Custom Components and Generic elements

Besides the built-in elements seen up to now, JasperReports supports two technologies to plug-in new elements, respectively called Custom components and *Generic elements*. Both are supported by iReport. Without a specific plug-in offered by the custom element provider, there is not much to do with them, you can just set the common element properties. The custom element developer should provide a plug-in for iReport through which at least add the element to the report (maybe adding a palette item) and modify the element properties (implementing what is required to display the additional properties in the property sheet when the element is selected. The image, the textfield, and the chart elements can be used both as anchors into a document and as hypertext links to external sources or other local anchor. An anchor is a kind of label that identifies a specific position in the document.

These hypertext links and anchors are defined by means of the Hyperlink dialog, shown in **Figure 5-38**.

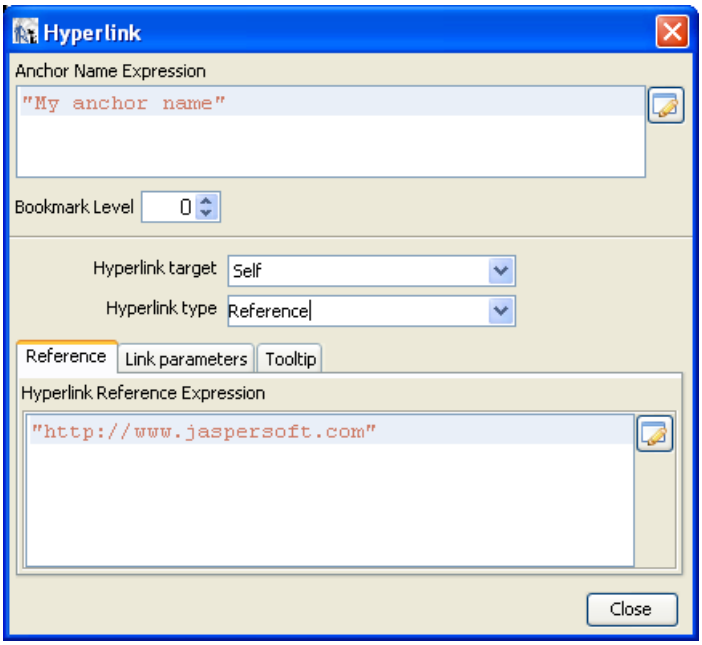



**Figure 5-38 Hyperlink Windows**

This dialog is divided in two parts. In the upper part is a text area through which it is possible to specify the expression that will be the name of the anchor. This name can be referenced by other links. If you plan to export your report as a PDF document, you can use the bookmark level to populate the bookmark tree, making the final document navigation much more easier. To make an anchor available in the bookmark, simply choose a bookmark level greater than 1. The use of a greater level makes possible the creation of nested bookmarks.

The lower part is dedicated to the link definition towards an external source or a position in the document. Through the Hyperlink target option, it is possible to specify whether the exploration of a particular link has to be made in the current window (this is the predefined setting and the target is Self) or in a new window (the target is Blank). This kind of behavior control makes sense only in certain output formats such as HTML and PDF, specially the last two possible targets (Top and Parent) used to indicate respectively to display the linked document in the current window but outside eventual frames, and in the parent window (if available).

The following text outlines some of the remaining options in the **Hyperlink** dialog.

### 5.19.1 Hyperlink Type

JasperReports provides five types of built-in hypertext links: Reference, LocalAnchor, LocalPage, RemoteAnchor and RemotePage. Anyway other types can be implemented and plugged into JasperReports (like the type `ReportExecution` used by JasperServer to implement the drill down features).

#### 5.19.1.1 Reference

The Reference link indicates an external source that is identified by a normal URL. This is ideal to point, for example, to a servelet to manage a record drill-down tools. The only expression required is the hyperlink reference expression.

#### 5.19.1.2 LocalAnchor

To point to a local anchor means to create a link between two locations into the same document. It can be used, for example, to link the titles of a summary to the chapters to which they refer.

To define the local anchor, it is necessary to specify a hyperlink anchor expression, which will have to produce a valid anchor name.

#### 5.19.1.3 LocalPage

If instead of pointing to an anchor you want to point to a specific current report page, you need to create a LocalPage link. In this case, it is necessary to specify the page number you are pointing to by means of a hyperlink page expression (the expression has to return an Integer object).

#### 5.19.1.4 RemoteAnchor

If you want to point to a particular anchor that resides in an external document, you use the RemoteAnchor link. In this case, the URL of the external file pointed to will have to be specified in the Hyperlink Reference Expression field, and the name of the anchor will have to be specified in the Hyperlink Anchor Expression field.

#### 5.19.1.5 RemotePage

This link allows you to point to a particular page of an external document. Similarly, in this case the URL of the external file pointed to will have to be specified in the Hyperlink Reference Expression field, and the page number will have to be specified by means of the hyperlink page expression.

Not all export formats support hypertext links.

### 5.19.2 Hyperlink Parameters

Sometimes you will need to define some parameters that must be “attached” to the link. The Link parameters table provides a convenient way to define them. The parameter value can be set using an expression. The parameter expression is supposed to be a string (since it will be encoded in the URL). But when using custom link types it makes sense to set different types for parameters.

### 5.19.3 Hyperlink Tooltip

The tooltip expression is used to set a text to display as tooltip when the mouse is over the element that represents the hyperlink (this only works when the document is exported in a format that supports this type of interactive use).

# 6 FIELDS, PARAMETERS, AND VARIABLES

In a report, there are three groups of objects that can store values:

- Fields
- Parameters
- Variables

iReport uses these objects expressions in data source queries. They must be declared with a discrete type that corresponds to a Java class, such as `String` or `Double`. After they have been declared in a report design, objects can modified or updated during the report generation process.

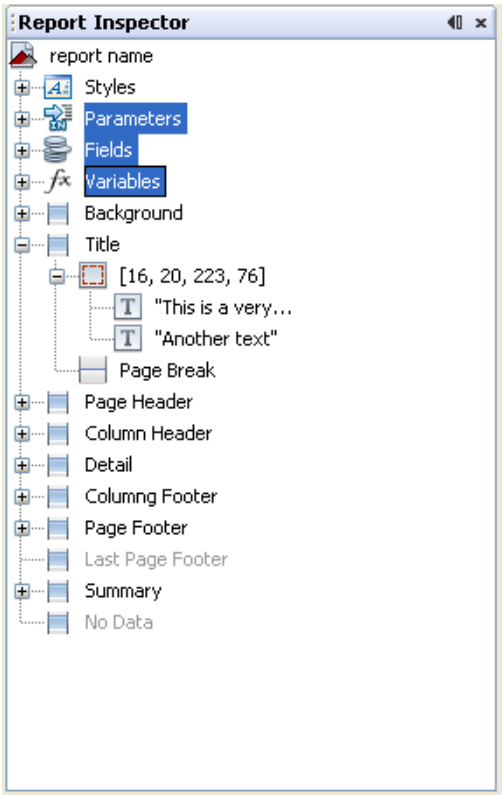



**Figure 6-1 Report objects in the outline view**

Fields, parameters, and variables must be declared in order to use them in a report.After they have been declared, they can be managed using the Report Inspector view (see Figure 6-1). In the Report Inspector you can modify or remove objects, and declare new objects as well.

## 6.1 Working with Fields

A print is commonly created starting from a data source that provides to the report a set of records composed of a series of fields exactly like the results of an SQL query.

iReport displays available fields as children of the **Fields** node in the document outline view. To create a field, right-click the **Fields** node and select the **Add Field** menu item. The new field will be included as an undefined child node in the Report Inspector, from which you can configure the field properties by selecting it and using the property sheet (**Figure 6-2**).

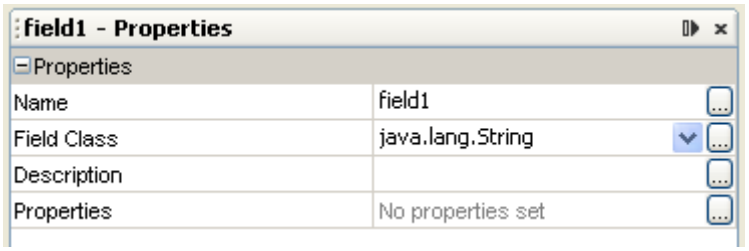



**Figure 6-2 Field Properties**

A field is identified by a unique name, a type, and an optional description. Additionally, you can define a set of name/value pair properties for each field. These custom properties are not used directly by JasperReports, but they can be used by external applications, or by some custom modules of JasperReports (like a special query executor). You can set the custom properties with the properties editor (**Figure 6-2**), which you can open by clicking on the editor button in the column to the right in the outline view, labeled "**...**".

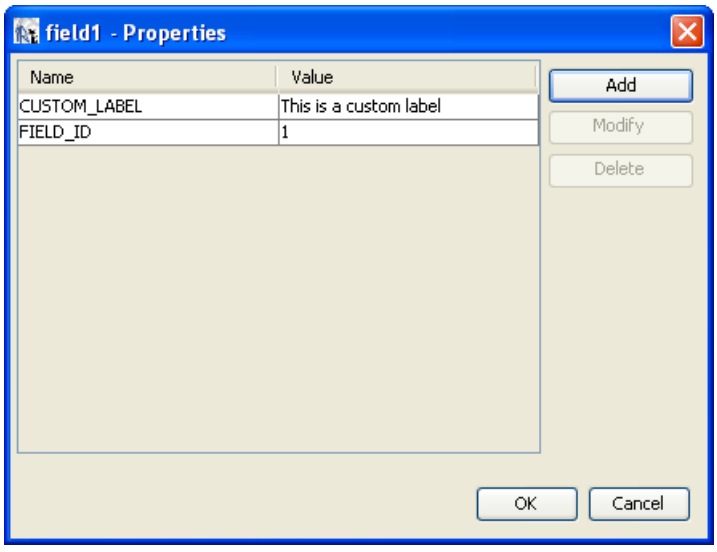



**Figure 6-3 Field - Custom Properties**

Before the introduction of the custom properties, iReport includes additional information regarding the selected field in its description field. An example of this would be the definition of fields to be used with an XML data source (i.e., a data source based on an XML file), where the field name can be arbitrary while the description holds an Xpath expression to locate the value within the XML document.

iReport determines the value for a field based on the data source you are using. For instance, when using an SQL query to fill the report, iReport assumes that the name of the field is the same as a name of a field in the query result set. You must ensure that the field name and type matches the field name and type in the selected data source. We will see, though, how you can systematically declare and configure large numbers of fields using the tools provided by iReport. Since the number of fields in a report can be quite large (possibly reaching the hundreds), iReport provides different tools for handling declaration fields retrieved from particular types of data sources.

Inside each report expression (like the one used to set the content of a textfield) iReport specifies a field object, using the following syntax:

```
$F{<field name>}
```

where <*field name*> must be replaced with the name of the field. When using a field expression (e.g., calling a method on it), keep in mind that it can have a value of `null`, so you should check for that condition. An example of a Java expression that checks for a `null` value is:

```
($F{myField} != null) ? $F{myFiled}.doSomething() : null
```

This method is generally valid for all the objects, not just fields. Using Groovy or JavaScript this is rarely a problem since those languages handle a `null` value exception in a more transparent way, usually returning an empty string. This is another reason why I recommend that you use Groovy or JavaScript instead of Java.

In **Chapter 10** we'll see that in some cases a field can be a complex object, like a JavaBean, not just a simple value like a String or an Integer. A trick to convert a generic object to a String is to concatenate it to a empty string in this way:

```
$F{myfield}+ “”
```

All Java objects can be converted into a string; the result of this expression depends on the individual object implementation (specifically, by the implementation of the `toString()` method). If the object is `null`, the result will return the literal text string “null”as a value.

### 6.1.1 Registration of the Fields from a SQL Query

The most common way to fill a report is by using an SQL query. iReport provides several tools to work with SQL, including a query designer, and a way to automatically retrieve and register the fields derived from a query in the report.

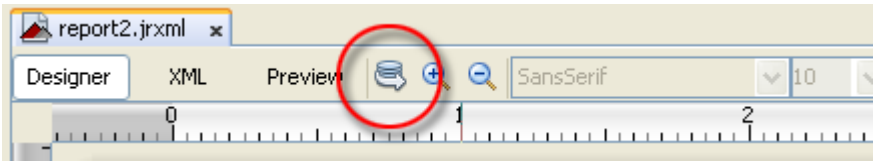



**Figure 6-4 Query Dialog Button**

You can open the query dialog by clicking the cylinder icon in the designer tool bar (**Figure 6-4**).

Before you open the query dialog, however, pay attention to the active connection/data source (the selected item in the combo box located in the main tool bar). All the operations performed by the tools in the query dialog will use that data source, so if you want to work with a particular JDBC database, or with some other kind of data source, make sure that you select the correct connection/data source in the combo box.

The report query dialog includes four query tools, each accessed by the appropriate tab, as shown in **Figure 6-5**:

- **Report query**
- **JavaBean Datasource**
- **DataSource Provider**
- **CSV Datasource**

Right now I'm going to describe for you the features of the **Report query** tab. Here you can define a query that iReport will use when generating a report.

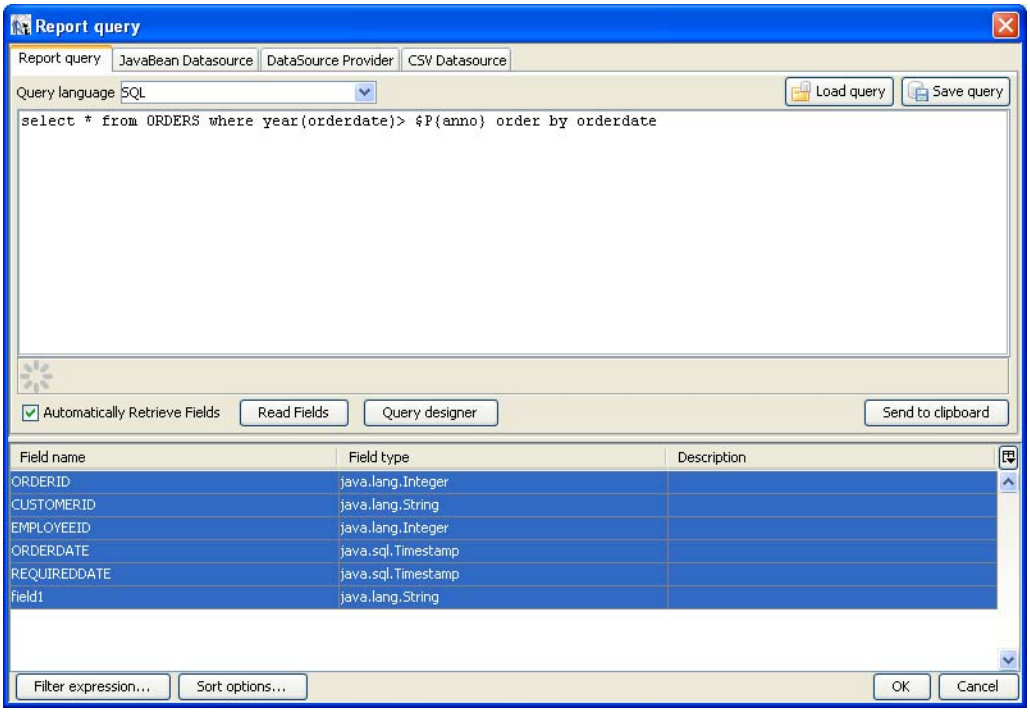



**Figure 6-5 Query Dialog**

iReport does not need you to define a query in order to generate a report. In fact, iReport could obtain data records from a data source that is not produced by a query execution. Regardless, here is where you define it. The language of the query can be one of those items listed in the combo box on the top of the query dialog. JasperReports supports the most common query languages:

- SQL
- HQL
- EJBQL
- Xpath
- MDX (both the standard and XMLA-encapsulated versions)

Let's focus on SQL. If the selected data source is a JDBC connection, iReport will test the access connection to the data source as you define the query. This allows iReport to identify the fields using the query metadata in the result set. The design tool lists the discovered fields in the bottom portion of the window. For each field, iReport determines the name and the Java type specified for that field by the JDBC driver.

A query that accesses one or more tables containing a large amount of data may require a long delay while iReport scans the data source to discover field names. You may want to disable the **Automatically Retrieve Fields** in order to quickly finish your query definition. When you have completed the query, click on the **Read Fields** button in order to start the fields discovery scan.

All fields used in a query must have a unique name. Use alias field names in the query for fields having the same name.

In case of an error during the query execution (due to a syntax error or to an unavailable database connection), an error message will be displayed instead of the fields list.

The field name scan may return a large number of field names if you are working with complex tables. I suggest that you review the list of discovered names and remove any fields that you are not planning to use in your report, in order to reduce unnecessary complexity.When you click the **OK** button all the fields in the list will be included in the report design. You can

also remove them later in the outline view, but it's a good idea at this point in the design process to remove any field names that you won't be using.

### 6.1.2 Accessing the SQL Query Designer

iReport provides a simple visual query designer to help you create simple SQL queries without having to know that particular language. You can access the tool by clicking the button labeled **Query designer** (a JDBC connection must be active, and the selected language of the query must be SQL).

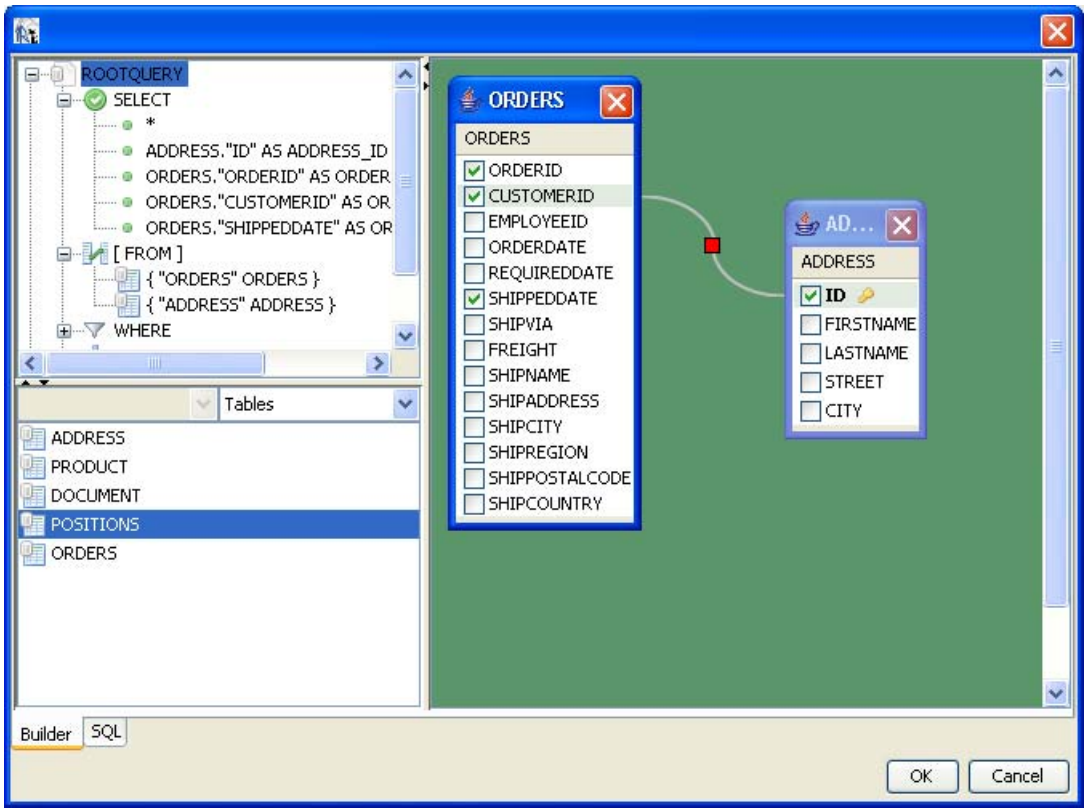



**Figure 6-6 SQLeonardo Query Designer**

This SQL query designer, which comes from the SQLeonardo open source project, provides a drag-and-drop way to create queries (see **Figure 6-6**).

To create a query you need to drag the required tables into the main panel. Check which fields you will need. The designer allows you to define table joins. To join two tables, drag the field of one table over the field of another. Edit the join type by right-clicking the red, square joint icon in the middle of the join line. To add a condition, right-click the **Where** node in the query tree. To add a field to the **Group By** and **Order By** nodes, right-click a field under the **SELECT** node.

### 6.1.3 Registration of the Fields of a JavaBean

One of the most advanced features of JasperReports is the capability to manage data sources that are not based on simple SQL queries. One example of this is JavaBean collections. In a JavaBean collection each item in the collection represents a record. JasperReports assumes that all objects in the collection are instances of the same Java class. In this case the "fields" are the object attributes (or even attributes of attributes).

By selecting the **JavaBean Datasource** tab in the query designer, you can register the fields which correspond to the specified Java classes. The concept here is that you will know which Java classes correspond to the objects that you will be using in your report.

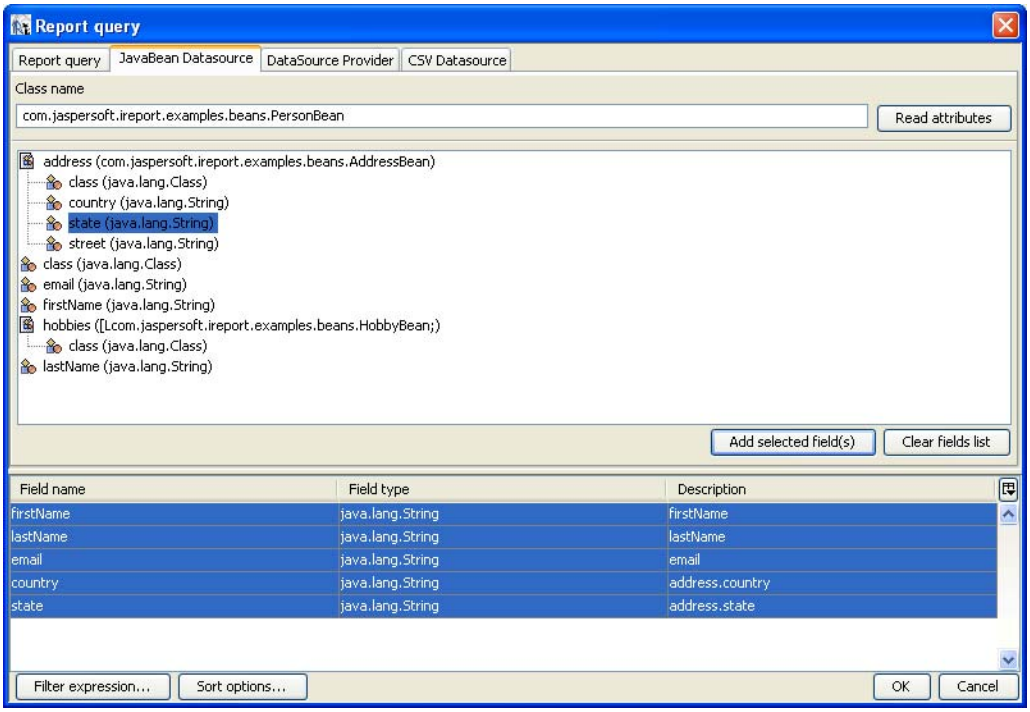



**Figure 6-7 JavaBeans Data Source Fields Registration**

Suppose that you are using objects with this Java class:

```
com.jaspersoft.ireport.examples.beans.PersonBean
```

To register fields for this class:

1. Put the class name into the class name textfield and click **Read attributes**. iReport will scan the class.
2. Check the scan results to make sure that iReport has captured the correct object attributes for that class type.
3. Select the fields you want to use in your report and press **Add selected field(s)**.

iReport now creates new fields corresponding to the selected attributes and adhesion to the list. The description, in this case, will be used to store the method that the data source must invoke in order to retrieve the value for the specified field.

iReport parses a description such as `address.state` (with a period character between the two attributes) as an attribute path. This attribute path will be passed to the function `getAddress()` in order to locate the target attribute, and then to `getState()` in order to query the status of the attribute. Paths may be arbitrary long, and iReport can recursively parse attribute trees within complex JavaBeans and in order to register very specific fields.

We have just discussed the two tools used most frequently to register fields, but we're not done yet. There are many other tools you can use to discover and register fields, for instance the HQL and XML node mapping tools. These will be discussed in Chapter **10, "Data Sources and Query Executors," on page 151**.

### 6.1.4 Fields and Textfield

To print a field in a text element, you will need to set the expression and the textfield class type correctly. You may also need to define a formatting pattern for the field. For more details about format patterns see Section **5.13, "Text Fields," on page 87**.

To create a corresponding textfield, just drag the field you wish to display from the Report Inspector view into the design panel. iReport will create a new textfield with the correct expression (e.g., `$F{fieldname}`) and assign the correct textfield class name.

## 6.2 Working with Parameters

Parameters are values usually passed to the report from the application that originally requested it. They can be used for configuring features of a particular report during report generation (such as the value to use for a condition in a SQL query) and to supply additional data to the report that can not properly provided by the data source (e.g., a custom title for the report, an application-specific path to search for images, or even a more complex object like an image).

Just as with other report objects, parameters must have a class type and must be declared in the report design (see **Figure 6-8**). The type of the parameters may be arbitrary, but the parameter name must be unique.

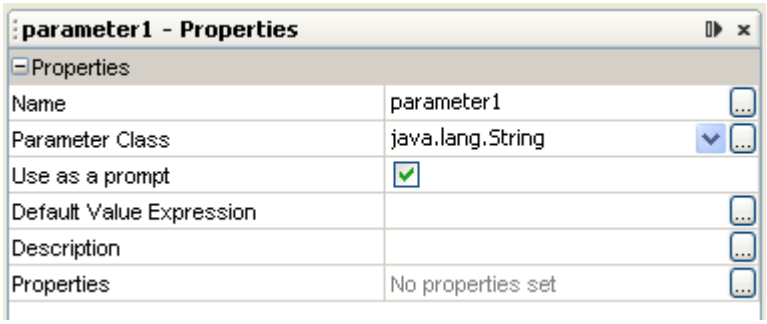



**Figure 6-8 Parameter Definition**

The property `Use as prompt` is not used directly by JasperReports. It is a special flag for the parameter that may be used by external applications; you can examine the report template to discover what parameters you should use to prompt for input.

The `Default Value Expression` permits you to set a pre-defined value for the parameter. This value will be used if no value is provided for that parameter from the application that executes the report. The type of value must be match the type declared in `Parameter Class`. In this particular expression is not possible to use fields and/or variables, since the value of the parameter must be set before fetching the first record from the data source.

You may legally define another parameter as the value of `Default Value Expression`, but using this method requires careful report design: iReport parses parameters in the same order in which they are declared, which requires that such a default value parameter must be declared before the current parameter. I realize this sounds a bit tricky, but it can be useful to employ a parameter that depends on another one, especially if we want to process it.

Let's look at an example. Suppose we have a query like this one:

```
SELECT * from ORDERS where ORDERDATE in between $P{DATE_START} and $P{DATE_END}
```

This query uses two parameters: `DATE_START` and `DATE_END`, both declared as `java.util.Date` (the object that represents a date in Java).

Let's say that we want the user to choose a period without discrete start and end dates, but just choose between relative values like “This month” or “Today”. We will set this text string value with a third parameter (let's call it `PERIOD`) that we will assign to contain the input date value from the user or the application requesting the report. In the default parameter expression for the parameters `DATE_START` and `DATE_END`, we will use the value of the parameter `PERIOD` to calculate the value of `DATE_START` and `DATE_END` relative to the value of `PERIOD`. Assuming we have a Java class able to calculate this Date, the default expression parameter for `DATE_START` could be something like:

```
MyPeriodCalculator.getDateStart( $P{PERIOD} )
```

So we have found a way to set the default value for a parameter dynamically, based on another parameter. I'm not going to provide example Java code for the hypothetical class `MyPeriodCalculator`, however, since designing specific function calls is beyond the scope of this book.

In the next section we will see how to use a parameter in an SQL query to specify not just the value of a parameter, but a piece or even the whole SQL query. This would allow you to dynamically create an input-dependent query to be stored in a parameter.

The parameter `Description` is another attribute that is not used directly by JasperReports, but like the `Use as a prompt` attribute, maybe passed to an external application.

Finally, just as with fields, you may specify pairs of type name/value as properties for each parameter. This is just a way to add extra information to the parameter, information that will be used by external applications. For example, the designer can use properties to include the description of the parameter in different languages, or perhaps add instructions about the format of the input prompt.

## 6.3 Using Parameters in a Query

Generally, parameters can be used in the query associated with a report (even if not all the languages support them). In this chapter we will focus on using parameters in SQL queries, probably the most common type of query.

Suppose we have a report that displays information about a customer. When we generate the report we would need to specify the ID of the customer to display. In order to get data regarding the customer we need to pass this information to the query; in other words, we want to filter the query to obtain only the data relevant to a particular customer ID. The query will look like:

```
select * from customers where CUSTOMERID = $P{MyCustomerId}
```

`MyCustomerID` is a parameter, and the syntax to use for the parameter in the query is the same used when referring to a parameter in an expression. Without going into too depth on how parameters work in SQL, let's just say that JasperReports will interpret this query as:

```
select * from customers where CUSTOMERID = ?
```

The question mark character is the canonical symbol in SQL that says “here goes a value provided to the SQL statement before query execution.” This is exactly what JasperReports will do: it will execute the query, passing the value of each parameter used in the query to the SQL statement.

This approach has a big advantage with respect to concatenating the mere parameter value to the query string: you do not have not to take care of special characters or to sanitize your parameter, since the database will do it for you. At the same time this method places a strong limit on the control you have on the query structure. For example, you cannot specify a portion of a query with a parameter (e.g., storing the entire `WHERE` or `GROUP BY` clause). The solution is to use the special syntax:

```
$P!{<parameter name>}
```

Note to the exclamation mark just after the `$P`. The exclamation mark notifies JasperReports to not bind the parameter to an SQL parameter (using the question mark (`?`) like in the previous case), but rather to calculate the value of the parameter and evaluate it as a raw sub-section of a query. For example, if you have a parameter named `MyWhere` with the value of "`where CUSTOMERID = 5`", the query:

```
select * from customers  $P!{CUSTOMERID}
```

will be transformed into:

```
select * from customers  where CUSTOMERID = 5
```

without using the logic of the SQL parameter. The drawback is that you must be 100% sure that the parameter value is correct in order to avoid an error during the query execution.

### 6.3.1 IN and NOT IN clause

JasperReports provides a special syntax to use with a `where` condition: the clause `IN` and `NOT IN`.

The clause is used to check whether a particular value is present in a discrete set of values. Here is an example:

```
SELECT * FROM ORDERS WHERE SHIPCOUNTRY IS IN ('USA','Italy','Germany')
```

The set here is defined by the countries USA, Italy and Germany. Assuming we are passing the set of countries in a list (or better a `java.util.Collection`) or in an array, the syntax to make the previous query dynamic in reference to the set of countries is:

```
SELECT * FROM ORDERS WHERE $X{IN, SHIPCOUNTRY,  myCountries}
```

where `myCountries` is the name of the parameter that contains the set of country names. The `$X{}` clause recognizes three parameters:

- Type of function to apply (`IN` or `NOT IN`)
- Field name to be evaluated

- Parameter name

JasperReports will handle special characters in each value. If the parameter is null or contains an empty list, meaning no value has been set for the parameter, the entire `$X{}` clause is evaluated as the always true statement "`0 = 0`".

## 6.4 Built-in Parameters

JasperReports provides some built-in parameters (they are internal to the reporting engine) that you may read but cannot modify. These parameters are presented in Table 6-1.

**Table 6-9Built-In Parameters**

| | |
|---|---|
| `REPORT_PARAMETERS_MAP` | This is the `java.util.Map` passed to the `fillReport` method, and it contains the parameter values defined by the user. |
| `REPORT_CONNECTION` | This is the JDBC connection passed to the report when the report is created through a SQL query. |
| `REPORT_DATASOURCE` | This is the data source used by the report when it is not using a JDBC connection. |
| `REPORT_SCRIPTLET` | This represents the scriptlet instance used during creation. If no scriptlet is specified, this parameter uses an instance of `net.sf.jasperreports.engine.JRDefaultScriptlet`.* |
| `IS_IGNORE_PAGINATION` | You can switch the pagination system on or off by setting a value for this parameter (it must be a Boolean object). By default, pagination is used. It is not used when exporting to HTML or Excel formats. |
| `REPORT_LOCALE` | This is used to set the locale used to fill the report. If no locale is provided, the system default will be used. |
| `REPORT_TIME_ZONE` | This is used to set the time zone used to fill the report. If no locale is provided, the system default will be used. |
| `REPORT_MAX_COUNT` | This is used to limit the number of records filling a report. If no value is provided, no limit will be set. |
| `REPORT_RESOURCE_BOUNDLE` | This is the resource bundle loaded for this report. See **Chapter 10** for how to provide a resource bundle for a report. |
| `REPORT_VIRTUALIZER` | This defines the class for the report filler that implements the `JRVirtualizer` interface for filling the report. |
| `REPORT_FORMAT_FACTORY` | This is an instance of a `net.sf.jasperreports.engine.util.FormatFactory`. The user can replace the default one and specify a custom version using a parameter. Another way to use a particular format factory is by setting the report property *format factory class*. |
| `REPORT_CLASS_LOADER` | This parameter can be used to set the class loader to use when filling the report. |
| `REPORT_URL_HANDLER_FACTORY` | Class used to create URL handlers. If specified, it will replace the default one. |
| `REPORT_FILE_RESOLVER` | This is an instance of `net.sf.jasperreports.engine.util.FileResolver` used to resolve resource locations that can be passed to the report in order to replace the default implementation. |
| `REPORT_TEMPLATES` | This is an optional collection of styles (`JRTemplate`) that can be used in the report in addition to the ones defined in the report. |

*.Starting with JasperReports 1.0.0, an implicit cast to the provided scriptlet class is performed. This simplifies many expressions that use the provided scriptlet class.

## 6.5 Passing Parameters from a Program

iReport passes parameters from a program "caller" to the print generator using a class that extends the `java.util.Map` interface. Consider the code in **Figure 3-4 on page 42** in **Chapter 3**, in particular the following lines:

**Figure 6-10** `JasperTest.java` **- Excerpt**

```
...
 HashMap hm = new HashMap();
   ...
 JasperPrint print = JasperFillManager.fillReport(
    fileName,
    hm,
  new JREmptyDataSource());
 ...
```

`fillReport` is a key method that allows you to create a report instance by specifying the file name as a parameter, a parameter map, and a data source. (This example uses a dummy data source created with the class `JREmptyDataSource` and an empty parameter map created using a `java.util.HashMap` object.

Let's see how to pass a simple parameter to a reporting order to specify the title of a report.The first step is to create a parameter in the report to host the title (that will be a String). We can name this parameter `REPORT_TITLE` and the class will be `java.lang.String` (see **Figure 6-11**).

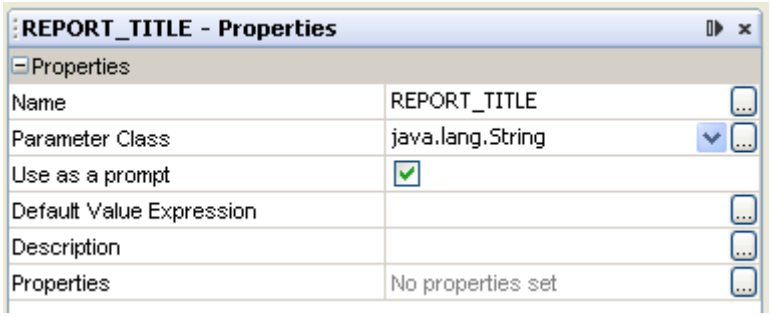



**Figure 6-11 Definition of the parameter to host the title**

All the other properties can be left as they are; in particular we don't want set a default value expression.By dragging the parameter into the title band we can create a textfield to display the `REPORT_TITLE` parameter (shown in **Figure 6-12**)

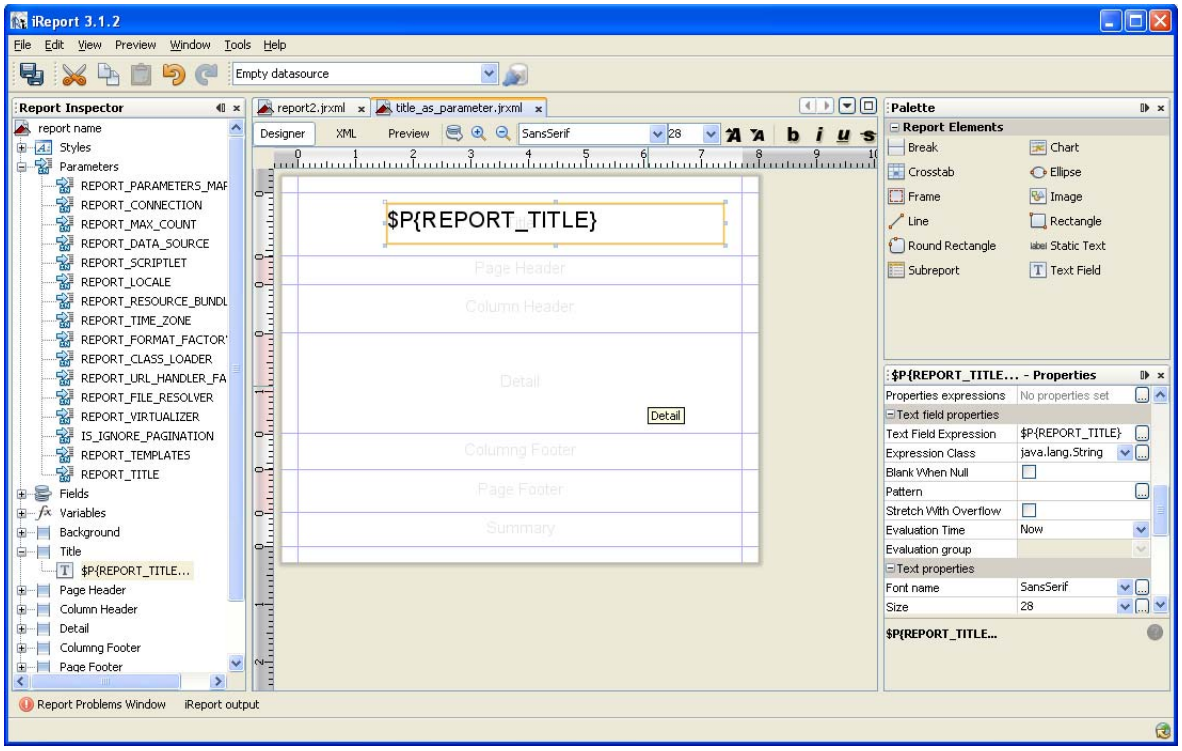



**Figure 6-12 Design panel with the `REPORT_PARAMETER` displayed in the title band**

This example report is very simple. We just want focus our attention on how to display the parameter.

To set the value for the `REPORT_TITLE` parameter in our application, just modify the code of the previous source code example to add:

```
...
HashMap hm = new HashMap();
hm.put("REPORT_TITLE","This is the title of the report");
...
JasperPrint print = JasperFillManager.fillReport(
fileName,
hm,
new JREmptyDataSource());
...
```

We have included a value for the `REPORT_TITLE` parameter in the parameter map. You do not need to pass a value for all the parameters. If you don't provide a value for a certain parameter, JasperReports will assign the value of `Default Value Expression` to the parameter with the empty expression evaluated as `null`.

Printing the report, iReport includes the String `This is the title of the report` in the title band. In this case we just used a simple String. However, it is possible to pass much more complex objects as parameters, such as an image (`java.awt.Image`) or a data source instance configured to provide a specified subreport with data. The only very important thing to remember is that the object passed in the map as the value for a certain parameter must have the same type (or at least be a super class) of the type of our parameter in the report. Otherwise, iReport fails to generate the report, returning a `ClassCastException` error. This is actually pretty obvious: you cannot, for example, assign the value of a `java.util.Date` to a a parameter declared as an Integer.

## 6.6 Working with Variables

Variables are objects used to store the results of calculations such as subtotals, sums, and so on. Again, just as with fields and parameters, you must define the Java type of a variable when it is declared.

To add a new variable, select the variables node in the outline view and select the menu item **Add Variable.**

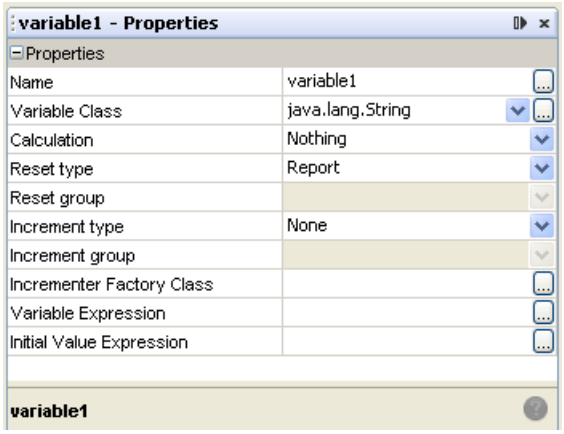



**Figure 6-13 Variable Properties**

Figure 6-13 shows the property sheet for a variable. Below is a list describing the meaning of each property.

`Name` This is the name of the variable. The name must be unique (meaning you can not have two variables with the same name). Similar to fields and parameters, you refer to a variable using the following syntax in an expression:

```
$V{<variable name>}
```

`Variable Class` This is the Java type of the variable. In the combo box, you can see some of the most common types such as `java.lang.String` and `java.lang.Double`.

`Calculation` This is the type of a pre-defined calculation used to store the result by the variable. When the predefined value is Nothing, it means “don’t perform any calculation automatically”. JasperReports performs the specified calculation by changing the variable’s value for every new record that is read from the data source. To perform a calculation of a variable means to evaluate its expression (defined in the Variable Expression property). If the calculation type is Nothing, JasperReports will just assign to the variable the value that resulted from the evaluation of the variable expressions. If a calculation type is other than Nothing, the expression result will represent a new input value for the chosen calculation, and the variable value will be the result of this calculation. The calculation types are listed in Table 6-14.

**Table 6-14Calculation types for the variables**

| Type | Definition |
|---|---|
| `Distinct Count` | This counts the number of different expression results; the order of expression evaluation does not matter. |
| `Sum` | This adds to each iteration the expression value to the variable’s current value. |
| `Average` | This calculates the arithmetic average of all the expressions received in input.From a developer’s point of view, declaring a variable to type System is pretty much like just declare a variable in a program, without actually use it. Someone will set a value for it. This “someone” is usually a scriptlet tied to the report or some other Java code executed through an expression. |

**Table 6-14Calculation types for the variables**

| Type | Definition |
|---|---|
| `Lowest` | This returns the lowest expression value received in input. |
| `Highest` | This returns the highest expression value received in input. |
| `StandardDeviation` | This returns the standard deviation of all the expressions received in input. |
| `Variance` | This returns the variance of all the expressions received in input. |
| `System` | This does not make any calculation, and the expression is not evaluated. In this case, the report engine keeps only the last value set for this variable in memory.* |

*.From a developer's point of view, declaring a variable to type `System` is pretty much like just declaring a variable in a program, without actually using it. Someone will set a value for it. This "someone" is usually a scriptlet tied to the report or some other Java code executed through an expression.

`Reset type` This specifies when a variable value has to be reset to the initial value or simply to null if no initial value expression has been provided. The variable reset concept is fundamental when you want to make some group calculations such as subtotals or averages: in that case when a group changes, you should reset the variable value and start over the calculation. The reset types are listed in Table **6-15**.

**Table 6-15Reset Types**

| Reset Type | Description |
|---|---|
| `None` | The initial value expression is always ignored. |
| `Report` | The variable is initialized only once at the beginning of report creation by using the initial value expression. |
| `Page` | The variable is initialized again in each new page. |
| `Column` | The variable is initialized again in each new column (or in each page if the report is composed of only one column). |
| `Group` | The variable is initialized again in each new group (the group specified in the Reset group setting) |

`Reset Group` This specifies the group that determines the variable reset if the Group reset type is selected.

`Increment type` This specifies when a variable value has to be evaluated; by default a variable is update every time a record is fetched from the data source. Sometimes the calculation we want to perform must be performed only at certain times. The increment types are the same as the calculation types listed in Table **6-14**. To clarify a little bit the use of increment type, consider this example: suppose to have a report with a list of names ordered alphabetically and grouped by the first letter, so we have the group for the letter A with containing a certain amount of names, the group B and so on up to Z. Suppose we want to calculate with a variable the average number of names for each letter. What we need to do is to create a variable that perform an average calculation an the number or records present in each group. To correctly perform this calculation, the variable must be updated only when the first letter in the names change, that's what happens at the end of each group. In this case the increment type (meaning the exact moment at which a new input value must be acquired to perform the calculation) should be *Group*.

| | |
|---|---|
| `Increment group` | This specifies the group that determines the variable increment if the Group increment type is selected. |
| `Custom Incrementer Factory Class` | This is the name of a Java class that implements the `JRIncrementerFactory` interface, which is useful for defining operations such as the sum for non-numerical types. In other words a developer has the ability to implements his custom calculation. |
| `Variable expression` | This is the expression that identifies the input value of the variable to each iteration. The result must be congruent to the type of the variable and its calculation; in example if we are just counting objects, the expression can return any kind of result, the variable will be incremented only when a not null result is provided, independently by the expression result type, but if we are summing something, the calculator expects an object of the same type of the variable (like a Double or an Integer). |
| `Initial value expression` | This is an expression used to set the initial value for the variable. If blank, the variable is initialized to null. |

## 6.7 Built-In Variables

As with parameters, JasperReports provides some built-in variables (which are directly managed by the reporting engine). You can read these variables but cannot modify them. Table **6-16** lists the built-in variables.

**Table 6-16Built-In Variables**

| Variable Name | Definition |
|---|---|
| `PAGE_NUMBER` | Contains the current number of pages. At “report” time, this variable will contain the total number of pages. |
| `COLUMN_NUMBER` | Contains the current number of columns. |
| `REPORT_COUNT` | Contains the current number of records that have been processed. |
| `PAGE_COUNT` | Contains the current number of records that have been processed in the current page. |
| `COLUMN_COUNT` | Contains the current number of records that have been processed during the current column creation. |
| `<group name>_COUNT` | Contains the current number of records that have been processed for the group specified as a variable prefix. |

## 6.8 Evaluating Elements During Report Generation

There is a strict correlation between the physical location of an element in a report and the point at which JasperReports evaluates the element during report generation. When the report filling process starts, JasperReports retrieves the first record from the data source, updates the value of the fields and recalculates the necessary variables. The first band to be evaluated is the title, followed by the page header, the column header, group headers, detail, and so on. When all the detail bands have been filled, the engine next retrieves the second record, updating all the fields and variables again, and continues the fill process. By

default, the engine evaluates the expression of textfield and image elements as they are encountered during the report generation process.

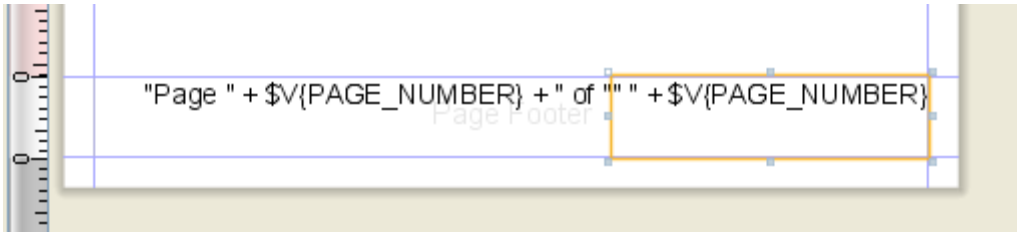



**Figure 6-17 How to print Page X of Y using the evaluation time**

Sometimes this is not what the we want. An example is when we want to show the final result of a calculation in a textfield that is evaluated before the end of the calculation, like printing the total number of pages in the report in each page footer (as in the example shown in **Figure 6-17**). There is no variable that contains the total number of pages in the report, there is just the `PAGE_NUMBER` variable that defines the value of the current page number. In this case we have to force JasperReports to wait to fill the particular element until the calculation process completes.

This is how you can set the evaluation time for textfields and images: by delaying the evaluation of their expression. Returning to the example of Page X of Y, we need two textfields:

- First, to print the current page (or better the string “Page X of”), where X is the current value of the variable `PAGE_NUMBER`
- Second, another textfield to print Y (the total number of pages)

For this last textfield we set the evaluation time to `REPORT`, which signifies “when the last page has been reached.”At that point the value of `PAGE_NUMBER` will contain the total number of pages.

You can use the same method to print a subtotal in the header of a group. The calculation reliquaries the records in the group to be processed first, but it is possible to place a textfield showing the variable associated with the calculation in a textfield of the group header, with the evaluation time set to `GROUP` (and the evaluation group to the proper group).

There are two particular evaluation times for textfields that deserve attention: evaluation time `BAND` and evaluation time `AUTO`.

The evaluation time `BAND` forces JasperReports to evaluate a variable derived from a calculation performed after processing the entire band. This is usually used in the detail band in two cases:

- Including the value returned from a subreport (e.g., the number of records printed in the subreport)
- Using the value of a variable is set by an external agent, such as a scriptlet

The evaluation time `AUTO` allows you to mix values that must be determined at different evaluation times, like the current value of a field and a variable. The most common case is when we want to calculate a percentage. Suppose we have a list of numbers, and we want to print the percentage of incidence for each single number with respect to the total of all the numbers. You can calculate the percentage bedevilling the value of a specific number at evaluation time `NOW` by the variable that calculates the total with an evaluation time equal towny the last record is processed that will be required to calculate the entire sum (i.e. `REPORT` or `GROUP`). In other words, an evaluation time corresponding to its reset type.

Here comes the conflict: we need to consider two values having different evaluation times. The `AUTO` evaluation time provides the solution. JasperReports will use the evaluation time `NOW` for the fields participating in the expression, while waiting to evaluate the variables until the evaluation time occurs that corresponds to their reset type.

# 7 Bands and Groups

In this chapter, I will explain how to manage bands and groups when using iReport. In **Chapter 4**, you learned how reports are structured, and you have seen how the report is divided into bands, horizontal portions of a page that are printed and modified in height according to band properties and content. Here, you will see how to adjust the properties of these bands. You will also learn how to use groups, how to create some breaks in a report, and how to manage subtotals.

## 7.1 Modifying Bands

JasperReports divides a report in eight main bands and the background (plus the special `No Data` band). To this set of standard bands you can include two supplemental bands for each group: the `Group Header` band and the `Group Footer` band.

When you select a band in the report outline view, the band properties are displayed in the property sheet (Figure 7-1).

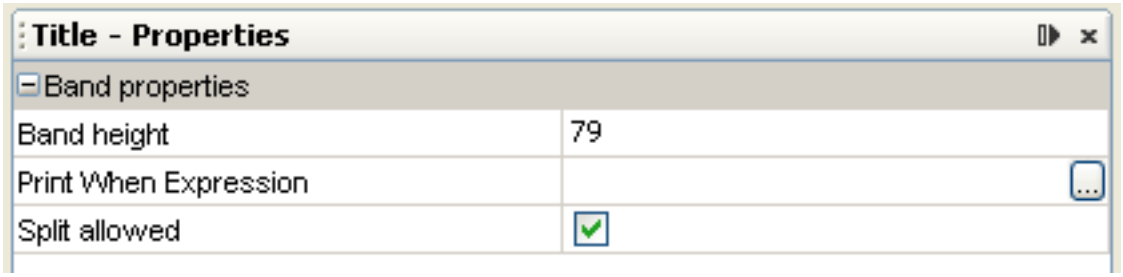



**Figure 7-1 Band Properties**

The band height represents the height of the band at design time. If the content of the band expands vertically (i.e. due to a subreport or a long text in a textfield with the `Stretch with Overflow` property set to true), the band increases in height accordingly during the report execution. The band height is expressed in pixels (always using the same resolution of 72 pixels per inch). You can set the height with the property sheet or by dragging the bottom border of the band with the mouse directly into the designer window.

Consecutive zero-height bands may become obscured while working in the design panel. You can increase the height of a selected band by pressing the Shift key while dragging the bottom margin of the band down in the panel.

The `Print When Expression` is used to hide or display the band under some circumstances described by the expression. This expression must return a Boolean value. In particular, it must return `TRUE` to display the band and `FALSE` to hide it. By default, when no value is defined for that expression, the band is displayed.

JasperReports reserves enough space in a page for bands like the title, the page header and footer and the column header and footer. All the other bands cannot fit in the remaining space when repeated several times. This may result in a detail band beginning in a page and ending in another band. If you want to be ensure that a band displays completely within one page, deselect the `Split allowed` property: all times the band is printed, JasperReports will check the available space in the current page, if it is not enough, the band will start on the next page. Of course this does not mean that the band will completely feet in the next page, this still depends of the band content.

The default report template includes all the predefined bands, except the `Last Page Footer` and the `No Data` bands. If you are not interested in using a band you can remove it by right-clicking the band (or the band node in the outline view) and selecting the menu item **Delete Band**. When a band is no longer present in the report, it is displayed as grayed node (see **Figure 7-2**). To add the band to the report, right-click the band and select **Add Band**.

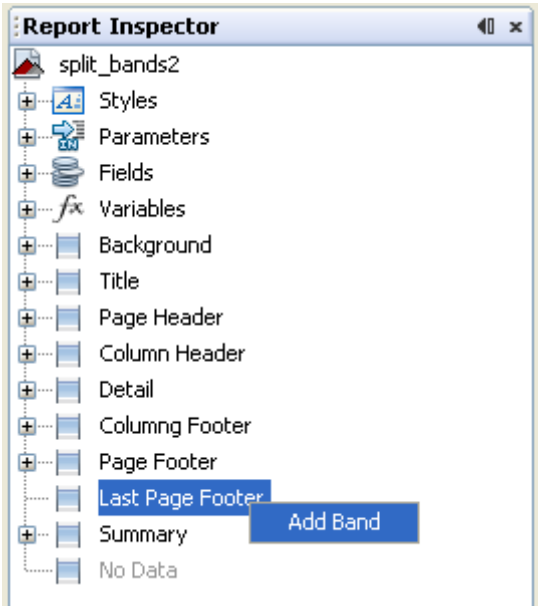



**Figure 7-2 Adding a pre-defined band**

In general, there is no valid reason to remove a band apart the generation of a less complex JRXML file (the report source code). In order to prevent the printing of a band, just set its height to 0. The only exceptions are the `Last Page Footer` and the `No Data` bands. If present, the former always replaces the `Page Footer` band in the last page, so if we don't want or need this behavior the band must be not present. The `No Data` band is a very special band that replaces the entire report if the data source does not contain any records and if at the document level the property `When No Data Type` has been set to `No Data Section`.

## 7.2 Working with Groups

Groups allow you to organize the records of a report in order to create some structures. A group is defined through an expression. JasperReports evaluates this expression thus: a new group begins when the expression value changes. An

expression may be represented just by a specific field (i.e. you may want to group a set of contacts by city, or country), but it can be more complex as well. For example, you may want to group a set of contact names by initial letter.

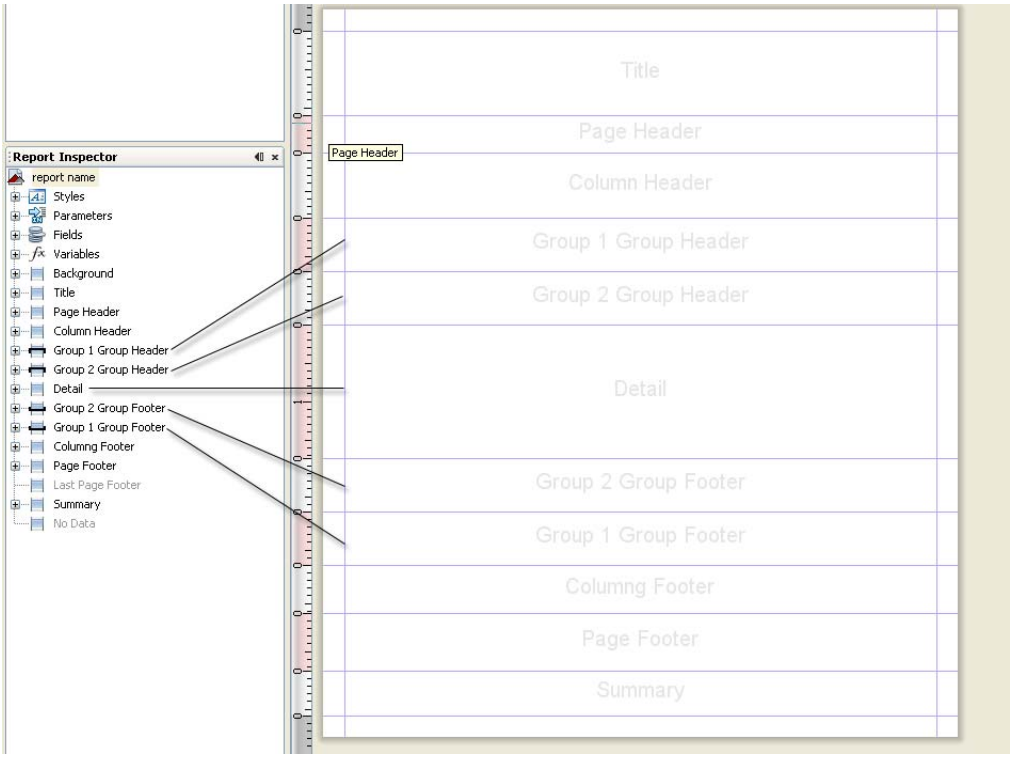



**Figure 7-3 Group Bands**

Each group can have an optional header and a footer band. Group header and footer bands are printed just before and after the detail band. You can define an arbitrary number of groups (i.e. you can have a first grouping level that groups contacts by Country and a nested group grouping the contacts in the country by City). The order of the groups is very important, the first group is the most “external”, meaning that other groups will group records inside the previous group.

The group order can be changed by right-clicking a group node (header or footer) and selecting the **Move Group Up** or **Move Group Down** menu items.

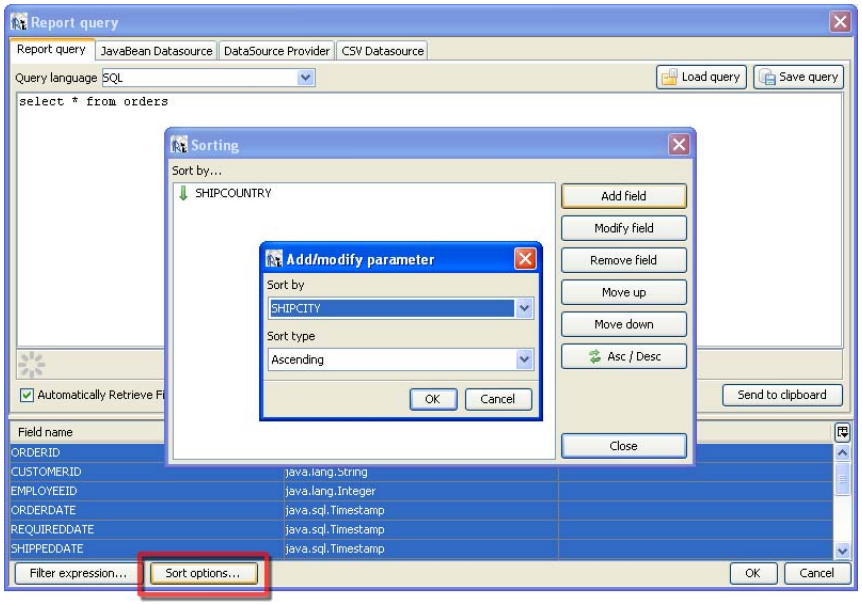



**Figure 7-4 Sorting Options**

The method with which JasperReports groups records is by evaluating the group expression. Every time its value changes, a new group instance is created. The engine does not perform any record sorting (if not explicitly requested), so when we define groups we should always take care of the records sorting. That is, if we want to group a set of addresses by country, the records we select for the report should already by ordered by country. It is simple to sort data when using an SQL query by using the `ORDER BY` clause. When this is not possible (i.e. when obtaining the records from an XML document), we can request that JasperReports sort the data for us. This can be done using the sort options available in the query window (**Figure 7-4**).

In order to use the Sort options, you must have some fields already registered in the report. Sorting can only be performed by field (you can not sort records using an expression). You can specify how many fields you need to sort your records. Each field can use a different sort type (ascending or descending). This sorting is performed in memory, so it's use is discouraged if you are working with very large amount of data, but it is very useful with a reasonable number of records (dependent by the available memory).

Let's see how groups work using an example. Suppose you have a list of people: you want to create a report where the names of these people are grouped based on the initial letter (like in a phone book). Run iReport and open a new empty report. Next, you will take the data from a database by using a SQL query (we will use the sample database provided with JasperReports) having a proper `ORDER BY` clause. For this example, use the following SQL query:

```
SELECT * FROM ADDRESS ORDER BY LASTNAME, FIRSTNAME
```

The selected records will be ordered according to the last and first name of the selected customers. The fields selected by the query should be `ID`, `FIRSTNAME`, `LASTNAME`, `STREET` and `CITY`.

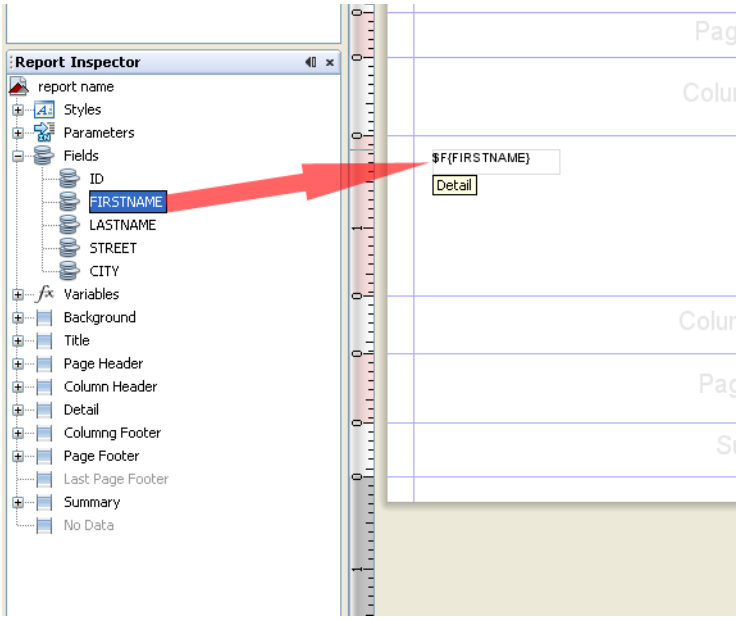



**Figure 7-5 Dragging a field into the detail band**

Before continuing on with creating your group, make sure that everything works correctly by inserting in the details band the `FIRSTNAME`, `STREET` and `CITY` fields (move them from the outline view to the detail band, as shown in **Figure 7-5**):

Then create a layout similar to the one proposed in **Figure 7-6** and preview the report:


**Figure 7-6 Layout before adding the groups**

The result should be similar to that of **Figure 7-7**.

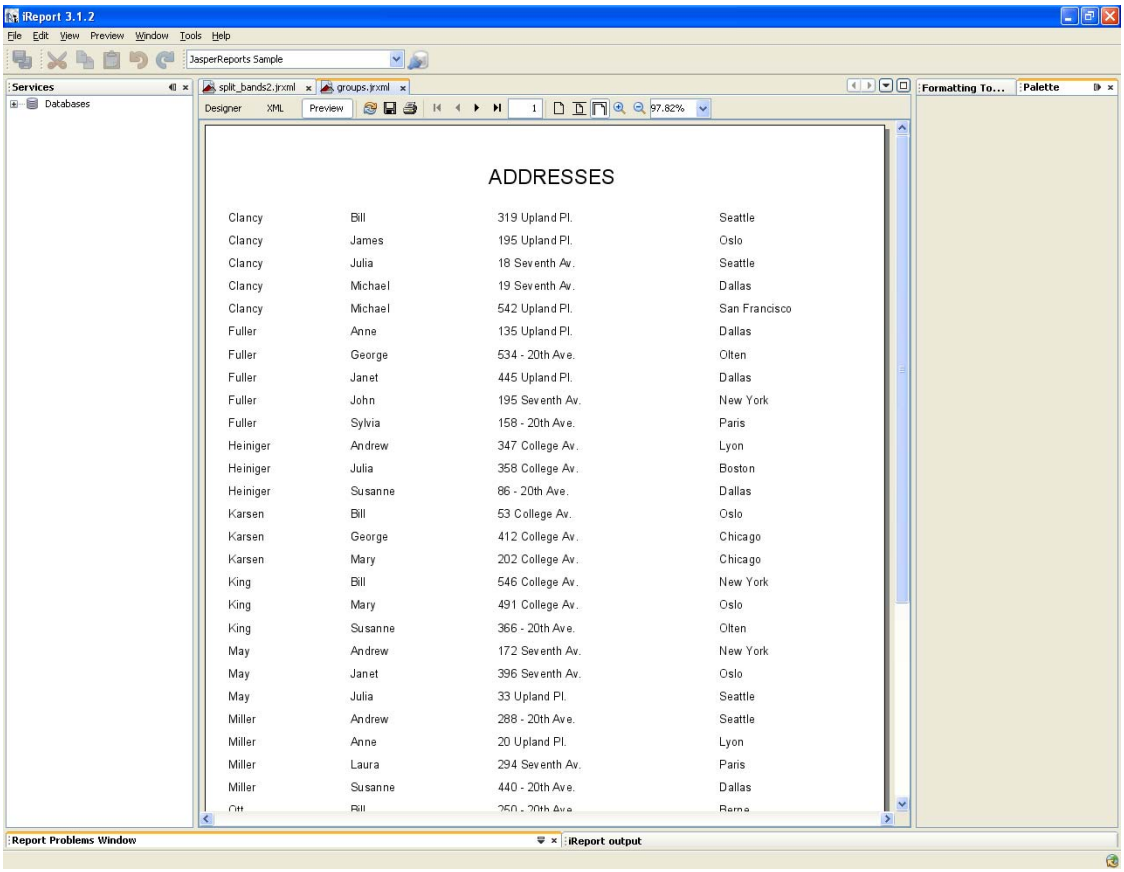



**Figure 7-7 The addresses not grouped**

What we have is just a simple flat report showing an ordered list of addresses. Let's proceed to group all these records by the first letter of the contact last name. The first letter of the first name can be extracted with a simple expression (both in Groovy and JavaScript). Here is:

```
$F{LASTNAME}.charAt(0)
```

If you use Groovy or JavaScript as suggested, remember to set it in the document properties.

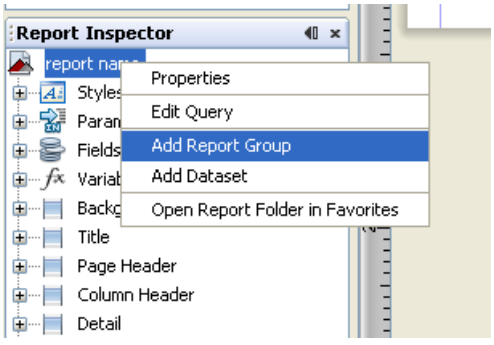



**Figure 7-8 Add Report Group**

To add the new group, select the document root node in the outline view an select the **Add Report Group** menu item (**Figure 7-9**).

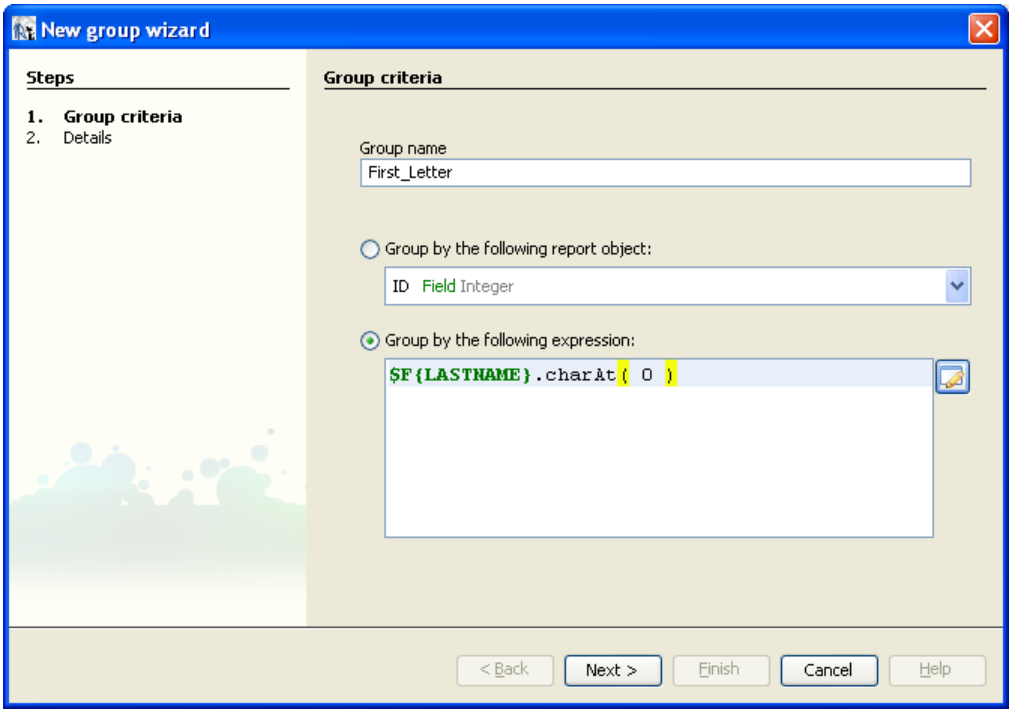



**Figure 7-9 The first step of the new group wizard**

This opens a simple wizard to create the group (**Figure 7-9**). Set the group name (i.e. `First_Letter`) and use the expression we have seen to extract the first letter from a string.

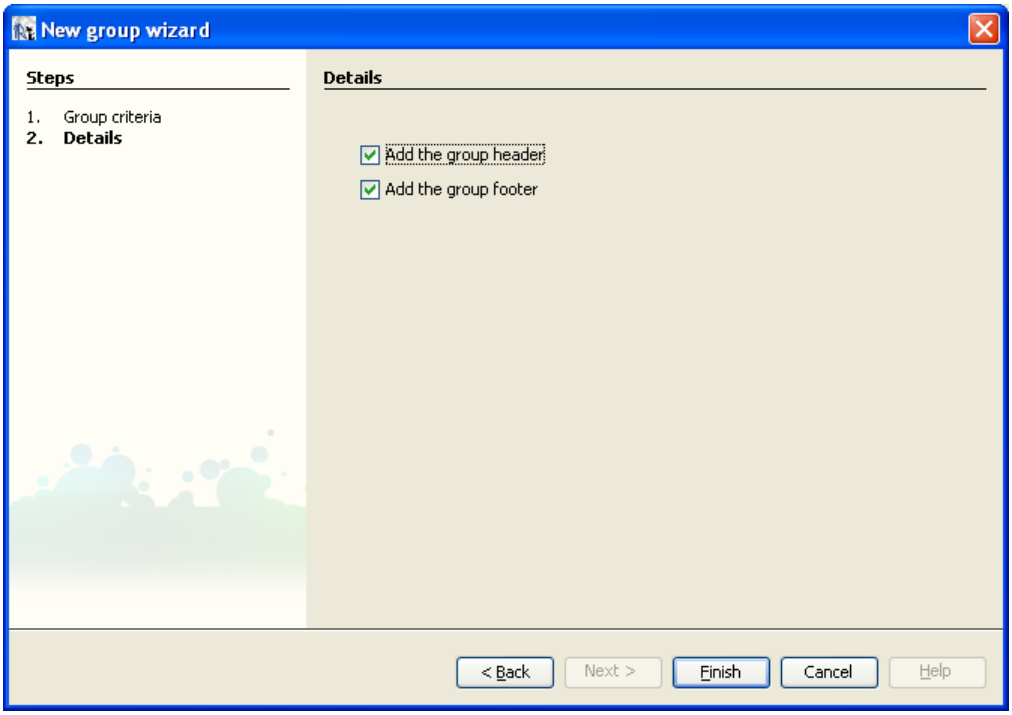



**Figure 7-10 The second step of the new group wizard**

In the second step (**Figure 7-10**) we have the option to create a the header and the footer bands for the group. Press **Finish** to complete the group creation.

The new two bands (group header and footer) will appear in the design window, and the corresponding nodes will be added to the report structure in the outline view.

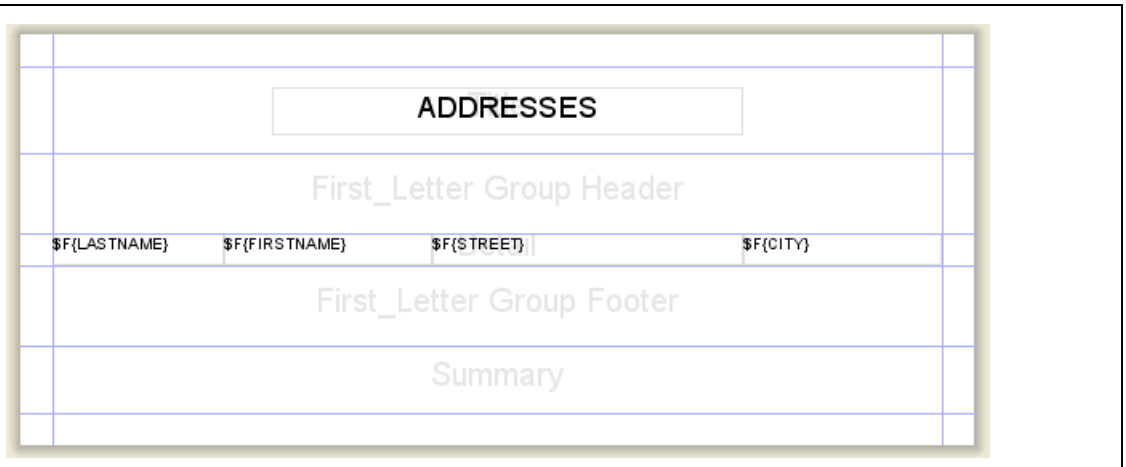



**Figure 7-11 The new group bands in the design panel**

When you add a group to the document, iReport creates an instance of the built-in variable <*group name*>_`COUNT` for the new group. In our case the variable is named `First_Letter_COUNT`. It represents the number of records processed for the group, so if we display this variable in the group footer using a textfield, it will display how many records the group contains.

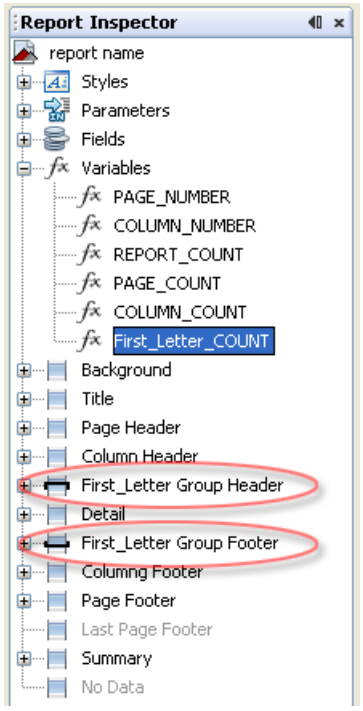



**Figure 7-12 The group in the outline view**

Now we can add some content to the group header and footer. In particular, we can add the initial letter to which the group refers too, and in the footer the `First_Letter_COUNT` variable. For the letter, just add a Textfield in the group header and use the same textfield expression as you did for the group. The textfield class can be set to String (because we are using Groovy or JavaScript). If you use Java, the expression for the textfield should be changed a little bit, Java is a bit more severe in terms of type matching, and since the `charAt()` function returns a char, we can convert this value in a String by

concatenating an empty String. (This is actually a dirty but simple way to cast any Java object in a String without checking if the object is `NULL`). So the expression in Java should be:

```
"" +$F{LASTNAME}.charAt(0)
```

**Figure 7-13** shows the final layout:


**Figure 7-13 Final Layout**

The blue field (in the group footer) displays the variable `First_Letter_COUNT` that we created by dragging this variable from the outline view into group footer band. If we want to display the same value in the group header, we need to change the textfield evaluation time to `Group` and set the evaluation group to `First_Letter`. See Section **6.8, "Evaluating Elements During Report Generation," on page 108** for a discussion of evaluation times.

**Figure 7-14** shows the final result.

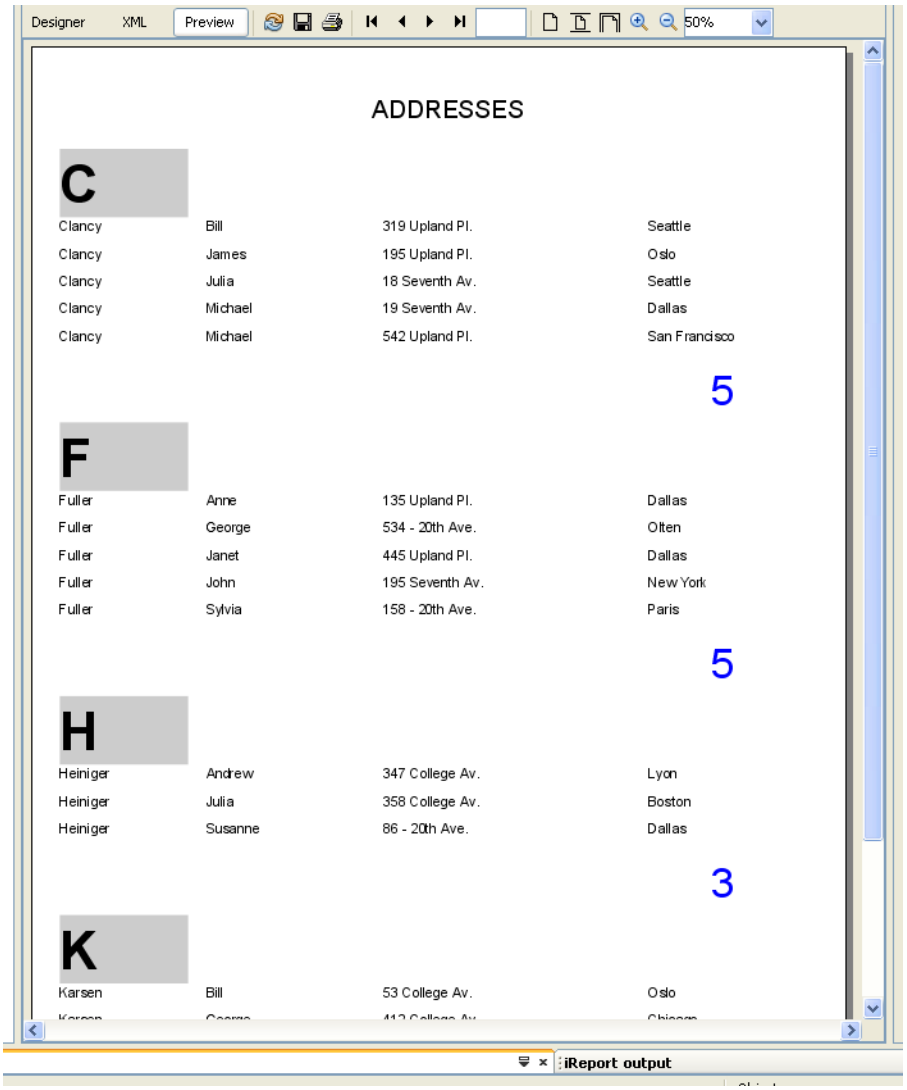



**Figure 7-14 The final result**

## 7.3 Other Group Options

In the previous sample we learned how to create a group using the group wizard, set the group name, and set the group expression. There are many other options that you can be set to control how a group displays in a report. By selecting a group band in the outline view (header or footer), in the property sheet you will see all these options (**Figure 7-15**).

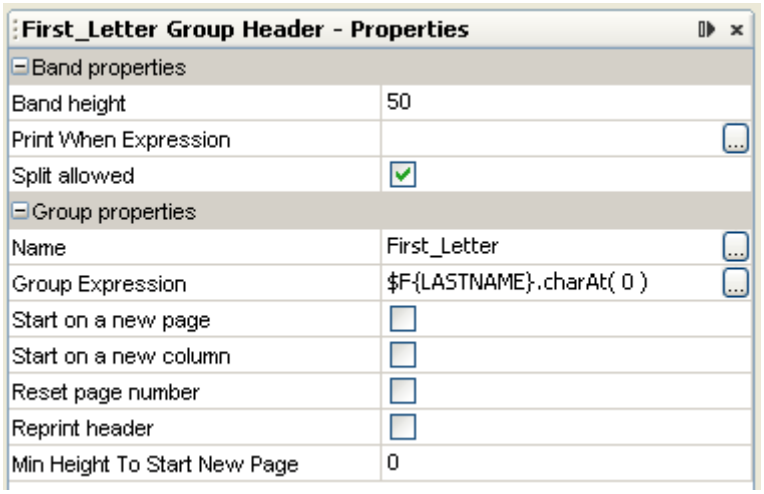



**Figure 7-15 Group Options**

`Group Expression` — This is the expression that JasperReports will evaluate against each record. When the expression changes in value, a new group is created. If this expression is empty, it is equal to `NULL` and since a null expression will never change in value the result is a single group header and a single group footer, respectively, after the first column header and before the last column footer.

`Start on a New Column` — If this option is selected, it forces a column break at the end of the group (that is, at the beginning of a new group); if in the report there is only one column, a column break becomes a page break.

`Start on a New Page` — If this option is selected, it forces a page break at the end of the group (that is, at the beginning of a new group).

`Reset Page Number` — This option resets the number of pages at the beginning of a new group.

`Print header on each page` — If this option is selected, it prints the group header band on all the pages on which the group's content is printed (if the content requires more than one page for the printed report).

`Min height to Start New Page` — If the value is other than 0, JasperReports will start to print this group on a new page, if the available space remained in the current page is inferior to the minimum specified. This option is usually used to avoid to span on two pages a report section composed of more fields that we want to remain together (such as a title followed by the text of a paragraph).

# 8 Fonts and Styles

Fonts describe the characteristics (shape and dimension) of text. In JasperReports, you can specify the font properties for each text element.

You can save time defining the look of your elements, included all the font settings by using styles. A style is a collection of predefined properties that refer to aspects of elements (like background color, borders, and font). You can define a default style for your report that all undefined properties of your elements refer to by default.

## 8.1 Working with Fonts

Usually a font is defined by the following basic characteristics:

- Font name (font family)
- Font dimension
- Attributes (bold, italics, underlined, barred)

If plan to export a report as a PDF file, JasperReports requires the following additional information:

| | |
|---|---|
| *PDF font name* | The name of the font (it could be a predefined PDF font or the name of a TTF file present in the classpath) |
| *PDF embedded* | A flag that specifies whether an external TrueType font (TTF) file should be included in the PDF file |
| *PDF encoding* | A string that specifies the name of the character encoding |

If the report is not exported to PDF format, the font the report engine uses is the one specified by font name and enriched with the specified attributes. In the case of a PDF document, the PDF font name identifies the font used. The report engine ignores the Bold and Italics attributes when exporting a report as a PDF since these particular characteristics are part of the font itself, and can not overridden. If you take a look at the list of pre-defined PDF fonts you will see something like:

- Helvetica
- Helvetica-Bold
- Helvetica-BoldOblique
- Helvetica-Oblique

and so on. So if your report includes text formatted with Helvetica, and the textfield must be rendered in bold, you have to choose from the PDF font name the item *Helvetica-Bold*.

The *Underline* and *Strike Through* attributes can be used without extra exceptions.

## 8.2 Using TTF Fonts

You can use an external TrueType font. To do so, the external fonts (files with .ttf extensions) must appear in the classpath. Any TrueType fonts you use must be available both as you design the report in iReport and whenever the JasperReports report engine generates an instance of the report, such as when a servelet or Java or Swing program prints a version of it.

In the **Font name** combo box in the property sheet for static and text fields, only the system fonts, managed by the Java Virtual Machine (JVM), are shown (see Figure 8-1). These are usually inherited by the operating system. Therefore, must install an external TrueType font on your system before use it in non-PDF reports.

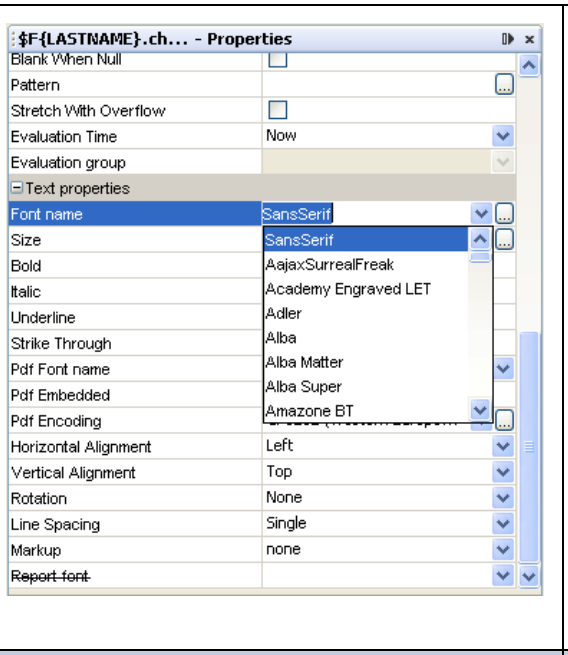



**Figure 8-1 System Fonts**

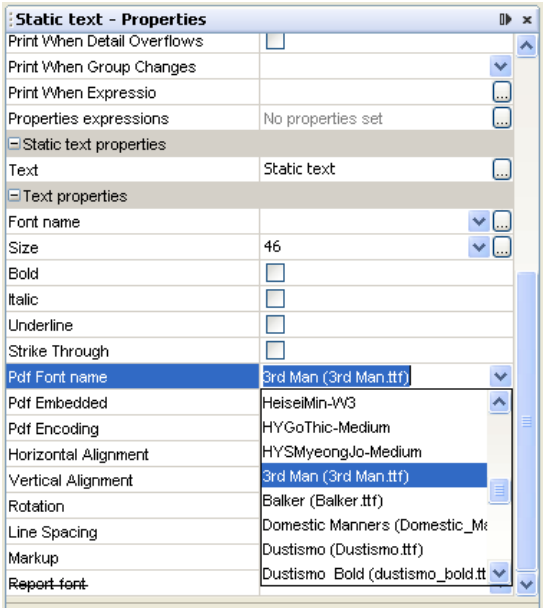



**Figure 8-2 PDF Font Names**

Anyway, the Font name property can be freely edited. If the specified font name is not found, JasperReports will use the default font instead (Sans Serif).

The list of available PDF Font names is compiled from the set of built-in PDF fonts and the list of the TrueType fonts found in the font paths (each item is presented with the font name and the name of the TrueType font to which it refers to).

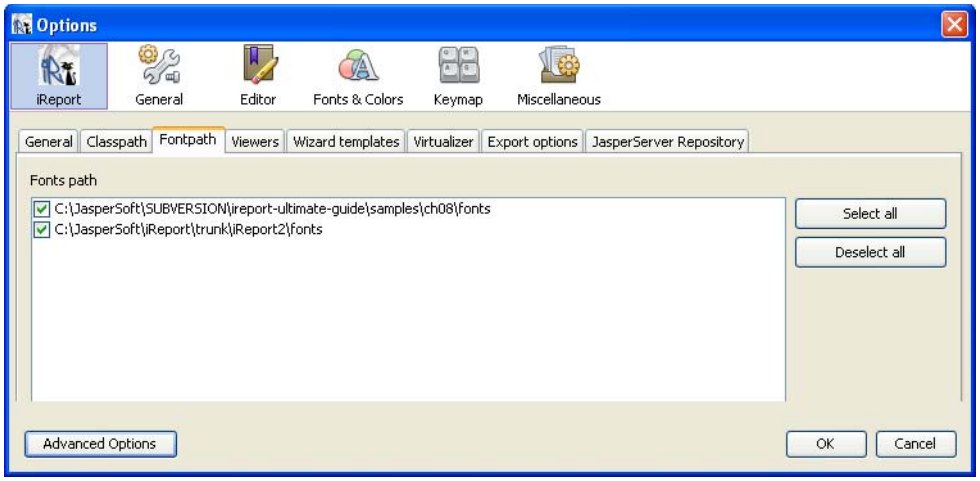



**Figure 8-3 Font Paths**

You can set the font paths in the options dialog. In general, JasperReports (the report engine) looks for fonts in the classpath. Since scanning the entire classpath for all the available fonts can significantly delay report generation, iReport uses a subset of the classpath paths when looking for fonts. To set the fonts paths, add that paths to the classpath first (see the classpath tab in the options dialog), and then check them in the **Fontpath** tab.

Avoid adding hundreds of TrueType fonts to the fontpath because this slows down the start of iReport. For Windows in particular, avoid adding the `%WINDIR%\fonts` directory to the fontpath.

If you need to use a TrueType font that is not available when you are designing your report, edit the TTF file name directly in the combo box. Please note that if the file is not found when the report is run, this will result in an error when you export it as a PDF).

If the selected font is an external TTF font, to ensure that the font is viewed correctly in the exported PDF document, select the **PDF Embedded** check box: this will force the report engine to include the required fonts as metadata in the PDF file, but will increase the document size.

## 8.3 Character Encoding

Correct character encoding is crucial in JasperReports, particularly when you have to print in PDF format. Therefore, it is very important to choose the right PDF encoding for characters.

The encoding specifies how characters are to be interpreted. In Italian, for example, to print correctly accented characters (such as è, ò, a, and ù), you must use CP1252 encoding (Western European ANSI, also known as WinAnsi).iReport provides an extensive set of predefined encoding types in the **PDF Encoding** combo box in the **Font** tab of the element properties window.

If you have problems with reports containing non-standard characters in PDF format, make sure that all the fields have the same encoding type and check the charset used by the database from which the report data is read.

## 8.4 Use of Unicode Characters

You can use Unicode syntax to write non-Latin-based characters (such as Greek, Cyrillic, and Asian characters). For these characters, specify the Unicode code in the expression that identifies the field text. For example, to print the Euro symbol, use the Unicode `\u20ac` character escape.

The expression `\u20ac` is not simple text; it is a Java expression that identifies a string containing the € character. If you write this text into a static text element, "\u20ac" will appear; the value of a static field is not interpreted as a Java (or other language) expression (this only happens with the textfields where the context is provided using an expression).

## 8.5 Working with Styles

iReport displays the available styles in the outline view, in the node labeled **Styles**. To create a new style, right-click the **Styles** node and select **Add style** from the contextual menu (see Figure 8-4).

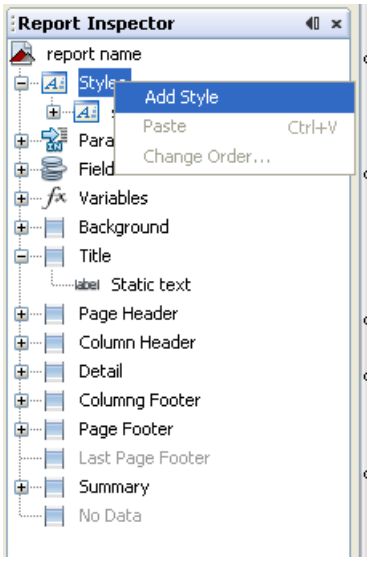



**Figure 8-4 Adding a new style to the document**

You can define many properties for a style, which are then shown in the property sheet (Figure 8-5). Leave the default value when you don't want set a specific value for a property. To reset a value, right-click the property name and select **Reset to**

**default value**. This works with all the element properties that support a default value. For instance, those are all properties that can be overridden using a style when the style is applied to the element.

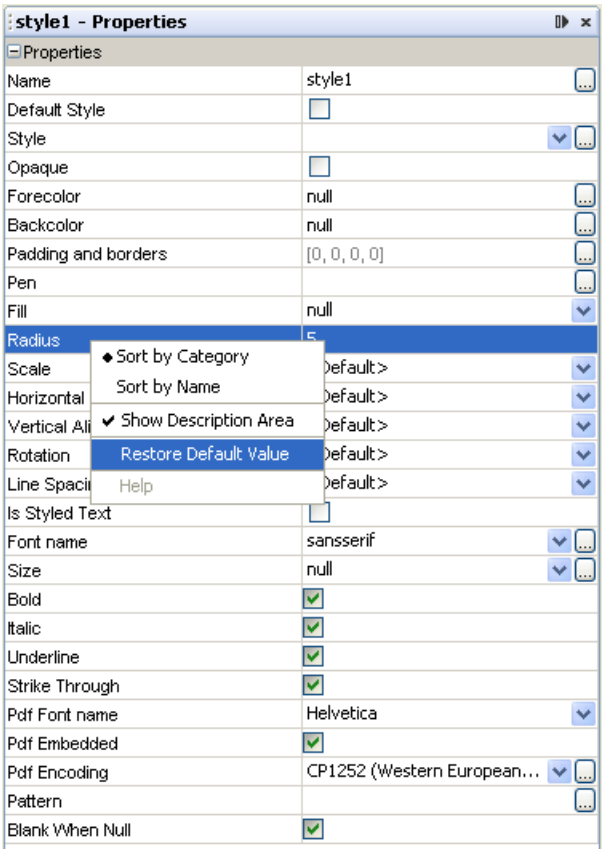



**Figure 8-5 All the properties of a style**

The only mandatory property of a Style is the Style name. All other fields are optional.

You can choose a specific style to be the default style for your report. If set a default style, all the element properties having an unspecified value will implicitly inherit a value from the default style.

The Parent style property defines the style from which the current one inherits default properties.

The other properties fall into the following four categories: common properties, graphics properties, border and padding properties, and text properties. For details about each of these properties, refer to **Chapter 5**.

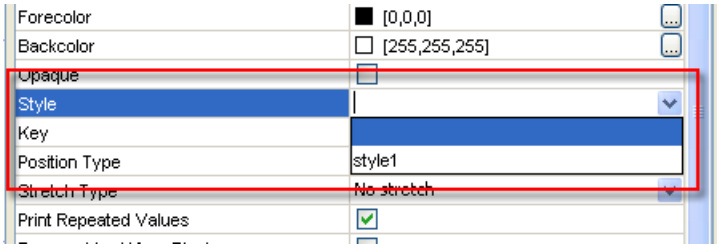



**Figure 8-6 Style property of an element**

To apply a style to an element, select the element and set the desired style in the property sheet (**Figure 8-6**).

## 8.6 Creating Style Conditions

You can design your report so that a style will change dynamically. For example, you can set the foreground color of a text field to black if a particular value is positive and red when it is negative. iReport creates conditional styles as deriving from an existing style, for which we set the condition and change some properties.

To apply a condition to a style, right-click on the style node and select **Add Conditional Style** (see Figure 8-7).

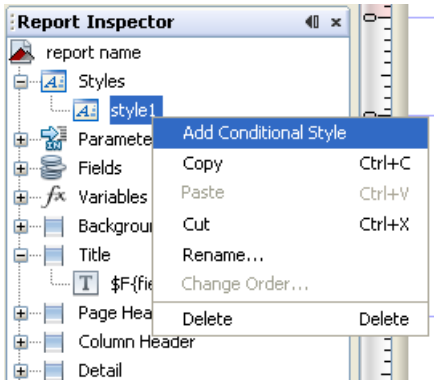



**Figure 8-7 New conditional style**

You can reconfigure all the values of the parent style. these new values will be used instead the ones defined in the parent when the condition is true. The new conditional style will appear in the outline view. You need to set the condition (actually an expression that returns a Boolean value) that will be evaluated during the rendering of elements that use that style.

In the condition expression you can use all properties of the report object. Please note that for their nature, these conditions can not be much generic, for instance you can not set a condition like “if the number is positive” or “if the string is null”. You must be very specific, saying in example what particular value (field, parameter, variable or any expression involving them) must be positive or null an so on.

A style can have an arbitrary number of conditional styles. A good example of this would be designing your report to display a particular field (let's say the total number of orders placed in a country) as a different color, depending the value of the field. You would set the foreground color as red in the base style, then add a conditional style to be used when the variable `$V{total_orders}` is less that 5 for which the color would be red, another conditional style with the foreground color set to

yellow when the same value is between 6 and 10, and green for another conditioned style for a number of order greater than 20.

| | | | | |
|---|---|---|---|---|
| 10248 | Vins et alcools | 7/16/96 12:00 AM | 59 rue de l'Abbaye | Reims |
| 10249 | Toms Spezialitäten | 7/10/96 12:00 AM | Luisenstr. 48 | Münster |
| 10250 | Hanari Carnes | 7/12/96 12:00 AM | Rua do Paço, 67 | Rio de Janeiro |
| 10251 | Victuailles en stock | 7/15/96 12:00 AM | 2, rue du Commerce | Lyon |
| 10252 | Suprêmes délices | 7/11/96 12:00 AM | Boulevard Tirou, 255 | Charleroi |
| 10253 | Hanari Carnes | 7/16/96 12:00 AM | Rua do Paço, 67 | Rio de Janeiro |
| 10254 | Chop-suey Chinese | 7/23/96 12:00 AM | Hauptstr. 31 | Bern |
| 10255 | Richter Supermarkt | 7/15/96 12:00 AM | Starenweg 5 | Genève |
| 10256 | Wellington | 7/17/96 12:00 AM | Rua do Mercado, 12 | Resende |
| 10257 | HILARION-Abastos | 7/22/96 12:00 AM | Carrera 22 con Ave. | San Cristóbal |
| 10258 | Ernst Handel | 7/23/96 12:00 AM | Kirchgasse 6 | Graz |
| 10259 | Centro comercial | 7/25/96 12:00 AM | Sierras de Granada | México D.F. |
| 10260 | Ottilies Käseladen | 7/29/96 12:00 AM | Mehrheimerstr. 369 | Köln |
| 10261 | Que Delícia | 7/30/96 12:00 AM | Rua da Panificadora, | Rio de Janeiro |
| 10262 | Rattlesnake Canyon | 7/25/96 12:00 AM | 2817 Milton Dr. | Albuquerque |
| 10263 | Ernst Handel | 7/31/96 12:00 AM | Kirchgasse 6 | Graz |
| 10264 | Folk och fä HB | 8/23/96 12:00 AM | Åkergatan 24 | Bräcke |
| 10265 | Blondel père et fils | 8/12/96 12:00 AM | 24, place Kléber | Strasbourg |
| 10266 | Wartian Herkku | 7/31/96 12:00 AM | Torikatu 38 | Oulu |
| 10267 | Frankenversand | 8/6/96 12:00 AM | Berliner Platz 43 | München |
| 10268 | GROSELLA- | 8/2/96 12:00 AM | 5ª Ave. Los Palos | Caracas |
| 10269 | White Clover Markets | 8/9/96 12:00 AM | 1029 - 12th Ave. S. | Seattle |
| 10270 | Wartian Herkku | 8/2/96 12:00 AM | Torikatu 38 | Oulu |
| 10271 | Split Rail Beer & Ale | 8/30/96 12:00 AM | P.O. Box 555 | Lander |
| 10272 | Rattlesnake Canyon | 8/6/96 12:00 AM | 2817 Milton Dr. | Albuquerque |
| 10273 | QUICK-Stop | 8/12/96 12:00 AM | Taucherstraße 10 | Cunewalde |

**Figure 8-8 Alternated color for each row**

Let's see an example of using a conditional style to achieve the effect of having an alternating background for each row. The effect is shown in **Figure 8-8**.

The trick is pretty simple. The first step is to add a frame element in the detail band. The frame will contain all the elements of the band, so we are using the frame like it would be the background for the band itself, in facts the frame should take all the space available in the band. Then all the textfields will be placed inside the frame (see **Figure 8-9** and **Figure 8-10**).

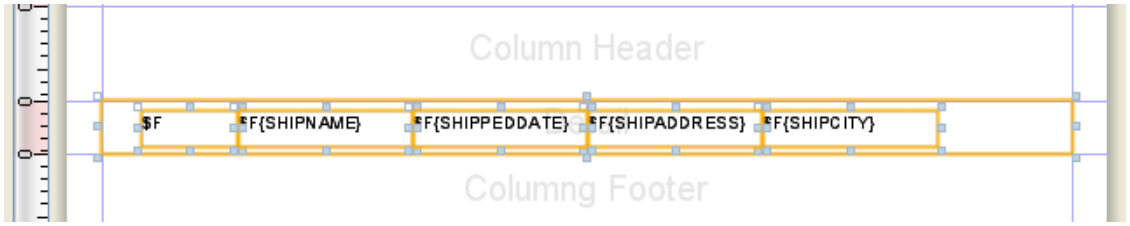



**Figure 8-9 All the elements are places inside the frame**

**Figure 8-9** does not reflects the real design, I just tried to project the idea that the textfields showing the real data are inside a the frame that covers the entire surface of the band.

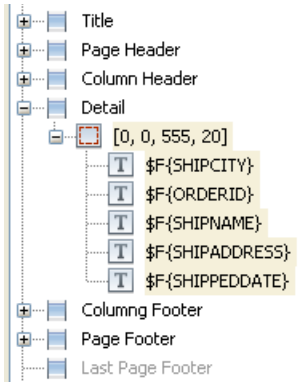



**Figure 8-10 The outline view shows the elements hierarchy**

This should be particularly clear looking at the outline view (**Figure 8-10**).

Now it's time to define a new style (let's call it style1). We will keep all the default values since we are not interested in changing them when the row number is odd (like 1, 3, 5, etc...). Now add a conditional style, and set as the condition the expression:

```
($V{REPORT_COUNT} % 2) == 0
```

Remember, we have been using Groovy or JavaScript as the report language. In Java the expression would be a bit more complicated, for example:

```
($V{REPORT_COUNT}.intValue() % 2) == 0
```

What the expression does is calculate the rest of the division by 2 (the operator % has this function), and we check if the remainder is 0. If it is, the current record (hold by the `REPORT_COUNT` built-in variable) is even.

For this conditional style, we set the background property to a light gray (or any other color of your choice) and the opaque property to true (otherwise the previous property will not take effect).Finally, we will apply the style to the frame element: select it and set the style property to *style1*, that style should appear in the list of possible choices.

Run the report, and what you get should be exactly what has been presented in **Figure 8-8**.

# 9 SUBREPORTS

Subreports represent one of the most advanced feature sets of JasperReports, and they enable the design of very complex reports. The aim is to be able to insert a report into another report created with modalities similar to the original one.

You have seen that to generate a report you need three things: a Jasper file, a parameters map (it can be empty) to set a value for the report parameters, and a data source (or a JDBC connection) that can be empty. In this chapter, I will explain how to pass these three objects to the subreport through the parent report and by creating dynamic connections that are able to filter the records of the subreport based on the parent's data. Then I will explain how to return information regarding the subreport creation to the parent report.

## 9.1 Creating a Subreport

As noted previously, a subreport is simply a report composed of its own JRXML source and compiled in a Jasper file. To create a subreport means to create a normal report. You have to pay attention only to the print margins, which are usually set to zero for subreports, just because a subreport is meant to be a portion of a page, not a document per se. The horizontal dimension of the report should be as large as the element into which it will be placed in the parent report. A subreport element is what we add to a report in order to insert a subreport. It is not necessary that the Subreport element be exactly as large as the report we will use as subreport; however, in order to avoid unexpected results, it is always better to be precise.

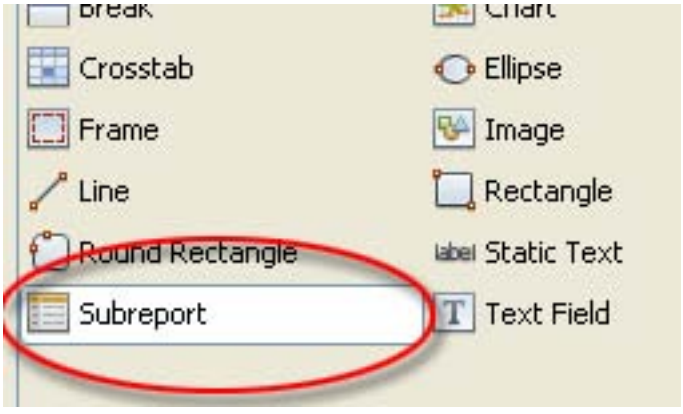



**Figure 9-1 The Subreport Element**

In the parent report it is possible to insert a subreport adding the Subreport element (circled in Figure 9-1); at design time the element will be rendered like a rectangle.

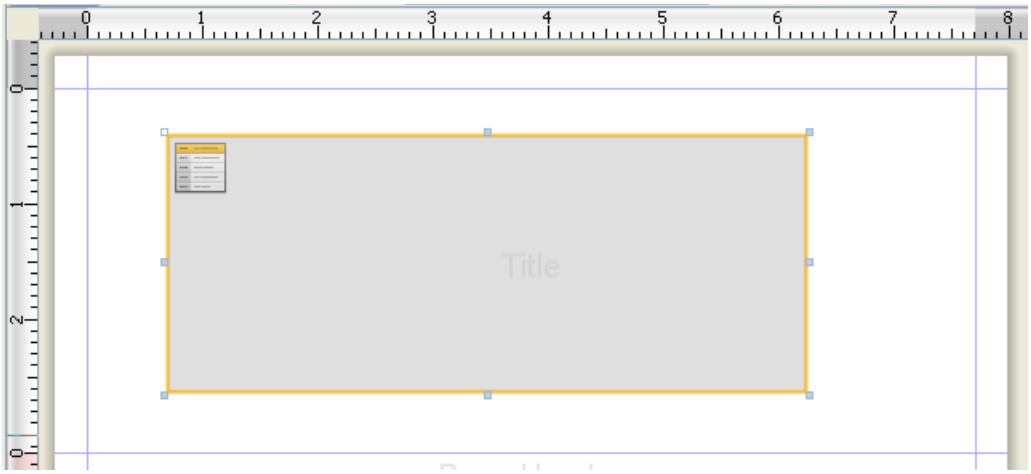



**Figure 9-2 A subreport element placed in the title band**

The Subreport element dimensions are not really meaningful because the subreport will occupy all the space it needs without being cut or cropped. You can think of a Subreport element as a place holder defining the position of the top-left corner to which the subreport will be aligned.

## 9.2 Linking a Subreport to the Parent Report

To link the subreport to the parent report means to define three things:

- How to recover the Jasper object that implements the subreport
- How to feed it with data
- How to set the value for the subreport parameters.

All this information is defined through the Subreport element property sheet (see **Figure 9-3**).

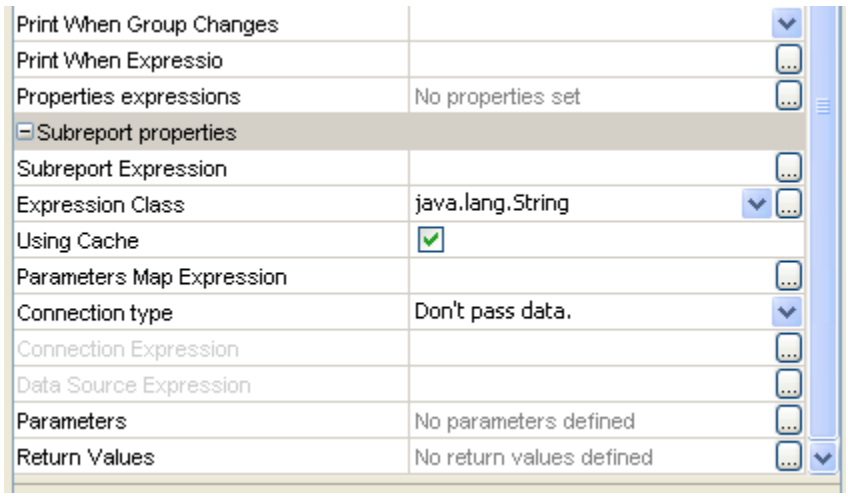



**Figure 9-3 Subreport Element Properties**

First, let's take a look at how subreport parameters are set.

## 9.3 Specifying the Subreport

When we add a subreport to a report, we assume we are able to somehow locate the Jasper file to use to generate the subreport. We instruct JasperReports how to locate this file or object setting the subreport expression. As in many other contexts, where expressions are involved, we need to set its type, the kind of object returned from the expression. It's easy to imagine that JasperReports will act differently accordingly to the expression type in order to load the subreport. The type is set specifying a value for the Expression Class property. It must be selected from the combo box (refer back to **Figure 9-3**). **Figure 9-4** lists the possible object types.

| Possible return types of the Subreport Expression | |
|---|---|
| *net.sf.jasperreports.engine.JasperReport* | It represents the jasper file pre-loaded in a JasperReport object |
| *java.io.InputStream* | It is an open stream of the jasper file |
| *java.net.URL* | It is an URL file that identifies the location of the jasper file |
| *java.io.File* | It is a file object that identifies the jasper file |
| *java.lang.String* | It identifies the name of the jasper file |

**Figure 9-4 Possible values for the subreport expression type**

If the expression is a string (`java.lang.String`), JasperReports will assume that the subreport must be loaded from a Jasper file and will try to locate the file in the same way other resources like images are looked for. Specifically, the string is at first interpreted as an URL. In case of failure (a `MalformedURLException` being returned), the string is interpreted as a physical path to a file; if the file does not exist, the string is interpreted as resource located in the classpath. This means that using an expression of type string, you are in some way trying to specify a file path; optionally, you can put your Jasper file in the classpath and refer it as resource (meaning your expression will be something like "`subreport.jasper`" assuming that the directory containing the file `subreport.jasper` is in the classpath).

Many people are concerned why a relative path cannot be used to locate the subreport file; in other words, why if I have a report in `c:\myreport\main_report.jasper` cannot refer to a subreport just by using an expression like: "..\\mysubreports\\mysubreport.jasper". Well, this cannot be done because JasperReports does not keep in memory the original location of the Jasper file that it's working with. This makes perfect sense considering that a Jasper object can be not necessarily being loaded from a physical file.

The first step of configuring a subreport element should now be clear: create an expression that can be used to load the Jasper object to use when filling the subreport portion of the document. From experience, it will be a Jasper file stored somewhere in the file system 99% of the time. If we are loading the subreport from the filesystem, I suggest two options to make the life of the designer and of the developer a bit easier: both are very similar to the ones explained talking about referencing images.

The first is to place the subreport file in a directory included in the classpath. This will permit you to use very simple subreport expressions like a string containing just the name of the subreport file (i.e. "`subreport.jasper`"). For instance, iReport always includes the classpath the directory in which resides the report that it is running, so when working with iReport, all the subreport jasper files located in the same directory as the parent report are found with this approach.

The second option is to parametrize the jasper file location and create on the fly the real absolute path of the file to load. This can be achieved by using a parameter containing the parent directory of the subreport (let's call it `SUBREPORT_DIRECTORY`) and create an expression like:

```
$P{SUBREPORT_DIRECTORY} + "subreport.jasper"
```

The advantages of this approach are multiple: it is possible to set a default value for the `SUBREPORT_DIRECTORY` parameter to be used at design time by specifying a local directory where we are used putting the jasper files. The developer that will integrate JasperReports in his application can set a different value for that location just by passing to the report a different customized value for the SUBREPORT_DIRECTORY parameter.

## 9.4 Specifying the Data Source

To set the subreport data source means to tell JasperReports how to retrieve data to fill the subreport. Usually we have to options: the subreport uses an SQL query and needs the same JDBC connection used by the parent (super-easy case), or we are using a different type of data source (like an XML file) and we need some data source elements to create an expression to create or just pass the requires data source instance. Finally, there is another very special case: it is when we don't pass any data to the subreport. We will deal with this last option in more depth since it can provide a way to just simulate inclusion of static portions of documents (like headers, footers, backgrounds and so on).

Using a JDBC connection makes the use of the subreport simple enough. In this case, a connection expression must identify a `java.sql.Connection` object (ready to be used, so a connection to the database is already opened). If we are not creating anything esoteric, we'll probably just use the same database connection to run the SQL query defined in the parent report, that can be referenced using the built-in parameter REPORT_CONNECTION. It must be clear that if we pass a JDBC connection to the subreport, it is because we defined an SQL query in the subreport, a query that will be used to fill it.

The use of a data source is more complex (even if sometimes necessary when a connection like JDBC is not being used), but it is extremely powerful. It requires writing a data source expression returning the `JRDataSource` instance you then use to fill the subreport. Depending by what you want to achieve, it is possible to pass the data source that will feed the subreport through a parameter or create it dynamically all the times is required. If the parent report is executed using a data source, this data source is stored in the REPORT_DATASOURCE built-in parameter; differently by the REPORT_CONNECTION parameter, the REPORT_DATASOURCE should never be used to feed a subreport: a data source is in general a consumable object that is usable for feeding a report only once, so a data source will satisfy the needs of a single report (or subreport). Therefore, the parameter technique is not suitable when every record of the master report has its own subreport (unless there is only one record in the master report). When we discuss data sources this will be much more clear and you will see how this problem is easily solved by using custom data sources, and how to create subreports using different type of connections and data sources.

## 9.5 Passing Parameters

When a report is invoked from a program (using for instance one of the `fillReport` methods), a parameters map is passed to set a value for its parameters. A similar approach is used to set a value for subreport parameters; with subreports you don't have to define a map (even if possible specifying a *Parameters Map Expression*), the report engine will take care of that for you, but you can still create a set of parameter name/object pairs that will be used to set the value of the subreport parameters.

The big advantage is that these values can be provided in the form of expressions, making it potentially dynamic. Subreport parameters are set using the *parameters* property (see **Figure 9-3**). Click the button labeled "**...**" to pop up the subreport

parameters dialog (see **Figure 9-5**). Also, in this case, the interface is quite self-explanatory: by clicking the Add button, you can bring up a dialog box in which it is possible to add a new parameter that will feed the parameters map of the subreport.

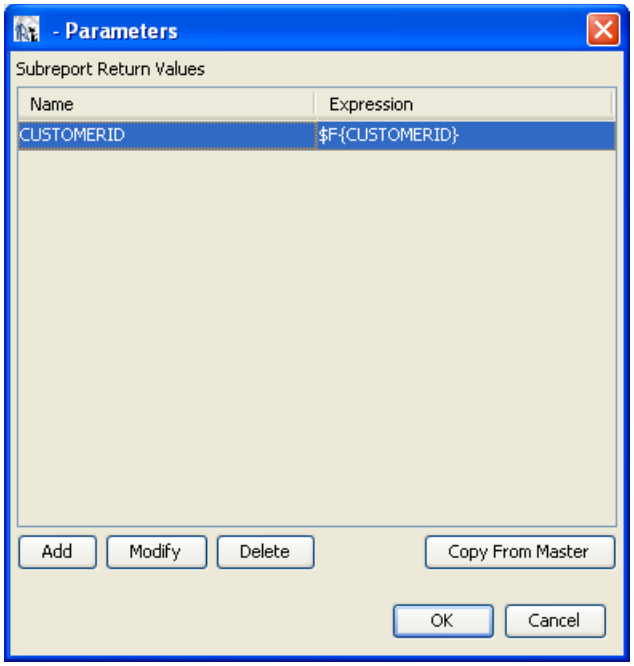



**Figure 9-5 Subreport Parameters Dialog**

The parameter name has to be the same as the one declared in the subreport. The names are case sensitive, which means that capitalization counts. If you make an error typing the name or the inserted parameter has not been defined, no error is thrown (but probably something would end up not working, and you would be left to wonder why).

In the *Value Expression* field in the Subreport parameter dialog, you supply a classic JasperReports expression in which you can use fields, parameters, and variables. The return type has to be congruous with the parameter type declared in the subreport; otherwise, an exception of `ClassCastException` will occur at runtime.

One of the most common uses of subreport parameters is to pass the key of a record printed in the parent report in order to execute a query in the subreport through which you can extract the records referred to (report headers and lines). In example you have in the master report a set of customer for which you want to show some extra information like a list of contacts. The subreports will use the customer ID to get the contacts. The customer ID should be passed to the subreport as parameter, and its value will change for each record in the master report.

As cited below, you have the option to directly provide a parameters map to be used with the subreport; the *Parameters Map Expression* allows you to define an expression, the result of which must be a java.util.Map object. It is possible, for example, to prepare a map designed for the subreport in your application, pass it to the master report using a parameter, and then use that parameter as expression (for example, $P{myMap}) to pass the map to the subreport. It is also possible to pass to the subreport the same parameters map that was provided to the parent by using the built-in parameter REPORT_PARAMETERS_MAP: in this case the right expression will looks like

```
$P{REPORT_PARAMETERS_MAP}
```

Since the subreport parameters can be used in conjunction with this map, you could even use this map to pass a set of common parameters (like for instance the username of the user that is executing the report and stuff like that).

## 9.6 A Step-by-Step Example

Let’s put into practice what you have learned in the previous sections. Say you want to print a set of countries in which orders have been placed and use a subreport to show the list of customers present in each country, using a subreport. You will use a

JDBC connection to the JasperReports sample database: the only two tables involved are orders (to extract the name of the countries) and address (to get the customer information).

Create a new empty report and call it `master.jrxml`. Click on the query dialog button in the tool bar to access open it (it's the first button representing a cylinder). We assume that the currently active connection points to JasperReports Sample database.

Set the query as follow. The query is designed to extract the distinct names of countries ordered by name:

```
select distinct shipcountry from orders order by shipcountry
```

If iReport does not provide it automatically, press read fields to get the fields from the query (actually just one: `shipcountry`, see **Figure 9-6**).

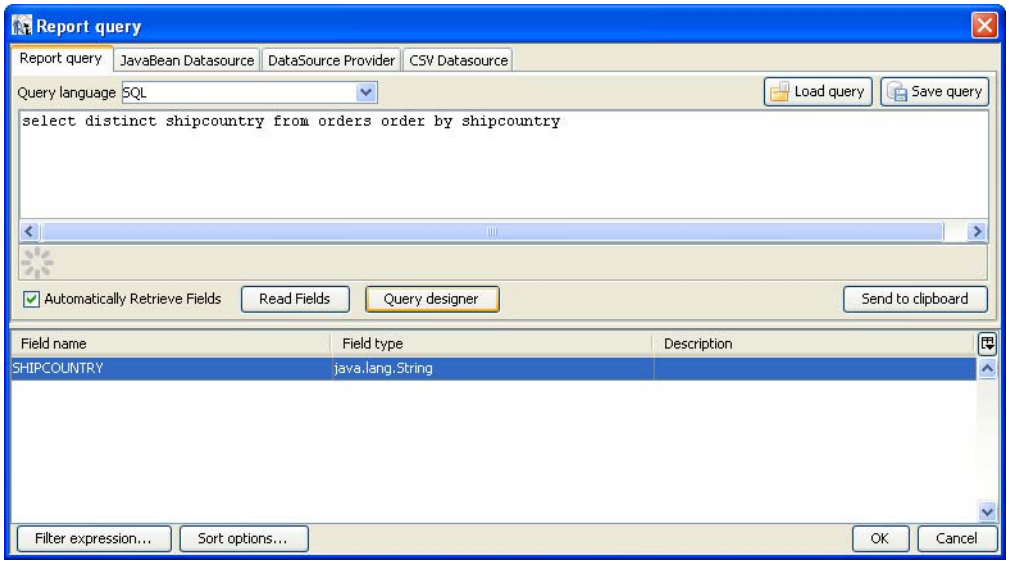



**Figure 9-6 The first step is to select the records for the master report**

Press **OK** to close the dialog, the `shipcountry` field should be appear in the field's list in the outline view.


**Figure 9-7 The `SHIPCOUNTRY` field has been placed in the detail**

Select it and drag the field in the details band, adjusting the textfield and font size (see **Figure 9-7**).


**Figure 9-8 Just a simple list of countries**

Test the report by clicking on the preview button. You should get something similar to **Figure 9-8**.

Next, you will create the first subreport to display the list of distinct customers in each country. Please note that the report we are creating could be easily realized without using a subreport, but we are just trying to keep things simple.

So let's start to create the report that will be used as subreport. It must have the following characteristics:

- No margins (this is not mandatory, but of course we don't need them);
- No parameter to host the name of the country for which we want to display a list of distinct customers
- The width must be congruent to the space we want to reserve to it in the master report, let's say the entire page width less the margins
- A set of textfields in the detail band to show first and last name of each customer

Create a new empty report called `subreport.jrxml` (if you pick a different name, keep it in mind, we will use it when connecting the master to the subreport). Remove the page margins, and adjust the page width (i.e. an A4 page has a width of 595 pixels, subtracting 20 pixel for the left margin, and 20 for the right one, the new page should be set to a width of 555 pixels and the margins to 0).

Add to this report a parameter, we'll call it: `COUNTRY`. The type must be set to `java.lang.String` and the default value to a blank string (“”) or if you prefer to a country name (like “Argentina”). This default value will be always overridden by what we will specify from the master. We are assigning it to the parameter now just because a default value in iReport (not in JasperReports) is mandatory when using the parameter inside the report query.

Open the query dialog. The query to select the customer information based on the order's country can be something like this:

```
select distinct shipname, shipcity from orders where shipcountry = $P{COUNTRY}
```

We are getting all the distinct address information of customers that place orders in a particular country represented by the `COUNTRY` parameter.

iReport should detect the following fields: `SHIPNAME` and `SHIPCITY` (**Figure 9-9**).

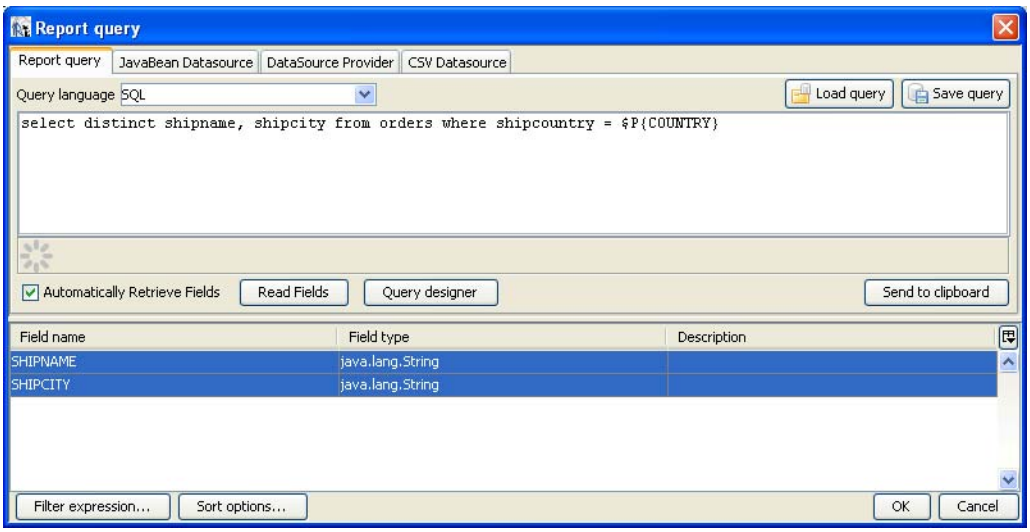



**Figure 9-9 The query of the report used as subreport**

Let's put both the fields in the detail band of the report (see **Figure 9-10**).

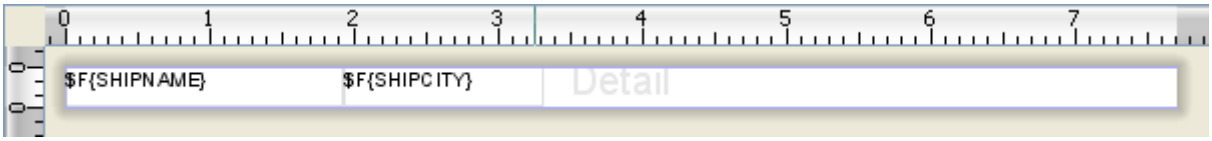



**Figure 9-10 Subreport Design**

We now have to try this report that will work as subreport. We need to test it because in this way iReport will generate the jasper file we need when using this report as subreport. When running it, pay attention to what iReport displays in the console view (see **Figure 9-9**). In particular pay attention to where the jasper file has been stored. By default, it should be the same location in which you saved the jrxml file. In this example I used the same directory for master and subreport templates, so it will contains both `master.jasper` and `subreport.jasper`.

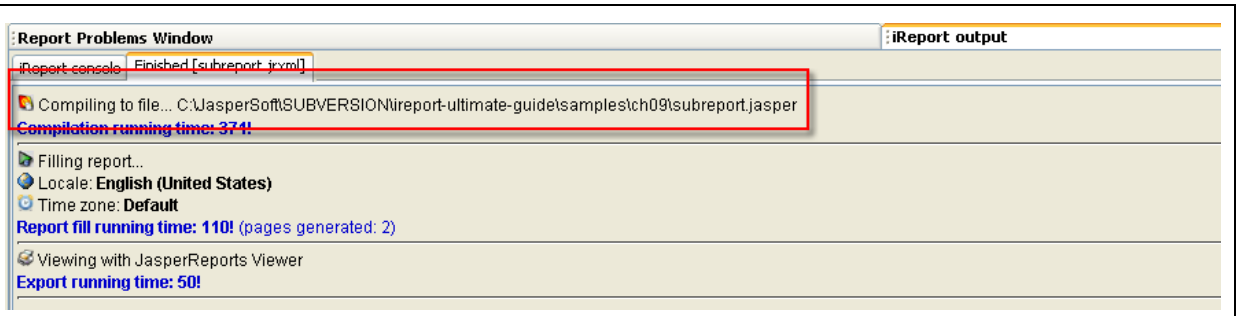



**Figure 9-11 Output of the execution of the subreport as standalone report**

Depending by the value of `COUNTRY` parameter, you can get an empty report or a report showing some items. It does not matter very much, just check that you generated your jasper file.

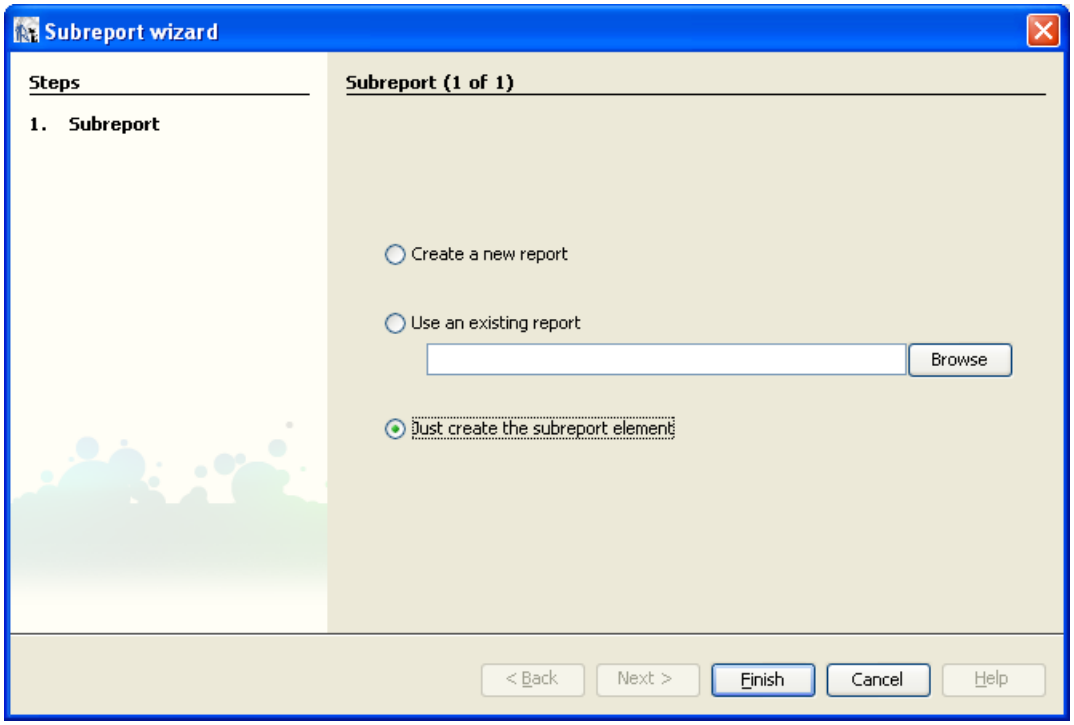



**Figure 9-12 Subreport Wizard**

What we have now is two reports: the master showing the country names, and the second one, showing a the customers for a particular country. Let's put them together: insert a Subreport element into the master report, in the detail band. When adding a subreport element, the subreport element wizard pops up (**Figure 9-12**). For now we will skip the wizard. When all the concepts about using subreports are clear, the wizard will provide a way to save some time. Check the option *Just create the subreport elements* and press *Finish*.


**Figure 9-13 The subreport element placed in the detail band**

Adjust the subreport element to use the whole band horizontal size. The vertical dimension of this element is not important because when you print the report, JasperReports will use all the vertical space necessary in spite of element size (see **Figure 9-13**).

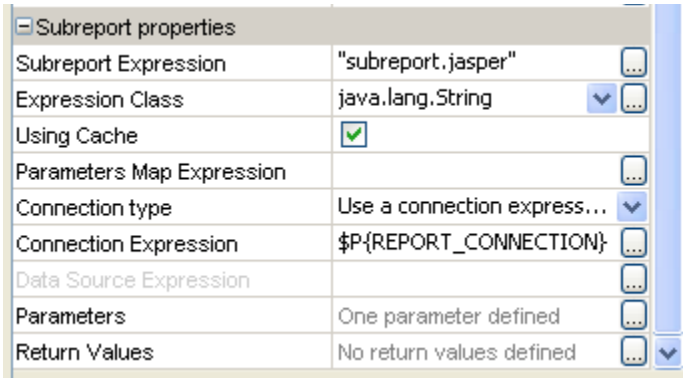



**Figure 9-14 Subreport Properties**

Now select the subreport element. The property sheet should now show its properties (see **Figure 9-14**). We want to use the same database connection we used with the master report to populate the subreport, so set the *connection type* to **Use a connection expression**. The expression will be just `$P{REPORT_CONNECTION}`.

As explained `REPORT_CONNECTION` is a built-in parameter holding a reference to the connection used in the report.

The second step is to define the subreport expression, used by JasperReports to locate the report that must be used as subreport. Assuming the subreport Jasper file is in the classpath (from an iReport prospective it's enough it is in the same directory as the master report), the expression we'll use is just:

```
"subreport.jasper"
```

Finally, since the subreport we are using requires the `COUNTRY` parameter, add a subreport parameter (by clicking on the "..." button of the *Parameters* property). Click *Add* and set as subreport parameter name `COUNTRY` (the name must match the

parameter name we defined creating the report we are referencing as subreport), and as expression for the value we just choose the field containing the country name, that's `$F{SHIPCOUNTRY}`.

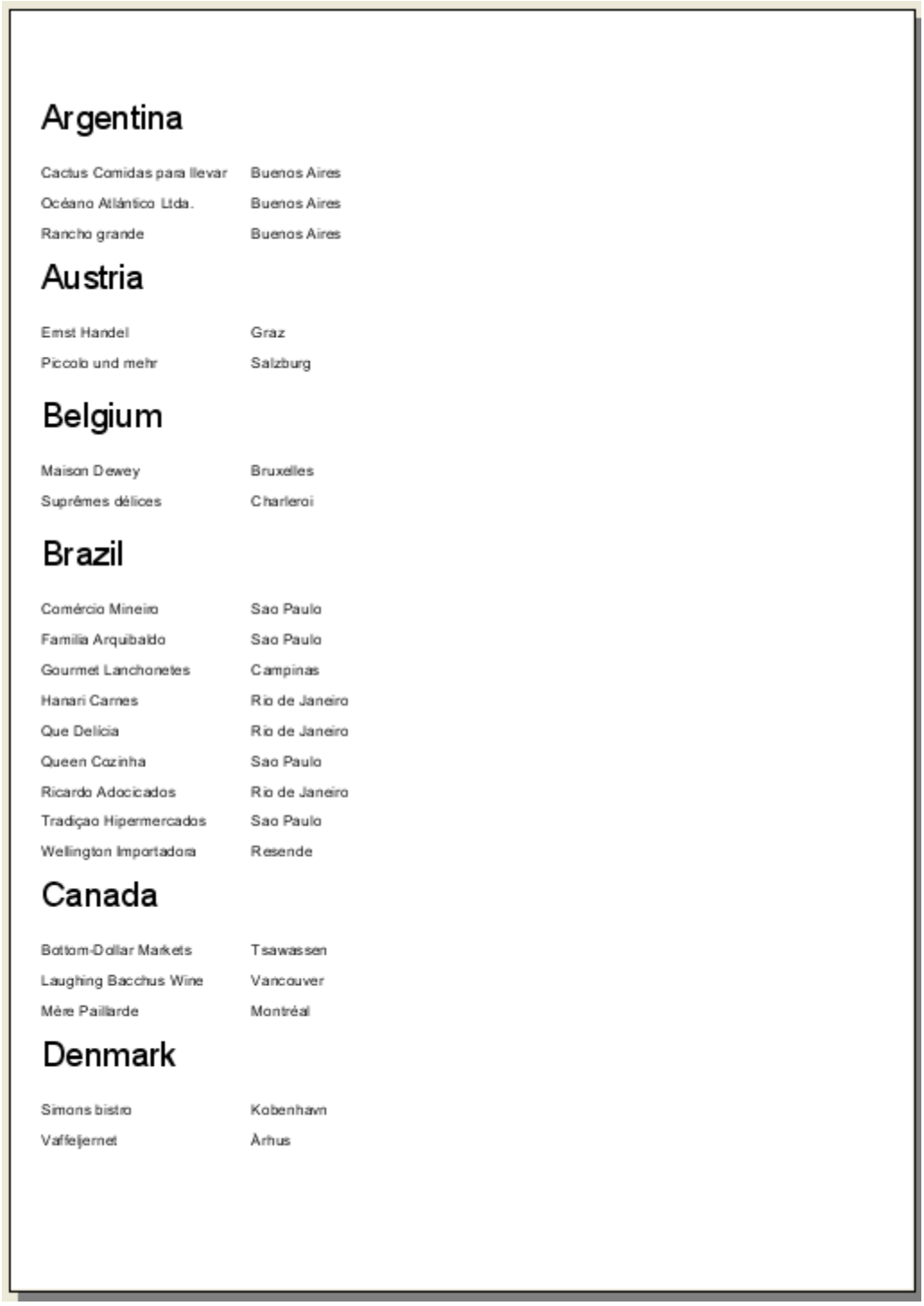



**Figure 9-15 The final result**

Preview the report. If everything has been done correctly, you should get a result like the one shown in **Figure 9-15**.

In this sample we just created the a basic subreport. The number of subreports that can be placed in a report is unlimited, and they can be used recursively, meaning that a subreport can contain other subreports. You can create very little subreports (of

the size of a textfield) and use them to lookup a value, again you can have a page layout where you place two subreport side by side showing two different lists of values, and so on.

A last note, when you have several subreports one after the other, be sure you set the position type of the report element to `Float`. In this way, you avoid the risk of overlapping the subreport when the space they require grown. Another suggestion, when using multiple subreports one after the other, is to use place each subreport in a different band splitting the detail band using what it's called a fake group, that's a group having as expression for instance the `REPORT_COUNT` parameter that ensures that for each detail you get for free the fake group header and footer bands. In this way you will optimize the report generation.

## 9.7 Returning Values from a Subreport

It is often useful to get in a report the result of some kind of calculation that has been performed in a subreport (for instance the number of records or some more complicated data).

JasperReports provides a feature that allows users to retrieve values from within a subreport. This mechanism works much the same as passing input parameters to subreports. The idea is to save values calculated during the filling of the subreport into variables in the master report.

Bindings between calculated values and local variables can be set in the Subreport return values property in the property sheet.

Let's see how to use it reusing the sample we created in the previous paragraph. Suppose we want to print in the master report the number of cities we have found for each country. By the subreport prospective, this value is represented by the `REPORT_COUNT` variable, that is not accessible directly from the master report. The first step it to create a variable to host this value at the end of the subreport elaboration. The variable must be consistent with the value it has to host, in that case it is an integer, so we have to create a new variable (let's call it for instance `SUBREPORT_COUNT`), the type must be `java.lang.Integer` and the calculation type must be set to `System`.

Now select the subreport element and change the *Return Values* property by clicking on the relative "..." button. The subreport return values dialog pops up (**Figure 9-16**).

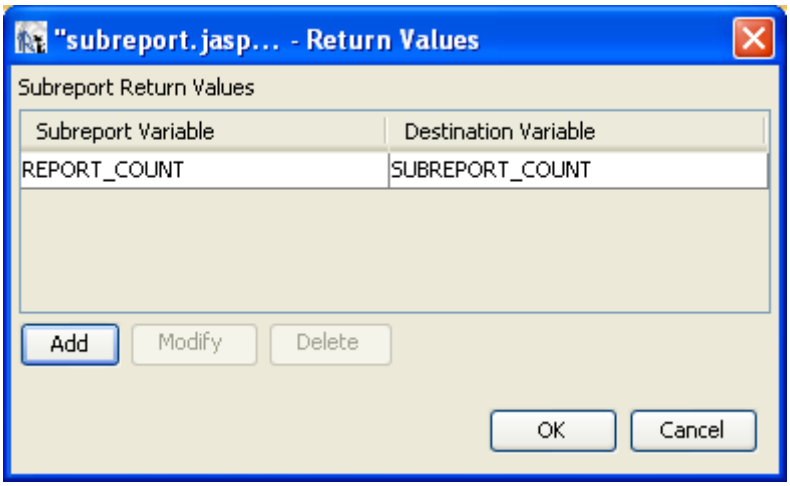



**Figure 9-16 Subreport return values**

Click the *Add* button to create the new return value; the return value dialog will appear (**Figure 9-17**):

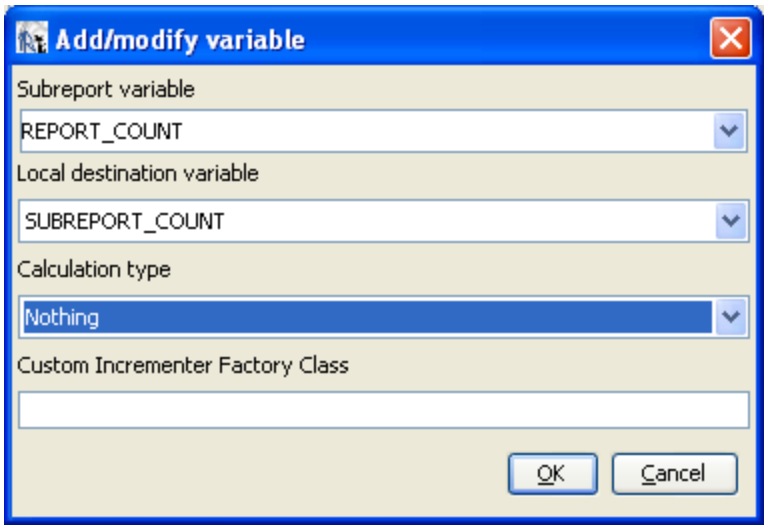



**Figure 9-17 The subreport return value definition dialog**

At this point, select a calculated value from the subreport (it must be a variable of the report used as subreport, the drop down list shows only the built-in variables), as well as the local variable that will contain the returned value (in our sample `SUBREPORT_COUNT`).

Next, select a calculation type. If you want a subreport value to be returned as is, select the value *Nothing*. Otherwise, several calculations can be applied. For example, if the desired value is the average of the number of records within a subreport that is invoked repeatedly, set the calculation type to Average.

When you create a new variable in your master report to be used like container for a returned value, set the variable calculation type to System. The effective calculation type performed on the variable values is the one defined in the dialog box shown in **Figure 9-17**.

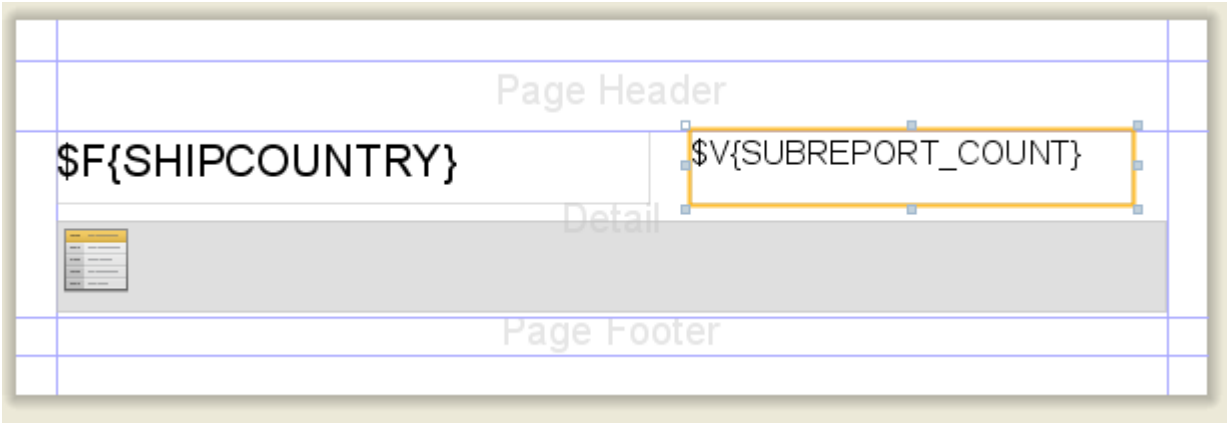



**Figure 9-18 A textfield showing the return value (with evaluation time set to Band)**

The value coming from the subreport is available only when the whole band containing the subreport is printed. If you need to print this value using a text field placed in the same band as your subreport, set the evaluation time of the text field to Band. This is what is presented in **Figure 9-18**.

The report preview should look like **Figure 9-19**.

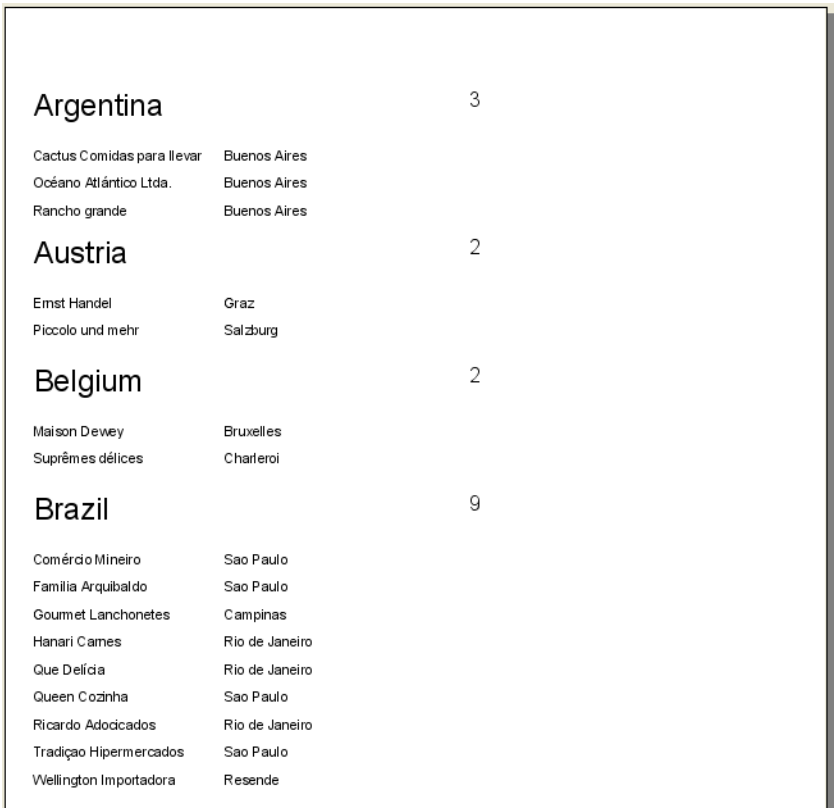



**Figure 9-19 The number of records come from subreports**

## 9.8 Using the Subreport Wizard

To simplify inserting a subreport, a wizard for creating subreports automatically starts when a Subreport element is added to a report.

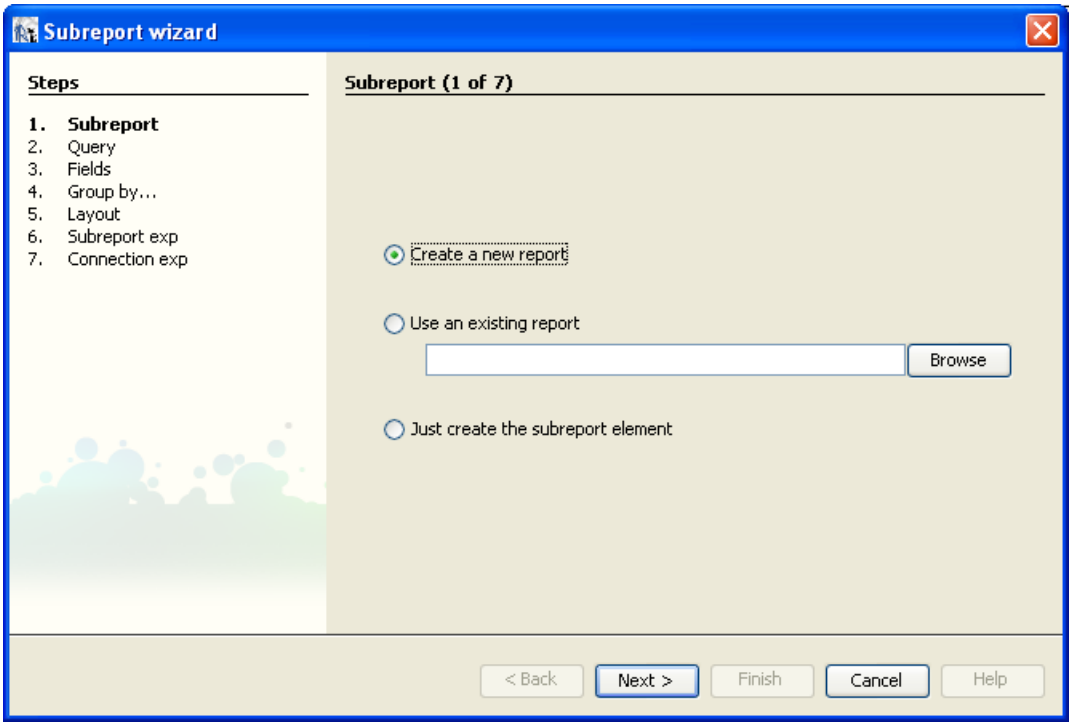



**Figure 9-20 The Subreport Wizard**

You can use the Subreport Wizard, shown in **Figure 9-20**, to create a brand new report that will be referenced as a subreport or to refer to an existing report. In the latter case, if the report you choose contains one or more parameters, the wizard provides an easy way to define a value for each of them.

### 9.8.1 Create a New Report via the Subreport Wizard

If you are adding a subreport to the current report the subreport wizard can create for you a new report that will be used as subreport.

The steps to create the new report are very similar to those you follow in the Report Wizard:

1. Select a connection or data source; if the data source requires a query (like a JDBC or Hibernate connection), you can write it in the text area or load it from a file.
2. Select fields.
3. Define grouping.
4. Select the layout.
5. Define the subreport expression.
6. Define the connection expression.

The subreport expression is used to refer to the subreport Jasper file. The wizard lets you do this in one of two ways:

- Store the path part of the subreport URL in a parameter, as shown in **Figure 9-21**, to make it modifiable at runtime by setting a different value for the parameter (the subreport path is the default value).

- Save the complete path in the expression.

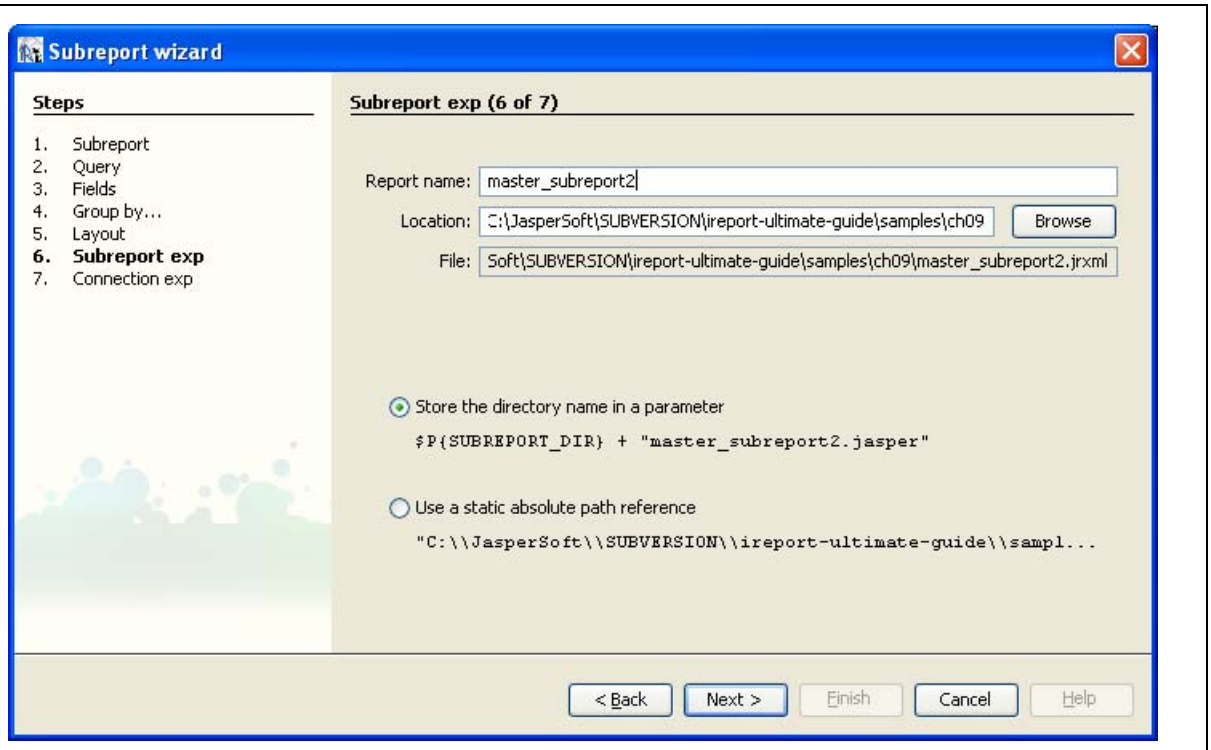



**Figure 9-21 The Subreport Expression**

If you choose the second option, the expression will store the whole absolute path to the subreport JASPER file. This assures that everything will work in iReport, but it is not very convenient in a different environment. For this reason, I suggest that you modify the expression using the method I described in **Specifying the Subreport** on **page 133**.

The subreport is not compiled when you create it. To test your report, you must first preview it: this will force it to be compiled.

When you create a new subreport, you can't specify parameters using the wizard. You will do able to add parameters and use them in the subreport query after the report is created. At this point, to filter your subreport query, follow these steps:

1. Add a parameter to the report implementing your subreport.
2. Use that parameter in your query with the typical syntax `$P{MyParam}` (or `$!P{MyParam}` if the parameter must be concatenated with the query as is).
3. In the master report, select the subreport element and add an entry in the Subreport parameters list. The new subreport parameter must be called like the counterpart in the subreport, and its value must be defined using an expression (see the section "Passing Parameters" earlier in this chapter for more details).
4. If your subreport is not based on a SQL or HQL query, you must still set the subreport data source expression to successfully run your report.
5. As mentioned earlier, you can use the Subreport Wizard to create a brand new report that will be referenced as subreport or to refer to an existing report. In the latter case, if the chosen report contains one or more parameters, the wizard provides an easy way to define a value for each of them.

### 9.8.2 Specifying an Existing Report in the Subreport Wizard

You can point to an existing report as a subreport through the Subreport Wizard. To do this, the first step is to select a JRXML or a JASPER file in the first screen of the wizard.

The second step of the wizard manages expressions for the connection or the data source used to fill the subreport (see **Figure 9-22**).

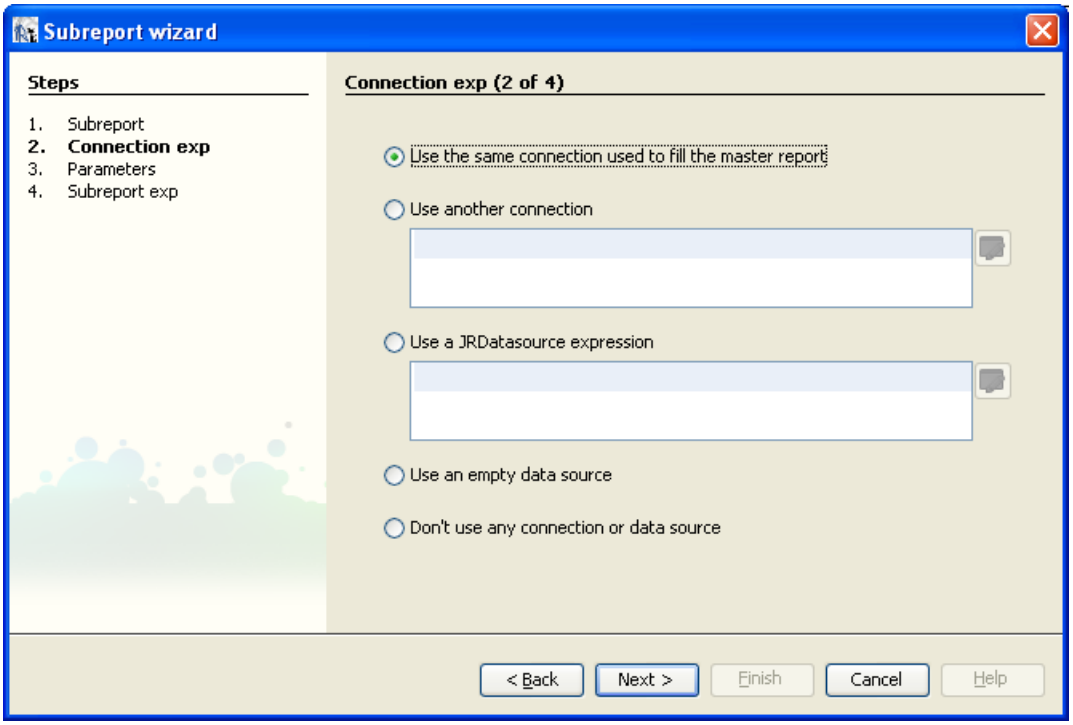



**Figure 9-22 Subreport Connection Setup**

You can select the *Use the same connection used to fill the master report* radio button option when using a JDBC-based subreport. The JDBC connection is passed to the subreport to execute it.

To specify a different JDBC connection, select the *Use another connection* radio button option.

Finally, to use a `JRDataSource` object to fill the subreport, select *Use a JRDataSource expression* and write an expression capable of returning a `JRDataSource` object.

By selecting the option *Use an empty datasource* iReport will use set the data source expression to:

```
new JREmptyDataSource()
```

that creates a special data source that provides (when declared in this way) a single record with all the field values set to null. This is very useful when the subreport is used to display static content.

In some cases you may want to avoid using a connection or a data source (last option). Again this is used when the subreport does not need any data just because you are displaying some static content. Usually a data source or a connection is always required, otherwise the report generated is blank or empty. When a subreport does not require a data source is implied that the

report property *When no data type* is set to *All data No Details* or *No Data Section* to ensure that at least a portion of the document is actually printed.

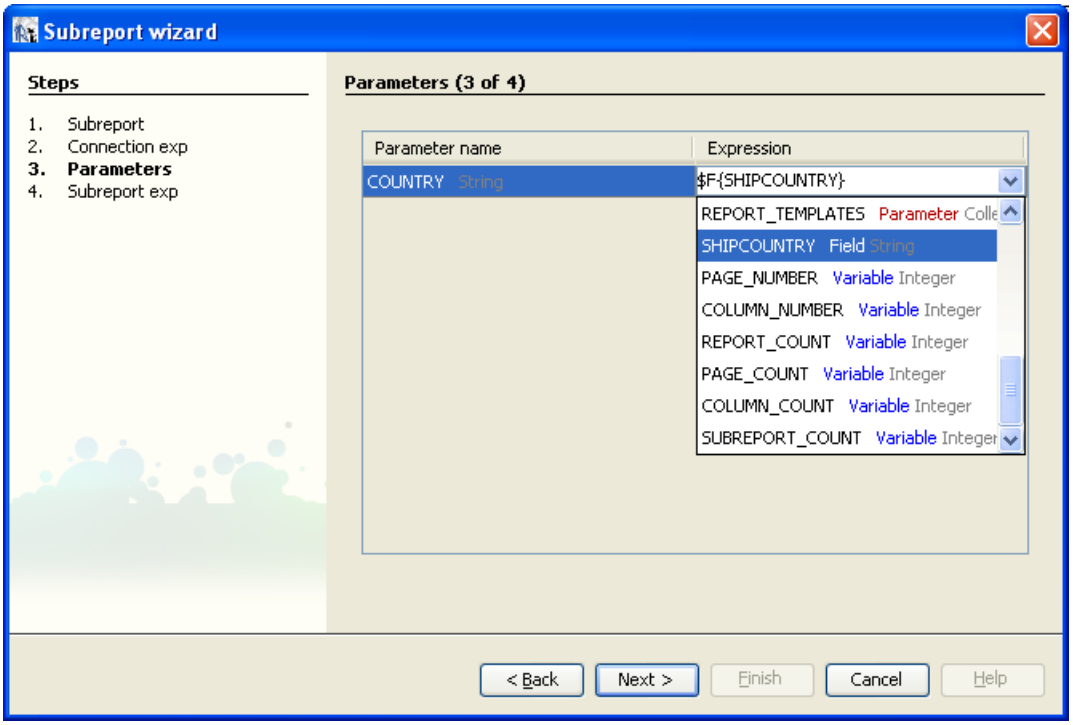



**Figure 9-23 Setting Subreport Parameters**

If the selected report contains parameters, they are listed in the next step (see **Figure 9-23**). For each parameter, you can set a value by choosing an object from the combo boxes. The combo box contains a set of suggested values. Of course, you can write your own expression, but no expression editor is provided in this context.

You can skip this step and edit the subreport parameters later using the canonical method explained previously in this chapter.

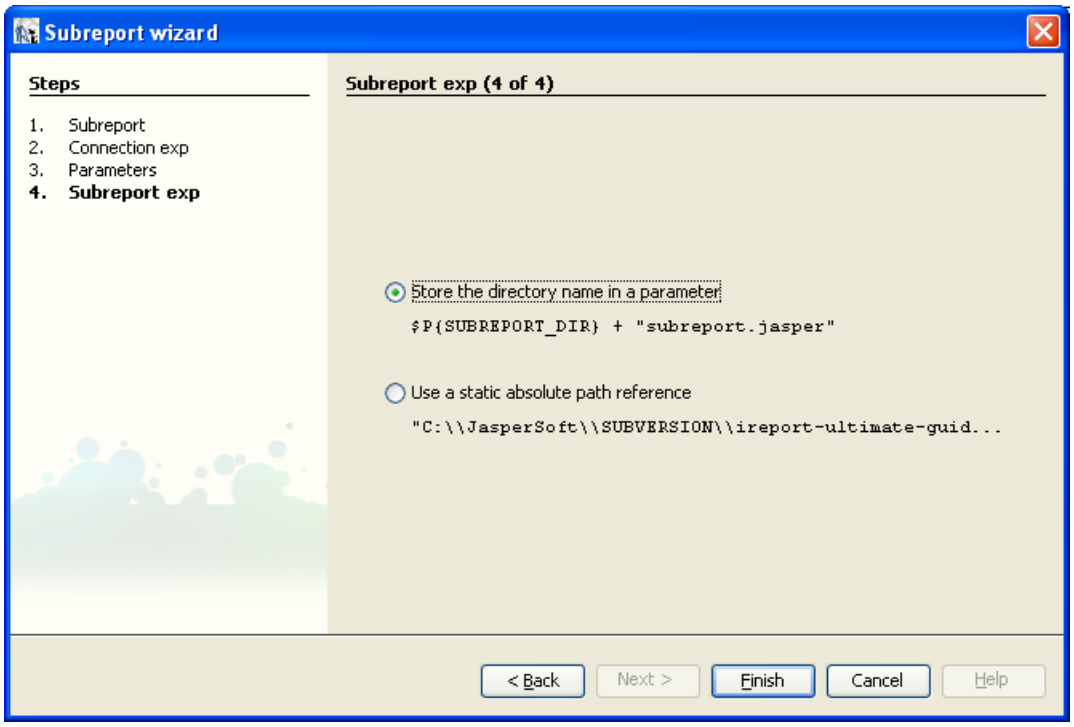



**Figure 9-24 Setting Subreport Expression**

Finally, you must designate how to generate the subreport expression. Just as for a new subreport, there are two options: store the path in a parameter to set it dynamically, or set a hard-coded path (see **Figure 9-24**).

Again, all choices can be modified after you leave the **Subreport Wizard**.

# 10 Data Sources and Query Executors

There are several ways that JasperReports can provide data to fill a report. For example, you can put an SQL query directly inside a report and provide a connection to a database against which to execute the query and read the resulting record set, or use a more sophisticated custom technology to provide a table-like set of values.

iReport provides direct support for a rich set of query languages, including SQL, HQL, EJBQL, and MDX, and supports other languages like XPath (XML Path Language).Moreover, in iReport, you can use custom languages by registering plug-in engines called query executors to interpret and execute the report query.

If you don't want to use a query language, or you simply don't want to put the query inside a report, you can use a JasperReports data source: basically, a JR data source is an object that iterates on a record set organized like a simple table. All the data sources implement the `JRDataSource` interface. JasperReports provides many ready-to-use implementations of data sources to wrap generic data structures, like arrays or collections of JavaBeans, result sets, table models, CSV and XML files, and so on. In this chapter I will present some of these data sources, and you will see how easy it is to create a custom data source to fit any possible need.Finally, you will see how to define a custom query language and a custom query executor to use it.

iReport provides support for all these things: you can define JDBC (Java Database Connectivity) connections to execute SQL queries, set up Hibernate connections using Spring, and test your own `JRDataSource` or your custom query language.

## 10.1 How a JasperReports Data Source Works

JasperReports is a records-based engine: this means that to print a report, you have to provide a set of records. When the report runs, JasperReports will iterate on this record set, creating and filling the bands according to the report definition. Bands, groups, variables—their elaboration is strictly tied to the record set used to fill the report. This is why JasperReports defines only one query per report. Multiple queries/data sources can be used when inserting subreports or defining subdatasets, each one with their own query (or data source), fields, parameters, variables, and so on. Subdatasets are only used to feed a crosstab or a chart.

Each record is a set of fields. These fields must be declared in the report in order to be used, as explained in **Chapter 6**.

But what is the difference between using a query inside a report and providing data using a `JRDataSource`? Basically, there is no difference. In fact, what happens behind the scenes when iReport uses a query instead of a `JRDataSource` is that JasperReports executes the query using a built-in or user-defined query executor that will produce a `JRDataSource`. There are circumstances where providing a JDBC connection to the engine and using a query defined at report level can simplify the use of subreports.

A `JRDataSource` is a consumable object, which means that you cannot use the same instance of `JRDataSource` to fill more than one report or subreport. A typical error is trying to use the same `JRDataSource` object (e.g., provided to the report as

parameter) to feed a subreport placed in the detail band: if the detail band is printed more than once (normally it is printed for every record present in the main data source), the subreport will be filled for each main record, and every time the subreport will iterate on the same `JRDataSource` provided. However, this will give results only the first time the data source is used.

At the end of this chapter, you will know how to avoid this kind of error, and you'll have all the tools to decide the best way to fill your report:

- Using a query in a language supported by JasperReports
- A built-in data source
- A custom data source
- A custom query language with the relative query executor

## 10.2 Understanding Data Sources and Connections in iReport

iReport allows you to manage and configure different types of data sources to fill the reports. These data sources are stored in the iReport configuration and activated when needed.

When I talk about data sources, you need to understand there is a distinction between real data sources (or objects that implement the `JRDataSource` interface) and connections, used in combination with a query defined inside the report. In addition, the term data source used in JasperReports is not the same as the concept in `javax.sql.Datasource`, which is a means of getting a physical connection to the database (usually with JNDI lookup). The data source object I refer to in the JasperReports realm already has concrete data inside itself.

Here is a list of data source and connection types provided by iReport:

- JDBC connection
- JavaBean collection data source
- XML data source
- CSV data source
- Hibernate connection
- Spring-loaded Hibernate connection
- JRDataSourceProvider
- Custom data source
- Mondrian OLAP connection
- XMLA connection
- EJBQL connection
- Empty data source

Finally, there is a special mode to execute a report, called query executor mode, that you can use to force the report creating without passing any connection to or data source for the report engine.

All the connections are "opened" and passed directly to JasperReports during report generation. For many connections, JasperReports provides one or more built-in parameters that can be used inside the report for several purposes (e.g., to fill a subreport that needs the same connection as the parent).

The XML data source allows you to take data from an XML document. A CSV (comma-separated values) data source allows you to open a CSV file for use in a report. The JavaBean set data source, custom data source, and JRDataSourceProvider allow you to print data using purposely written Java classes. The Hibernate connection provides the environment to execute HQL (Hibernate Query Language) queries (this connection can be configured using Spring too), EJBQL (Enterprise JavaBean Query Language) queries can be used with an EJBQL connection and MDX queries can be used with a native direct connection to a Mondrian server or using the standard XML/A interface to interrogate a generic OLAP database.

An empty data source is something like a generator of records having zero fields. This kind of data source is used for test purposes or to achieve very particular needs (like static content reports or subreports).

Connections and data sources are managed through the menu command *Tools→Report Datasources*, which opens the configured connections list (see Figure 10-1).

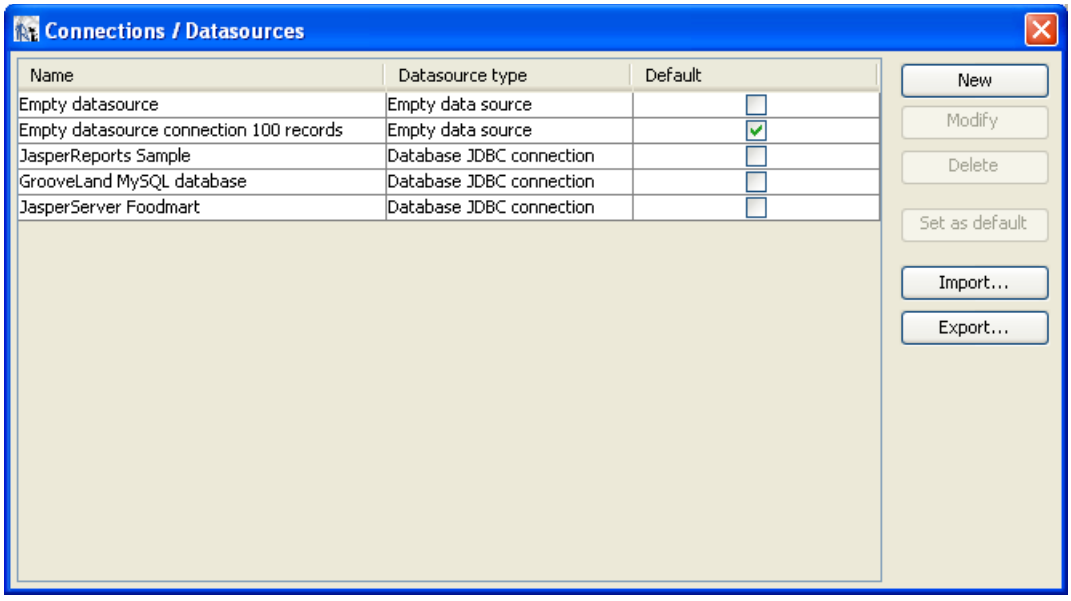



**Figure 10-1 Report Data Sources Dialog**

As mentioned previously, a connection and a data source are different objects. However, from this point on, I will use these two words interchangeably.

Even if you keep an arbitrary number of data sources ready to use, iReport works always with only one source or connection at a time. You can set the active data source in several ways. The most easy and intuitive is to select the right data source from the combo box located on the tool bar (see **Figure 10-2**).

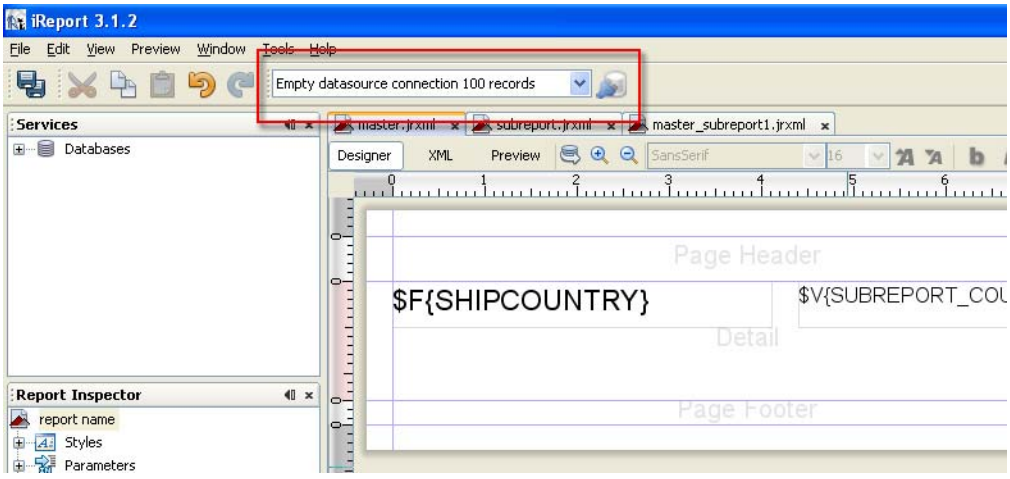



**Figure 10-2 The data sources combo box shows the active data source**

You can also set the active data source by selecting a data source in the **Connections/Datasources** dialog box and clicking the **Set as default** button (see Figure 10-1).

If no data source is selected, it is not possible to fill a report with data. When you starts iReport for the first time, a pre-configured empty data source is defined and selected by default. Datasources can be used in conjunction with the Report Wizard too, that's why configuring the connection to your data is usually the first step you do when starting working with iReport.

## 10.3 Creating and Using JDBC Connections

A JDBC connection allows you to use a relational DBMS (or, in general, whatever databasesthat are accessible through a JDBC driver) as a data source. To set a new JDBC connection, click the **New** button in the Connections/Datasources dialog box (shown earlier in **Figure 10-1, "Report Data Sources Dialog," on page 153**) to open the interface for creation of a new connection (or data source). From the list, select Database JDBC connection to bring up the window shown in **Figure 10-3**.

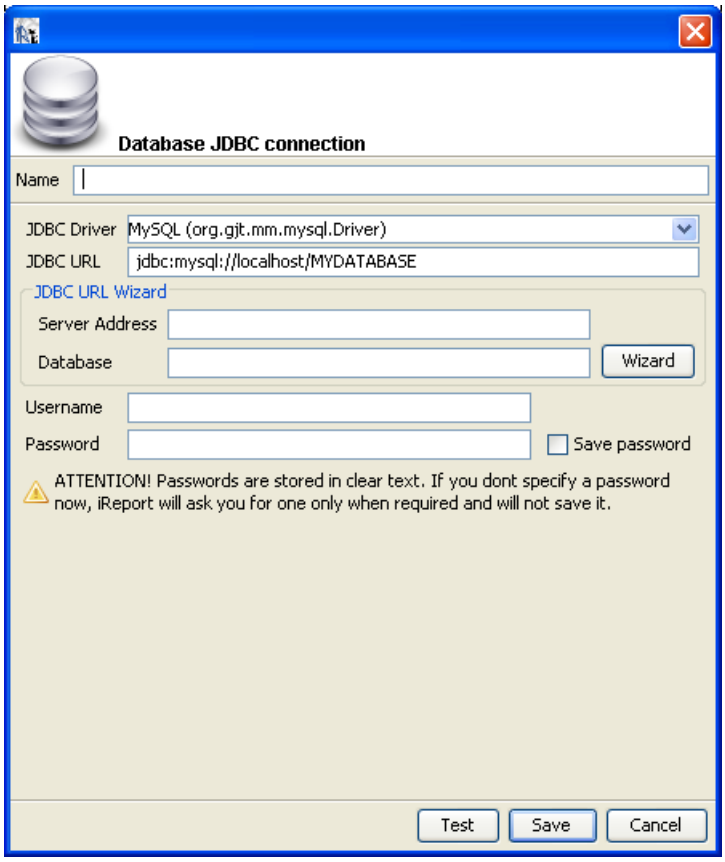



**Figure 10-3 Configuring a JDBC connection**

The first thing to do is to name the connection (possibly using a significant name, such as `Mysql – Test`). iReport will always use the specified name to refer to this connection.

In the **JDBC Driver** field, you specify the name of the JDBC driver to use for the connection to the database. The combo box proposes the names of all the most common JDBC drivers (see **Figure 10-4**).

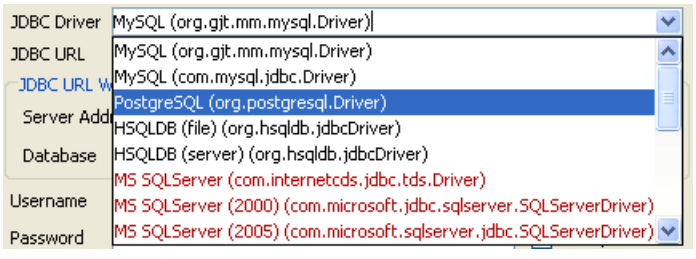



**Figure 10-4 JDBC drivers list**

If a driver is displayed in red, this means that the JDBC driver class for that driver is not present in the classpath and the it is not possible to use the driver without getting a exception. See later how to install a JDBC driver.

Thanks to the JDBC URL Wizard, it is possible to automatically construct the JDBC URL to use the connection to the database by inserting the server name and the database name in the correct text fields. Click the Wizard button to create the URL.

Enter a username and password to access the database. By means of a check box option, you can save the password for the connection.

iReport saves the password as clear text.

If the password is empty, it is better if you specify that it be saved.

After you have inserted all the data, it is possible to verify the connection by clicking the **Test** button. If everything is OK, the dialog box shown in **Figure 10-5** will appear.

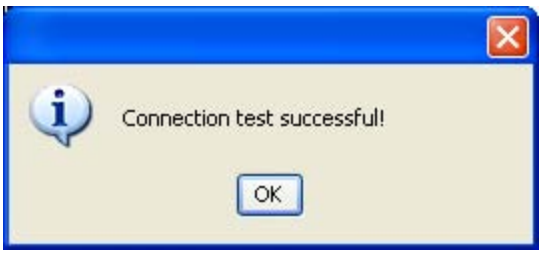



**Figure 10-5 Test Confirmation Dialog**

When you create a new data source, iReport sets it as the active one automatically for your convenience.

In general, the test can fail for a lot of reasons, the most frequent of which are the following:

- A `ClassNotFoundError` was thrown
- The URL is not correct
- Parameters are not correct for the connection (database is not found, the username or password is wrong, etc.)

Let's take a closer look at these issues.

### 10.3.1 ClassNotFoundError

The `ClassNotFoundError` exception occurs when the required JDBC driver is not present in the classpath. For example, suppose you wish to create a connection to an Oracle database. iReport has no driver for this database by default, but you could be deceived by the presence of the oracle.jdbc.driver.OracleDriver driver in the JDBC drivers list shown in the window

for creating new connections. If you were to select this driver, when you test the connection, the program will throw the `ClassNotFoundException`, as shown in **Figure 10-6**.

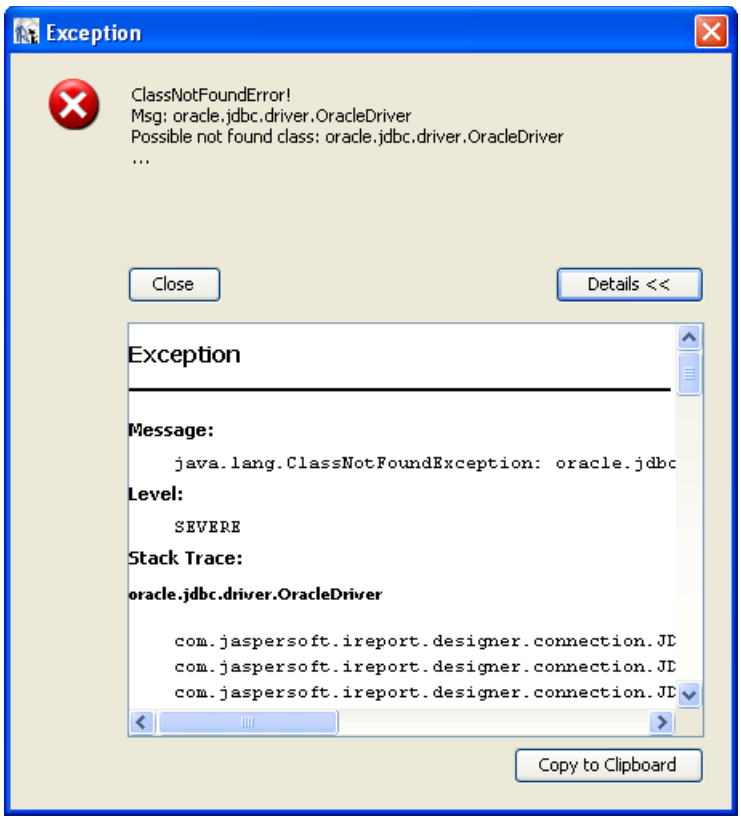



**Figure 10-6** `ClassNotFoundError` **Exception**

What you have to do is to add the JDBC driver for Oracle, which is a file named `ojdbc14.jar` (or `classes12.zip` or `classes11.zip` for older versions) to the classpath (which is where the JVM searches for classes). As iReport uses its own class loader, it will be enough add the `ojdbc14.jar` file to the iReport classpath in the options window (*Tools→Options*), the same can be done for directories containing classes and other resources.

### 10.3.2 URL Not Correct

If a wrong URL is specified (for example, due to a typing error), you'll get an arbitrary exception when you click the **Test** button. The exact cause of the error can be deduced by the stack trace available in the exception dialog box.

In this case, if possible, it is better to use the JDBC URL Wizard to build the JDBC URL and try again.

### 10.3.3 Parameters Not Correct for the Connection

The less -problematic error scenario is one where you try to establish a connection to a database with the wrong parameters (invalid username or password, nonexistent of not running database, etc.). In this case, the same database will return a message that will be more or less explicit about the failure of the connection.

### 10.3.4 Creating a JDBC Connection via the services view

iReport provides a second way to configure JDBC connection coming from the NetBeans platform. From the services view, select *New connection* (see **Figure 10-7**).

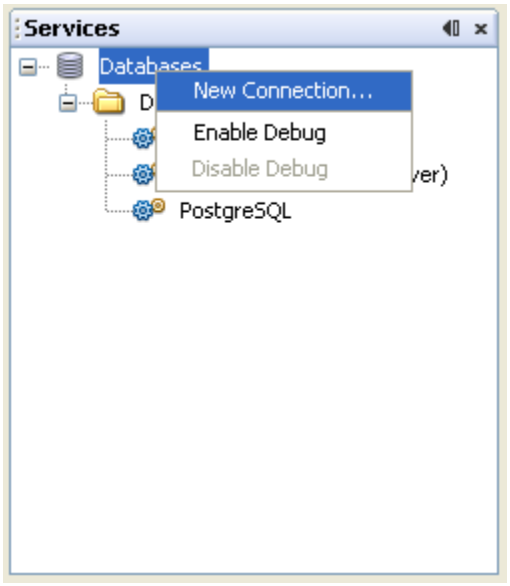



**Figure 10-7 The services view**

The services view allows you to register new JDBC drivers (by default iReport ships with drivers for MySQL, PostreSQL and the JDBC-ODBC bridge, which is not recommended).

The interface to configure a JDBC connection is similar to the one proposed by iReport and is shown in **Figure 10-8**.

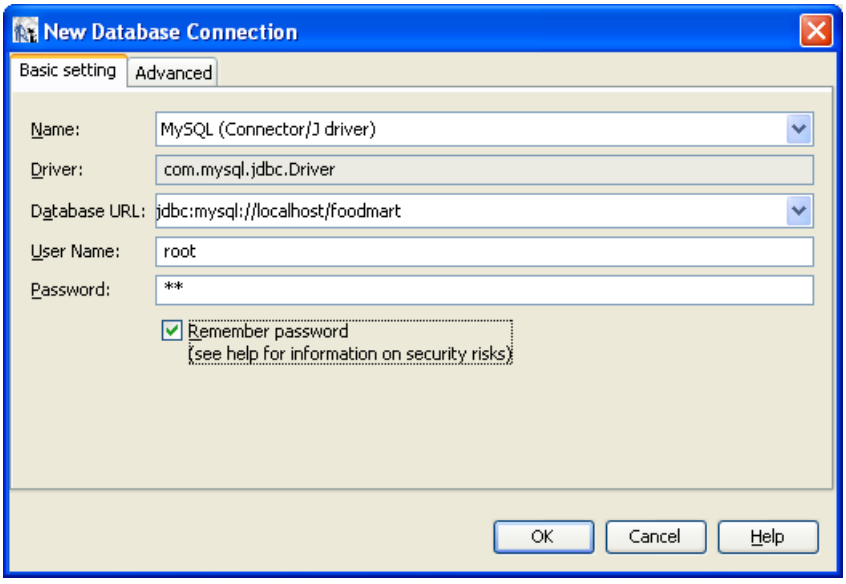



**Figure 10-8 Creation of a database connection using the services view**

When the connection has been configured, it will appear in the services view (**Figure 10-9**).

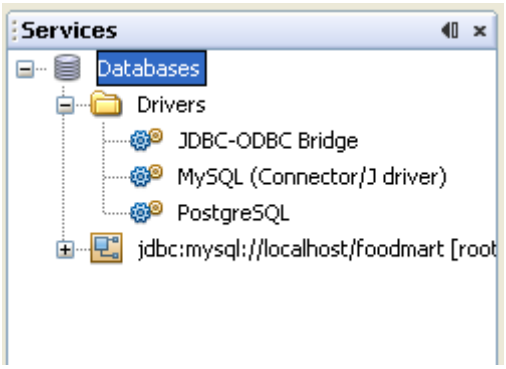



**Figure 10-9 The new connection in the services view**

The last step to use this connection to fill a report is to create a new iReport connection/data source that points the one just configured. Follow the steps indicated to create a new connection/data source and from the connection type list select *NetBeans Database JDBC connection* as shown in **Figure 10-10**.

In the following step, just select the configured connection from the combo box.

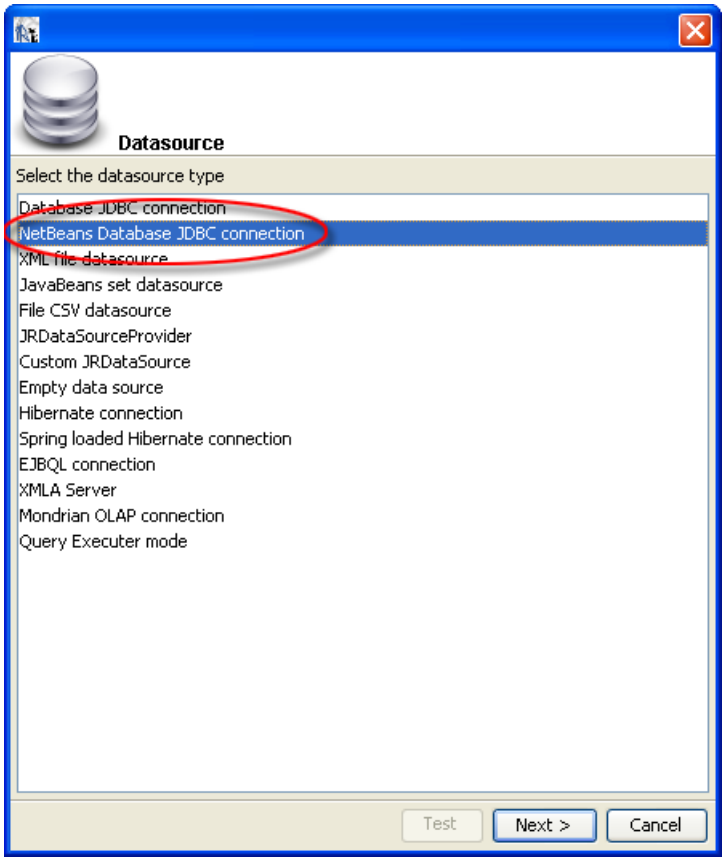



**Figure 10-10 The NetBeans “bridge” connection**

There are no distinct advantages to using one method or the other to configure a JDBC connection, it's your choice.

## 10.4 Working with Your JDBC Connection

When the report is created by using a JDBC connection, you specify a SQL query to extract from the database the records to print. This connection can also be used by a subreport or, for example, by a personalized lookup function for the decoding of a particular data. For this reason, JasperReports puts at your disposal a special parameter named REPORT_CONNECTION of the `java.sql.Connection` type that can be used in whatever expression you like, with the parameters syntax as follows:

```
$P{REPORT_CONNECTION}
```

This parameter contains exactly the `java.sql.Connection` class passed to JasperReports from the calling program.

The use of JDBC or SQL connections represents the simplest and easiest way to fill a report. (The details for how to create a SQL query are explained in Chapter **6, "Fields, Parameters, and Variables," on page 95**.

### 10.4.1 Fields Registration

In order to use SQL query fields in a report, you need to register them (it is not necessary to register all the selected fields—those effectively used in the report are enough). For each field, you must specify its name and type. Conversion of SQL and JAVA TypesTable 5 shows the mapping of the SQL types to the corresponding Java types.

**Table 10-11Conversion of SQL and JAVA Types**

| SQL Type | Java Object | SQL Type | Java Object |
|---|---|---|---|
| CHAR | *String* | REAL | *Float* |
| VARCHAR | *String* | FLOAT | *Double* |
| LONGVARCHAR | *String* | DOUBLE | *Double* |
| NUMERIC | *java.math.BigDecimal* | BINARY | *byte[]* |
| DECIMAL | *java.math.BigDecimal* | VARBINARY | *byte[]* |
| BIT | *Boolean* | LONGVARBINARY | *byte[]* |
| TINYINT | *Integer* | DATE | *java.sql.Date* |
| SMALLINT | *Integer* | TIME | *java.sql.Time* |
| INTEGER | *Integer* | TIMESTAMP | *java.sql.Timestamp* |
| BIGINT | *Long* | | |

Table 1 does not include the BLOB and CLOB types and other special types such as ARRAY, STRUCT, and REF because these types cannot be managed automatically by JasperReports. (However, it is possible to use them by declaring them generically as Object and by managing them by writing supporting static methods. The BINARY, VARBINARY, and LONGBINARY types should be dealt with in a similar way. With many databases, BLOB and CLOB can be declared as java.io.InputStream)

Whether a SQL type is converted to a Java object depends on the JDBC driver used.

For the automatic registration of SQL query fields, iReport relies on the type proposed for each field by the driver itself.

## 10.5 Sorting and Filtering Records

The records retrieved from a data source (or from the execution of a query through a connection) can be ordered and filtered. Sort and filter options may be set from the Report query dialog box by clicking the buttons labeled **Sort** options and **Filter Expressions** (see Figure 10-12).

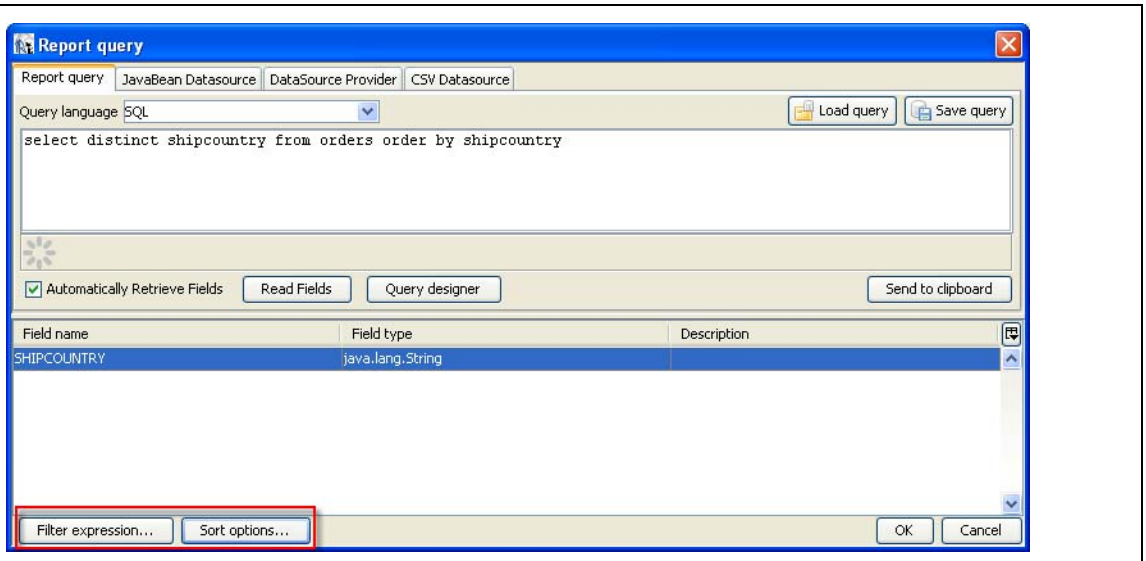



**Figure 10-12 The Filter expression and Sort options buttons**

The filter expression must return a Boolean object: `TRUE` if a particular record can be kept, `FALSE` otherwise.

The sorting is based on one or more fields. Each field can be sorted using an ascending or a descending order (see Figure 10-13).

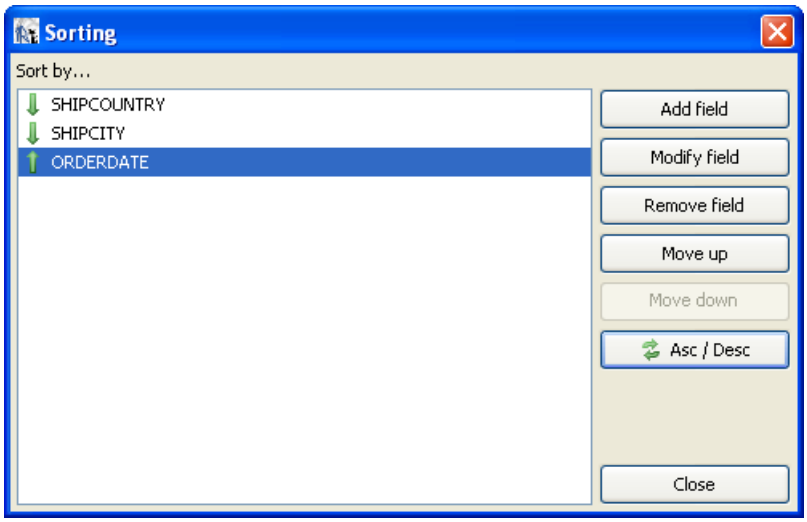



**Figure 10-13 Sort Options**

If no fields can be selected using the **Add field** button, be sure the report already contains some fields, in negative case, close the query dialog first to register the fields, the go back to the sort options.

## 10.6 Understanding the JRDataSource Interface

Before proceeding with exploration of the different data sources iReport provides at your disposal, it is necessary to understand how the `JRDataSource` interface works. Every JRDataSource must implement these two methods:

```
public boolean next()
public Object  getFieldValue(JRField jrField)
```

The first method is useful to move a virtual cursor to the next record: in fact, data supplied by a JRDataSource is ideally organized into records like in a table. The next method returns true if the cursor is positioned correctly in the subsequent record, false if there are no more available records.

Every time that JasperReports executes the method `next`, all the fields declared in the report are filled, and all the expressions (starting from those associated with the variables) are calculated again; subsequently, it will be decided whether to print the header of a new group, to go to a new page, and so on. When `next` returns false, the report is ended by printing all final bands (group footer, column footer, last page footer, and the summary). The `next` method can be called as many times as there are records present (or represented) from the data source instance.

The method `getFieldValue` is called by JasperReports after a call to `next` results in a true value. In particular, `getFieldValue` is executed for every single field declared in the report (see Chapter **6, "Fields, Parameters, and Variables," on page 95** for the details on how to declare a report field). In the call, a `JRField` object is passed as a parameter, it is used to specify the name, the description and the type of the field of which you want to obtain the value (all these information, depending by the specific data source implementation, can be combined to extract the field value).

The type of the value returned by the `getFieldValue` method has to be adequate to that declared in the `JRField` parameter, except for when a null is returned. If the type of the field was declared as `java.lang.Object`, the method can return an arbitrary type. In this case, if required, a cast can be used in the expressions. A cast is a way to dynamically indicate the type on an object, the syntax of a cast is:

```
(type)object
```

in example:

```
(com.jaspersoft.ireport.examples.beans.PersonBean)$F{my_person}
```

Usually a cast is required when you need to call a method on the object that belongs to a particular class.

## 10.7 Using JavaBeans Set Data Sources

A JavaBeans set data source allows you to use some JavaBeans as data to fill a report. In this context, a JavaBean is a Java class that exposes its attributes with a series of getter methods, with the following syntax:

```
public <returnType> getXXX()
```

where *<returnType>* (the return value) is a generic Java class or a primitive type (such as `int`, `double`, etc.).

In order to create a connection to handle JavaBeans, after clicking *New* in the **Connections/Datasources** *dialog*, select *JavaBeans set data source* in the list of data source types to bring up the dialog box shown in **Figure 10-14**.

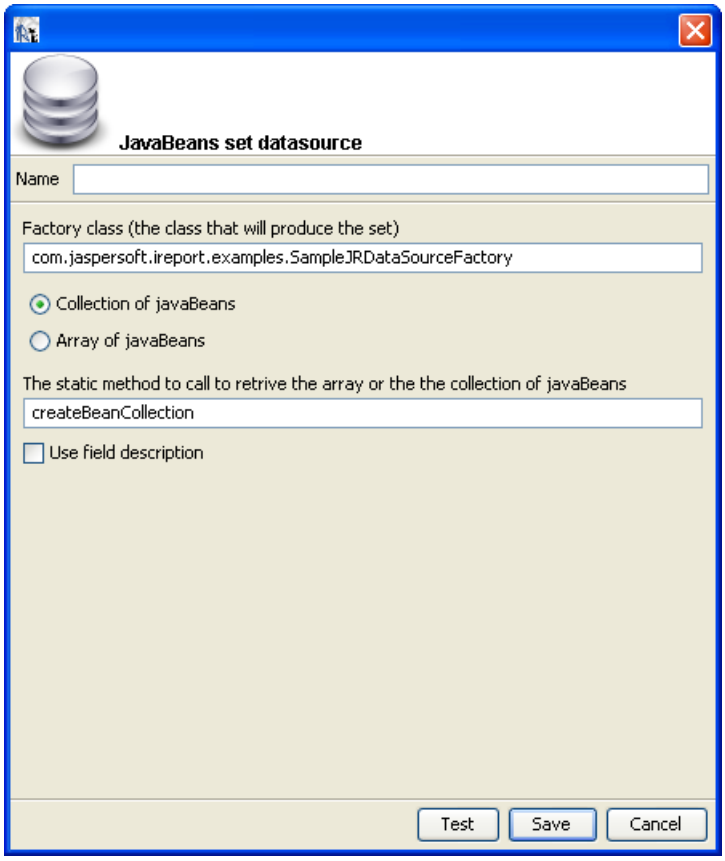



**Figure 10-14 JavaBeans set data source**

Once again, the first thing to do is to specify the name of the new data source.

The JavaBeans set data source uses an external class (named Factory) to produce some objects (the JavaBeans) that constitute the data to pass to the report. Enter your Java class (the complete name of which you specify in the Factory class field) that has a static method to instantiate different JavaBeans and to return them as a collection (`java.util.Collection`) or an array (`Object[]`). The method name and the return type have to be specified in the other fields of the window.

Let's see how to write this Factory class. Suppose that your data is represented by a set of object of type `PersonBean`; following is the code of this class, which shows two fields: name (the person's name) and age.

**Figure 10-15** `PersonBean` **Example**

```
public class PersonBean
{
 private String name = "";
 private int age = 0;

 public PersonBean(String name, int age)
 {
  this.name = name;
  this.age = age;
 }

 public int getAge()
 {
  return age;
 }

 public String getName()
 {
  return name;
 }
}
```

Your class, which you will name `TestFactory`, will be something similar to this:

**Figure 10-16** `PersonBean` **Example - Class Result**

```
public class TestFactory
{

 public static java.util.Collection generateCollection()
  {
  java.util.Vector collection = new java.util.Vector();

  collection.add(new PersonBean("Ted",  20) );
  collection.add(new PersonBean("Jack", 34) );
  collection.add(new PersonBean("Bob",  56) );
  collection.add(new PersonBean("Alice",12) );
  collection.add(new PersonBean("Robin",22) );
  collection.add(new PersonBean("Peter",28) );

  return collection;
 }
}
```

Your data source will represent five JavaBeans of `PersonBean` type.

The parameters for the data source configuration will be as follows:

- Factory name: “TestFactory"
- Factory class: `TestFactory`
- Method to call: `generateCollection`
- Return type: Collection of JavaBean

See **Figure 10-17**.

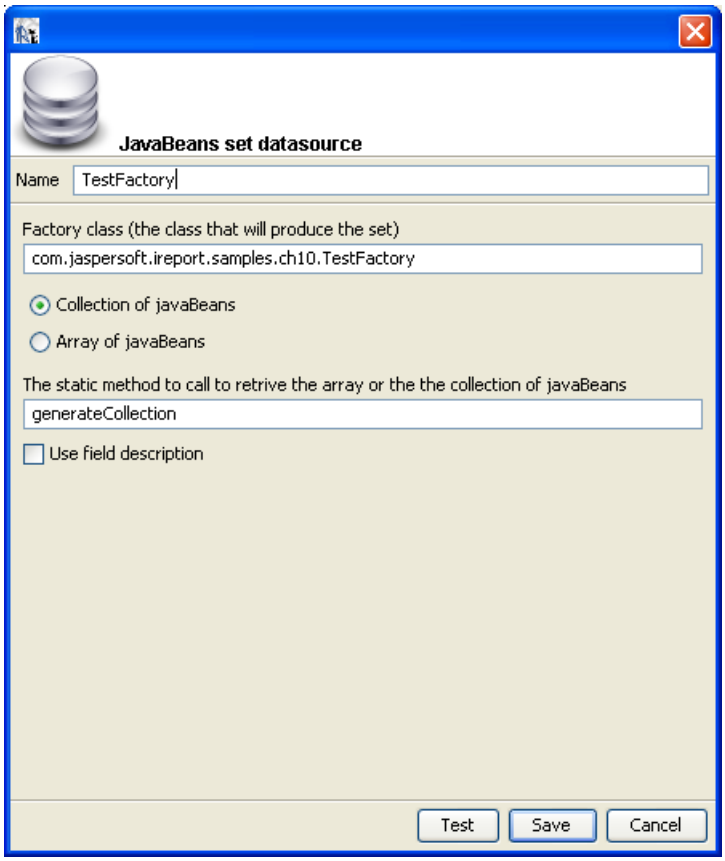



**Figure 10-17 Configuration of the factory data source**

## 10.8 Fields of a JavaBean Set Data Source

One peculiarity of a JavaBeans set data source is that the fields are exposed through “getter” methods. This means that if the JavaBean has a `getXyz()` method, xyz is the name of a record field (the JavaBean represent the record).

In this example, the PersonBean object shows two fields: name and age; register them in the fields list as String and Integer, respectively.

Create a new empty report and add the two fields by right-clicking the **Fields** node in the outline view and selecting *Add field*. The name and the type of the fields are: name (`java.lang.String`) and age (`java.lang.Integer`).

Drag the fields into the detail band and run the report (being sure the active connection is the Test Factory). **Figure 10-18** shows how your report should appear during design time, while **Figure 10-19** shows the result of the printed report filled with JavaBeans set.

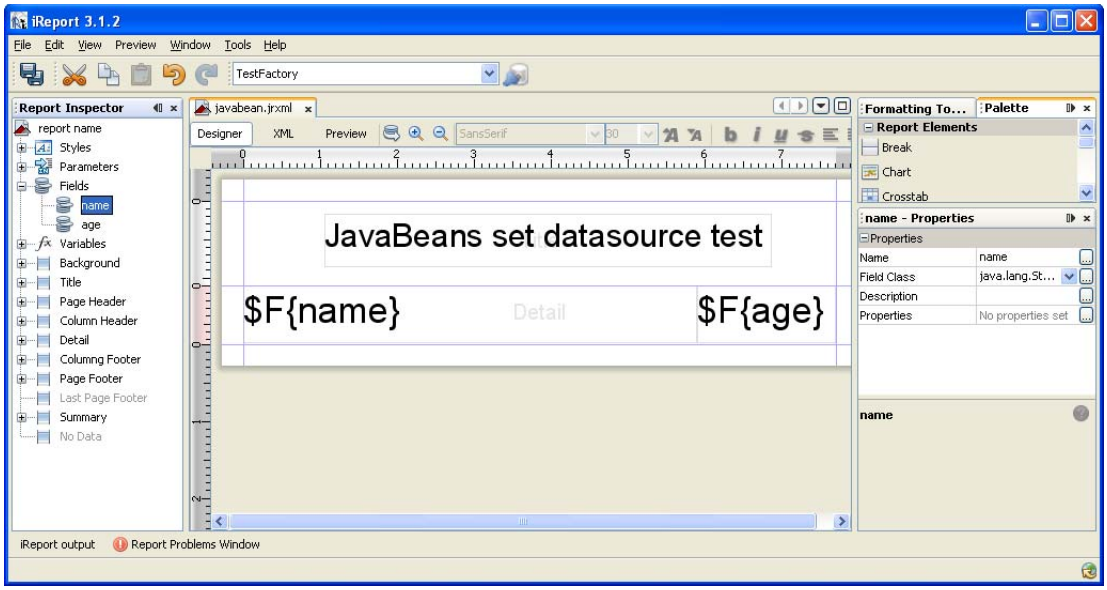



**Figure 10-18 Layout of the JavaBeans based report**

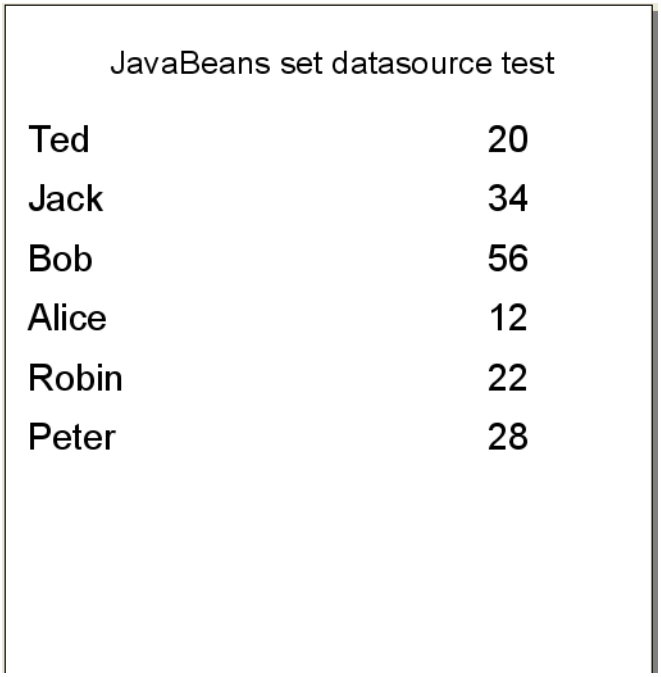



**Figure 10-19 The generated report**

To refer to an attribute of an attribute, you can use a special notation dividing attributes with points. For example, to access the street attribute of an hypothetical Address class contained in the PersonBean, you can use the syntax address.street. The real call would be `<bean>.getAddress().getStreet()`.

If the flag *Use field description* is set when you are specifying the properties of your JavaBeans set data source, the mapping between JavaBean attribute and field value is done using the field description and not the field name. In this case, the data source will consider only the description to look up the field value, and the field can have any name.

iReport provides a visual tool to map JavaBean attributes to report fields. To use it, open the query window, go to the tab JavaBean Data Source, insert the full class name of the bean you want to explore and press the *Read attributes* button. The tree below will be populated with the attributes of the specified bean class. If some attributes are java objects as well, then it can be

explored by double-clicking them to look for other attributes. To map a field, simply select an attribute name and press the *Add Selected Field(s)* button (as shown in **Figure 10-20**).

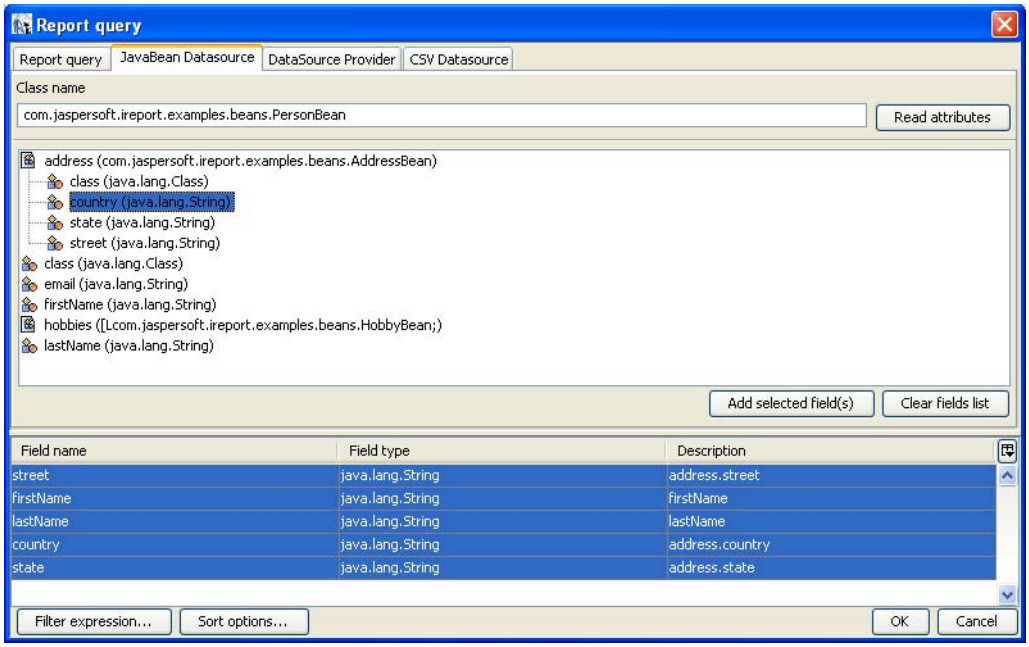



**Figure 10-20 Exploring a JavaBean to map the fields of the report**

## 10.9 Using XML Data Sources

JasperReports provides the ability of using an XML document as data source. An XML document is typically organized as a tree, and its structure hardly match the table-like form required by JasperReports. For this reason, it is necessary to make the data source know how and what nodes of the XML documents have to be selected and presented as records. To do this, you use an XPath expression to define a node set. The specifications of the XPath language are available at http://www.w3.org/TR/xpath, it is used to identify values or nodes in an XML document.

Some examples will be useful to help you to know how to define the nodes selection.

Consider the XML file in **Figure 10-21**. It is a hypothetical address book in which different people appear, grouped in categories. At the end of the categories list, a second list, of favorites objects, appears.

In this case, it is possible to define different node set types. The choice is determined by how you want to organize the data in your report.

**Figure 10-21 Example XML File**

```
<addressbook>
 <category name="home">
  <person id="1">
   <lastname>Davolio</lastname>
   <firstname>Nancy</firstname>
  </person>
  <person id="2">
   <lastname>Fuller</lastname>
   <firstname>Andrew</firstname>
  </person>
  <person id="3">
   <lastname>Leverling</lastname>
  </person>
 </category>
 <category name="work">
  <person id="4">
   <lastname>Peacock</lastname>
   <firstname>Margaret</firstname>
  </person>
 </category>
 <favorites>
  <person id="1"/>
  <person id="3"/>
 </favorites>
</addressbook>
```

To select only the people contained in the categories (that is, all the people in the address book), use the following expression:

```
/addressbook/category/person
```

Four nodes will be returned. These are shown in **Figure 10-22**.

**Figure 10-22 Node Set with Expression** `/addressbook/category/person`

```
<person id="1">
 <lastname>Davolio</lastname>
 <firstname>Nancy</firstname>
</person>
<person id="2">
 <lastname>Fuller</lastname>
 <firstname>Andrew</firstname>
</person>
<person id="3">
 <lastname>Leverling</lastname>
</person>
<person id="4">
 <lastname>Peacock</lastname>
 <firstname>Margaret</firstname>
</person>
```

If you want to select the people appearing in the favorites node, the expression to use is

```
/addressbook/favorites/person
```

The returned nodes will be two, as shown in **Figure 10-23**.

**Figure 10-23 Node Set with Expression** `/addressbook/favorites/person`

```
<person id="1"/>
<person id="3"/>
```

Here is another expression that is a bit more complex than the last example, but shows all the power of the Xpath language: the idea is to select the person nodes belonging to the work category. The expression to use is the following:

```
/addressbook/category[@name = "work"]/person
```

The expression will return only one node, that with an ID equal to 4, as shown in **Figure 10-24**.

**Figure 10-24 Node Set with Expression** `/addressbook/category[@name = "work"]/person`

```
<person id="4">
 <lastname>Peacock</lastname>
 <firstname>Margaret</firstname>
</person>
```

After you have created an expression for the selection of a node set, you proceed to the creation of an XML data source.

Open the window for creating a new data source and select XML File data source from the list of connection types to bring up the dialog box shown in **Figure 10-25**.

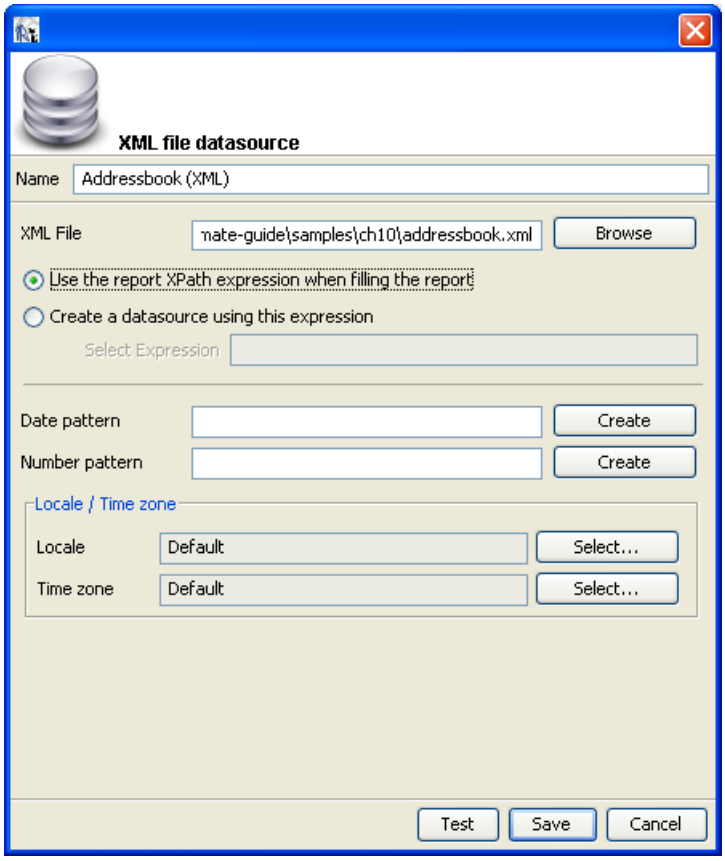



**Figure 10-25 Configuring an XML data source**

The only mandatory information to specify is the XML file name. You can provide to the engine a set of nodes selected using a predefined static XPath expression. Alternatively, the XPath expression can be set directly inside the report. I always suggest that you use a report-defined Xpath expression. The advantage of this solution is the ability to use parameters inside the XPath expression, which acts like a real query on the supplied XML data.

Optionally, you can specify Java patterns to convert dates and numbers from plain strings to more appropriate Java objects (like Date and Double). For the same purpose, you can define a specific locale and time zone to use when parsing the XML stream.

## 10.10 Registration of the Fields for an XML Data Source

In the case of an XML data source, the definition of a field in the report needs, besides the type and the name, a particular expression inserted as a field description. As the data source aims always to one node of the selected node set, the expressions are “relative” to the current node.

To select the value of an attribute of the current node, use the following syntax:

@<name attribute>

For example, to define a field that must point to the *id* attribute of a person (attribute id of the node person), it is sufficient to create a new field, name it as you want, and set the description to

@id

Similarly, it is possible to get to the child nodes of the current node. For example, if you want to refer to the `lastname` node, child of person, use the following syntax:

```
lastname
```

To move to the parent value of the current node (for example, to determine the category to which a person belongs), use a slightly different syntax:

```
ancestor::category/@name
```

The ancestor keyword indicates that you are referring to a parent node of the current node; in particular, you are referring to the first parent of category type, of which you want to know the value of the name attribute.

Now, let's see everything in action. Prepare a simple report with the registered fields shown in Table 10-26.

**Figure 10-26 Registered Fields for Example Report**

| Field name | Description | Type |
|---|---|---|
| `id` | @id | `Integer` |
| `lastname` | lastname | `String` |
| `firstname` | firstname | `String` |
| `name of category` | ancestor::category/@name | `String` |

iReport provides a visual tool to map XML nodes to report fields: to use it, open the query window and select XPath as query language. If the active connection is a valid XML DataSource, the associated XML document will be shown in a tree-view. To register the fields, set the record node by right-clicking a Person node and selecting the menu item Set record node (as shown in Figure 10-27): the record nodes will become bold. Then one by one, select the nodes or attributes and select the pop-up menu item *Add node as field* to map them to report fields. iReport will guess the correct XPath expression to use and will create the fields for you. You can modify the generated field name and set a more suitable field type after the registration of the field in the report (that happens when you close the query dialog window).

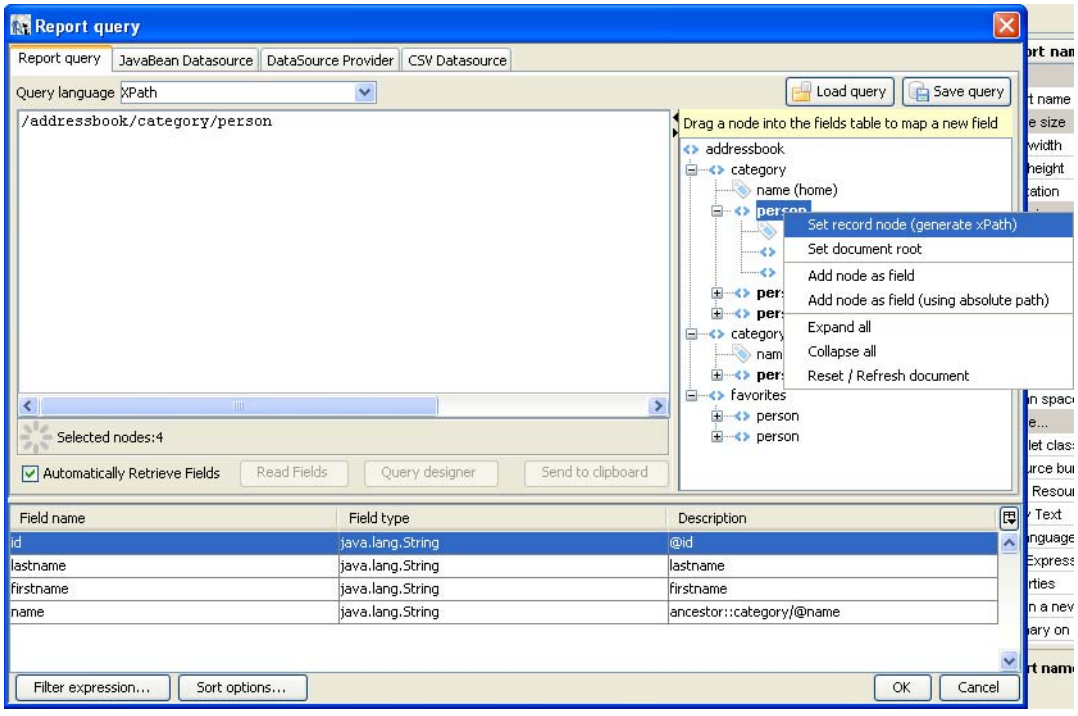



**Figure 10-27 The XML node mapping tool**

Insert the different fields into the detail band (as shown in **Figure 10-28**). The XML file used to fill the report is that shown in Table 2.

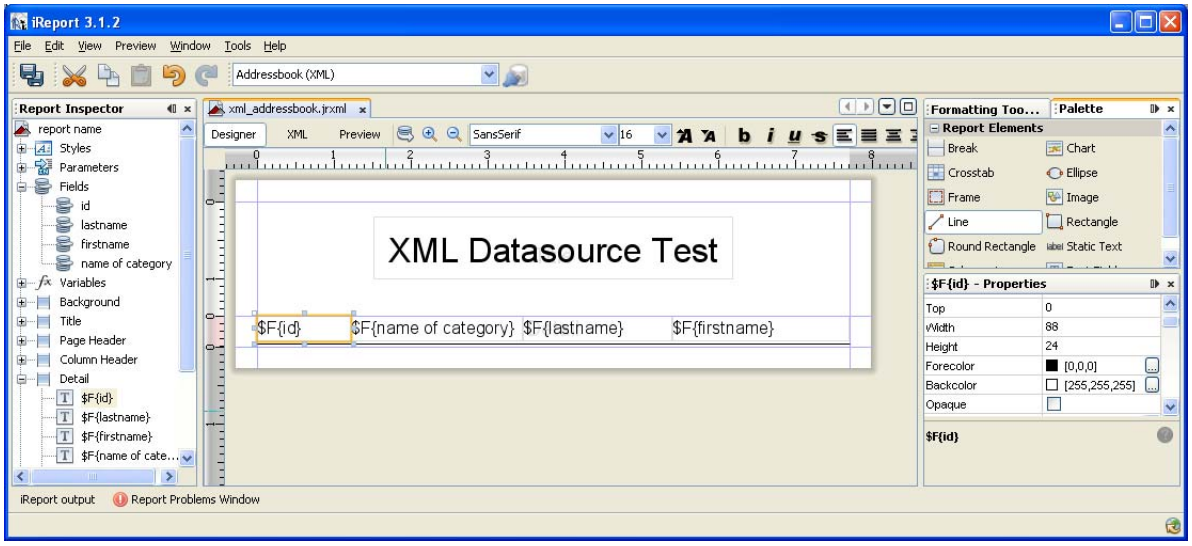



**Figure 10-28 The XML based report layout**

The XPath expression for the node set selection specified in the query dialog is:

```
/addressbook/category/person
```

The final result appears in **Figure 10-29**.

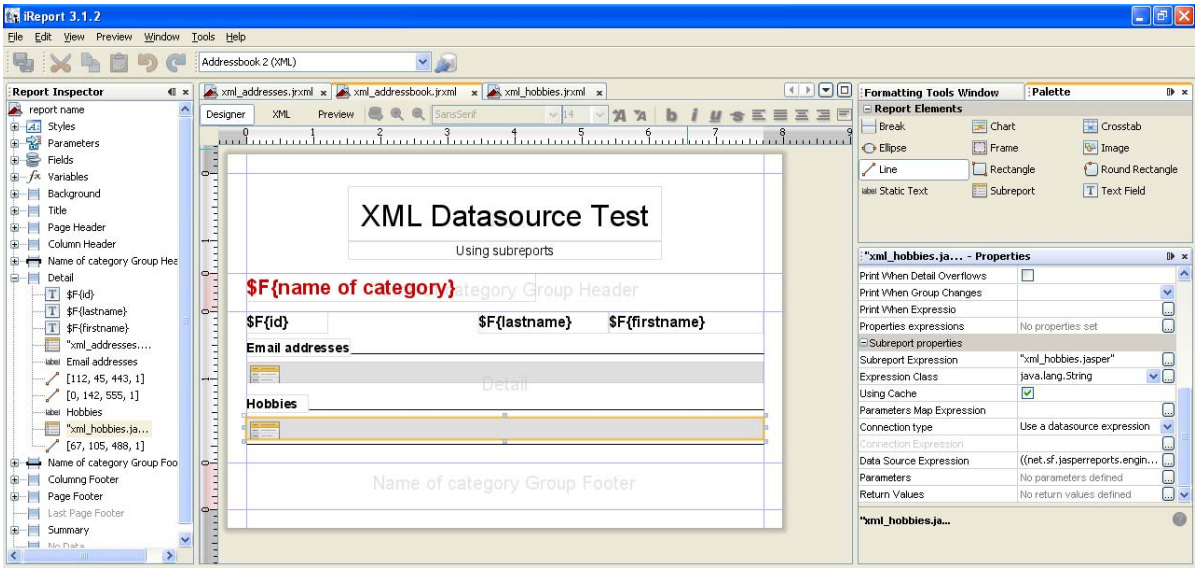



**Figure 10-29 The result of the XML based report**

## 10.11 XML Data Source and Subreports

A node set allows you to identify a series of nodes that represent, from a `JRDataSource` point of view, some records. However, due to the tree-like nature of an XML document, it may be necessary to see other node sets that are subordinated to the main nodes. Consider the XML in **Figure 10-30**. This is a slightly modified version of the document presented in Listing

10-1: for each person node, a hobbies node is added, which contains a series of hobby nodes, and one or more e-mail addresses.

**Figure 10-30 Complex XML Example**

```
<addressbook>
 <category name="home">
  <person id="1">
   <lastname>Davolio</lastname>
   <firstname>Nancy</firstname>
   <email>davolio1@sf.net</email>
   <email>davolio2@sf.net</email>
   <hobbies>
               <hobby>Music</hobby>
               <hobby>Sport</hobby>
   </hobbies>
  </person>
  <person id="2">
   <lastname>Fuller</lastname>
   <firstname>Andrew</firstname>
   <email>af@test.net</email>
   <email>afullera@fuller.org</email>
   <hobbies>
               <hobby>Cinema</hobby>
               <hobby>Sport</hobby>
   </hobbies>
  </person>
 </category>
 <category name="work">
  <person id="3">
   <lastname>Leverling</lastname>
   <email>leverling@xyz.it</email>
  </person>
  <person id="4">
   <lastname>Peacock</lastname>
   <firstname>Margaret</firstname>
   <email>margaret@foo.org</email>
   <hobbies>
               <hobby>Food</hobby>
               <hobby>Books</hobby>
   </hobbies>
  </person>
 </category>
 <favorites>
  <person id="1"/>
  <person id="3"/>
 </favorites>
</addressbook>
```

What you want to produce is a document more elaborate than those you have seen until now: for each person, you want to present the list of the e-mail addresses and the hobbies.

To obtain such a document, it is necessary to use subreports—in particular, you will need a subreport for the e-mail addresses list, one for hobbies, and one for favorite people (that is a set of nodes out of the scope of the XPath query we used). To generate these subreports, you need to understand how to produce new data sources to feed them. In this case, you use the `JRXmlDataSource`, which exposes two extremely useful methods:

```
public JRXmlDataSource dataSource(String selectExpression)
```

and

```
public JRXmlDataSource subDataSource(String selectExpression)
```

The difference between the two is that the first method process the expression applying it to the whole document (starting from the root) while the second assumes as document root the current node.

Both methods can be used in the data source expression of a Subreport element to produce dynamically the data source to pass to that element. The most important thing is that this mechanism allows you to make the data source production and the expression of node selection dynamic.

In this case, the expression to create the data source that will feed the subreport of the e-mail addresses will be

```
((net.sf.jasperreports.engine.data.JRXmlDataSource)
   $P{REPORT_DATA_SOURCE}).subDataSource("/person/email")
```

which says, “Starting from the present node (person), give me back all the e-mail nodes that are direct descendants.”

The expression for the hobbies subreport will be similar, except for the node selection:

```
((net.sf.jasperreports.engine.data.JRXmlDataSource)
   $P{REPORT_DATA_SOURCE}).subDataSource("/person/hobbies/hobby")
```

You declare the master report’s fields as shown in Table 2 earlier. In the subreport, you have to refer to the current node value, so the field expression will be simply a dot (.), as shown in **Figure 10-31**.

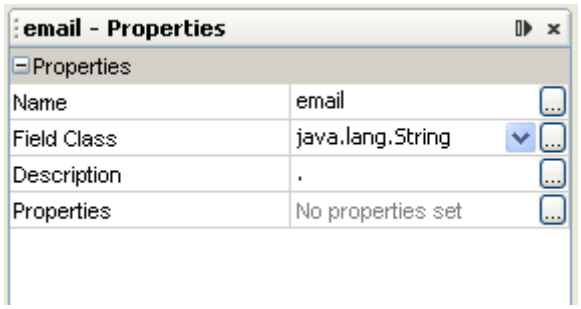



**Figure 10-31 Mapping of the e-mail field in the subreport**

Proceed with building your three reports: `xml_addressbook.jasper`, `xml_addresses.jasper` and `xml_hobbies.jasper`.

In the master report, xml_addressbook.jrxml, insert a group named “Name of category”, of which you associate the expression for the category field ($F{name of category}), as shown in **Figure 10-32**. In the Name of category header band, insert a field

through which you will view the category name. By doing this, the names of the different people will be grouped by category (as in the XML file).

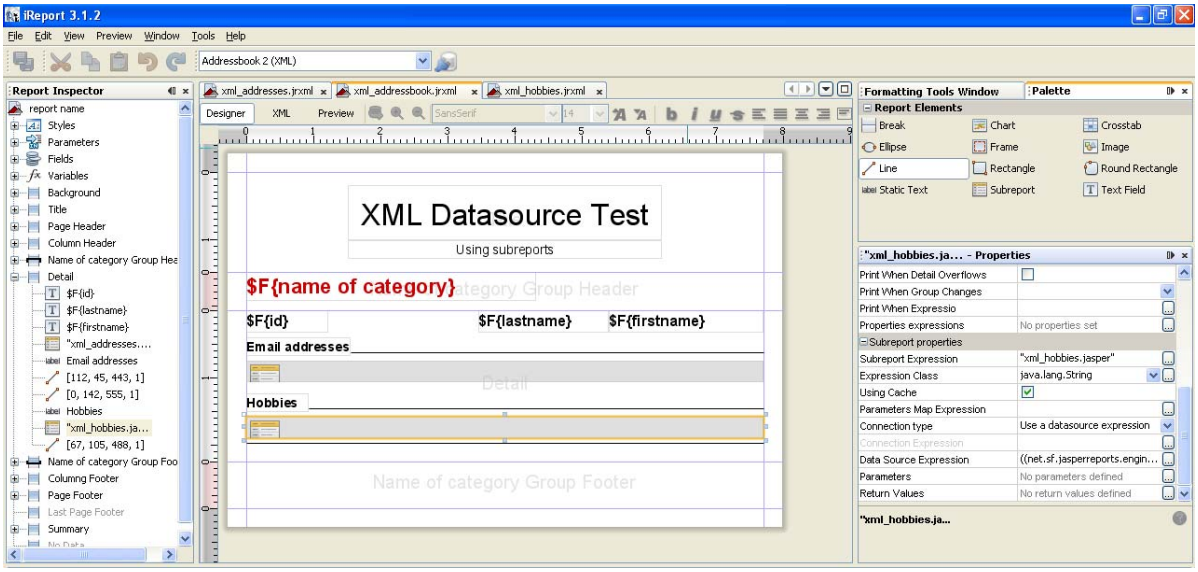



**Figure 10-32 The master report with the two subreports for addresses and hobbies**

In the detail band, position the id, lastname, and firstname fields. Underneath these fields, add the two Subreport elements, the first for the e-mail addresses, the second for the hobbies.

The e-mail and hobby subreports are identical except for the name of the only field present in these subreports (see **Figure 10-31**). The two reports should be as large as the Subreport elements in the master report, so remove the margins and set the report width accordingly.

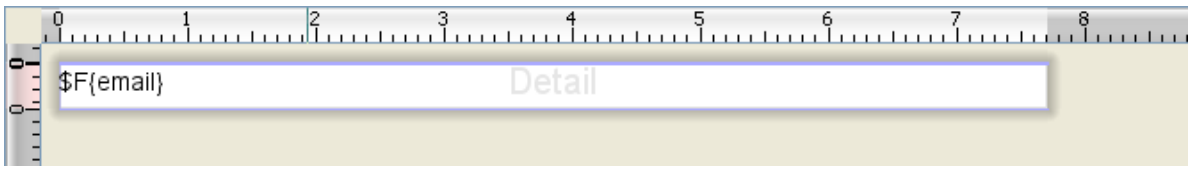



**Figure 10-33 The subreport layout**

Preview both the subreports. The purpose is just to compile them and generate the relative jasper files, if you get an error during the fill process, it's OK, we have not set an Xpath query so JasperReports get confused because is not able to get any data. You can workaround the problem setting a simple Xpath query (that will not be used in the final report) or you can preview the subreport using an empty data source (you have to select it from the combo box in the tool bar).

When the subreports are done, execute the master report. If everything is OK, you will see the print shown in **Figure 10-34**, which displays the groups of people in home and work categories and the subreports associated with every person on the list.

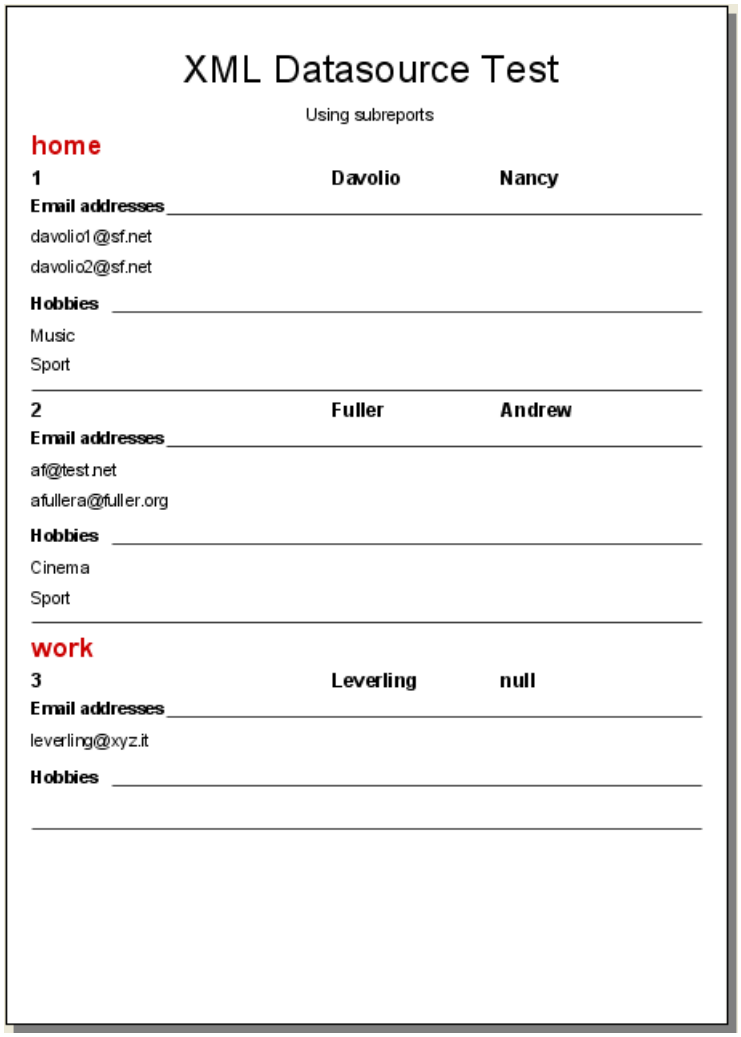



**Figure 10-34 The first page of the final result**

As this example demonstrates, the real power of the XML data source is the versatility of XPath, which allows navigating the node selection in a refined manner.

## 10.12 Using CSV Data Sources

Initially, the data source for CSV documents was a very simple data source proof of concept to show how to implement a custom data source. The CSV data source interface was improved when JasperReports added a native implementation to fill a report using a CSV file.

To create a connection based on a CSV file, click the *New* button in the **Connections/Datasources** dialog box and select File CSV data source from the data source types list to bring up the dialog box shown in **Figure 10-35**.

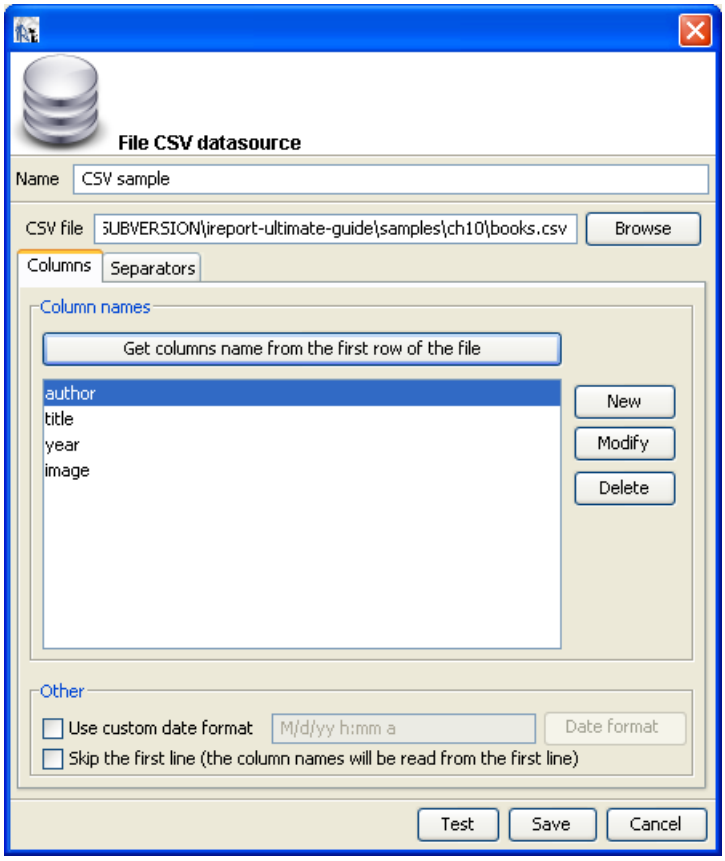



**Figure 10-35 CSV Data Source**

Set a name for the connection and choose a CSV file. Then declare the fields in this data source. If the first line in your file contains the names for each column, click the *Get column names from the first row of the file* button and select the *Skip the first line* check box option. This forces JasperReports to skip the first line (the one containing your column labels). In any case, the column names read from the file are used instead of the declared ones, so avoid modifying the names found with the Get column names button.

If the first line of your CSV file doesn't contain the column names, set a name for each column using the syntax COLUMN_0, COLUMN_1 and so on.

If you define more columns than the ones available, you'll get an exception at report filling time.

JasperReports assumes that for each row all the columns have a value (even if empty).

If your CSV file uses nonstandard characters to separate fields and rows, you can adjust the default setting for separators using the Separators tab, shown in **Figure 10-36**.

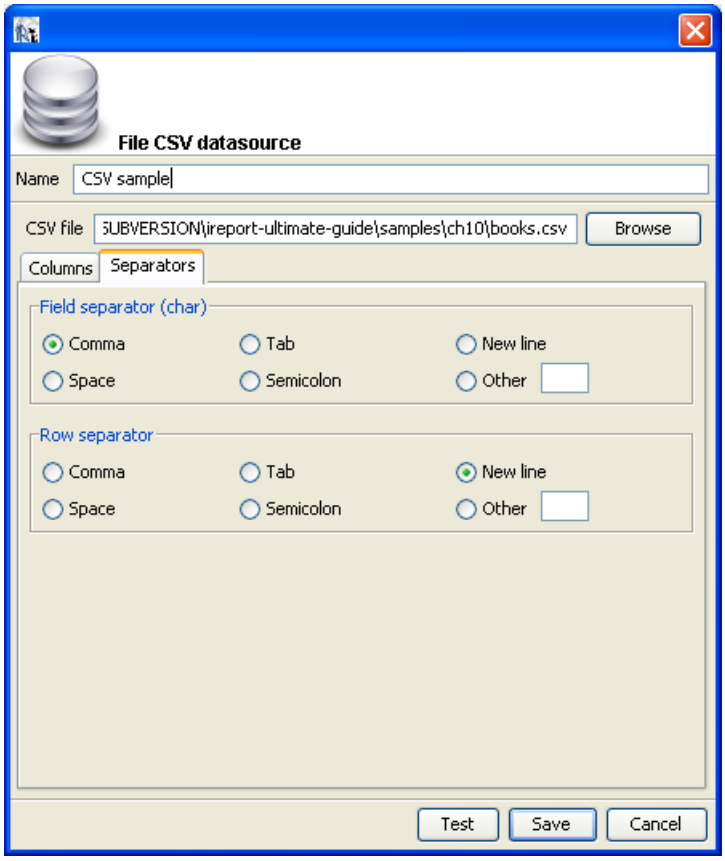



**Figure 10-36 Column and row separators**

## 10.13 Registration of the Fields for a CSV Data Source

When you create a CSV data source, you must define a set of column names, which will be used as fields for your report. To add them to the fields list, set your CSV data source as the active connection and open the Report query dialog box. Go to the tab labeled **CSV Datasource** and click the **Get fields from data source** button, as shown in **Figure 10-37**.

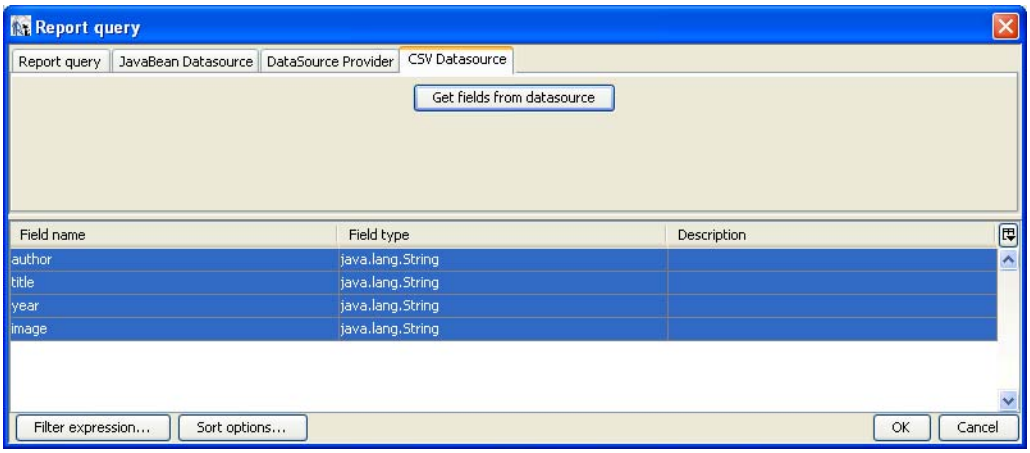



**Figure 10-37 Registering CSV file fields**

By default, iReport sets the class type of all fields to `java.lang.String`. If you are sure that the text of a particular column can be easily converted to a number, a date, or a Boolean value, set the correct field type yourself once your fields are added to your report.

The pattern used to recognize a timestamp (or date) object can be configured at the data source level by selecting the *Use custom date format* check box option.

## 10.14 Using JREmptyDataSource

JasperReports provides a special data source named `JREmptyDataSource`.

This source returns true to the `next` method for the record number (by default only one), and always returns null to every call of the `getFieldValue` method. It is like having some records without fields, that is, an empty data source.

The two constructors of this class are:

```
public JREmptyDataSource(int count)
public JREmptyDataSource()
```

The first constructor indicates how many records to return, and the second sets the number of records to one.

By default iReport provides a pre-configured empty data source that returns a single record. To create a new empty data source with more records, select the *Empty* Datasource from the list of available connection types, you will prompt with the interface shown in **Figure 10-38**.

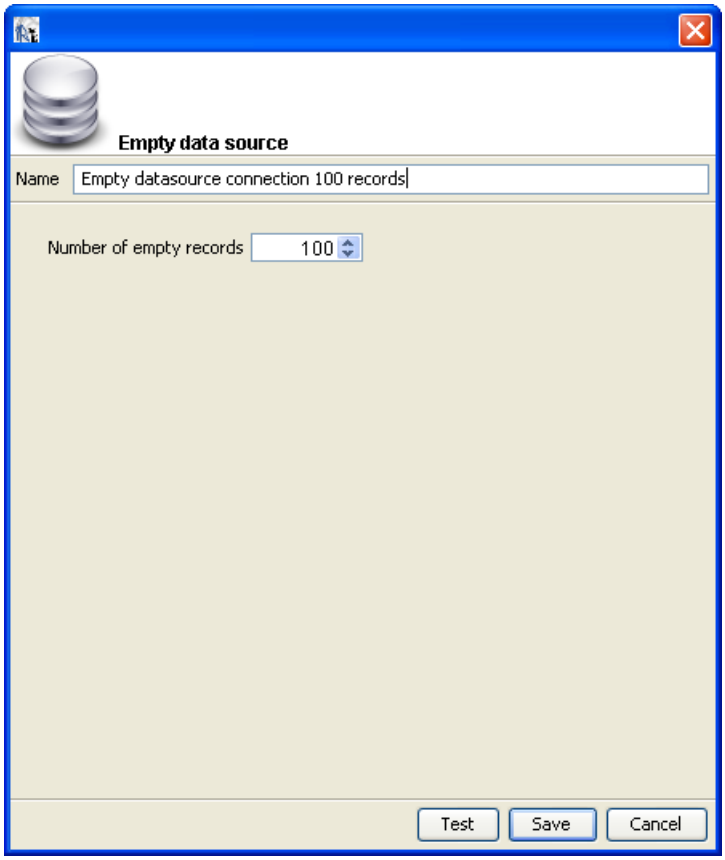



**Figure 10-38 Empty data source**

Set the number or empty records you need. Remember, whatever field you will add to the report, it's value will be set to null. Since the data source does not care about field names or type, it is perfect to test any report (keeping in mind the fields will be always set to null).

## 10.15 Using HQL and Hibernate Connections

JasperReports provides a way to use HQL directly in your report. To do so, first set up a Hibernate connection. Expand your classpath to include all classes, JARs, and configuration files used by your Hibernate mapping. In other words, iReport must be able to access all the *.hbm.xml files you plan to use, the JavaBeans declared in those files, the `hibernate.cfg.xml` file, and other optional JARs used (e.g., those that access the database under Hibernate).

To add these objects to the classpath, select **Tools→Options** and move to the classpath tab.

Once you've expanded the classpath, open the Connections/Datasources dialog box, click the New button, and choose the Hibernate connection as your data source type. This brings up the dialog box shown in **Figure 10-39**.

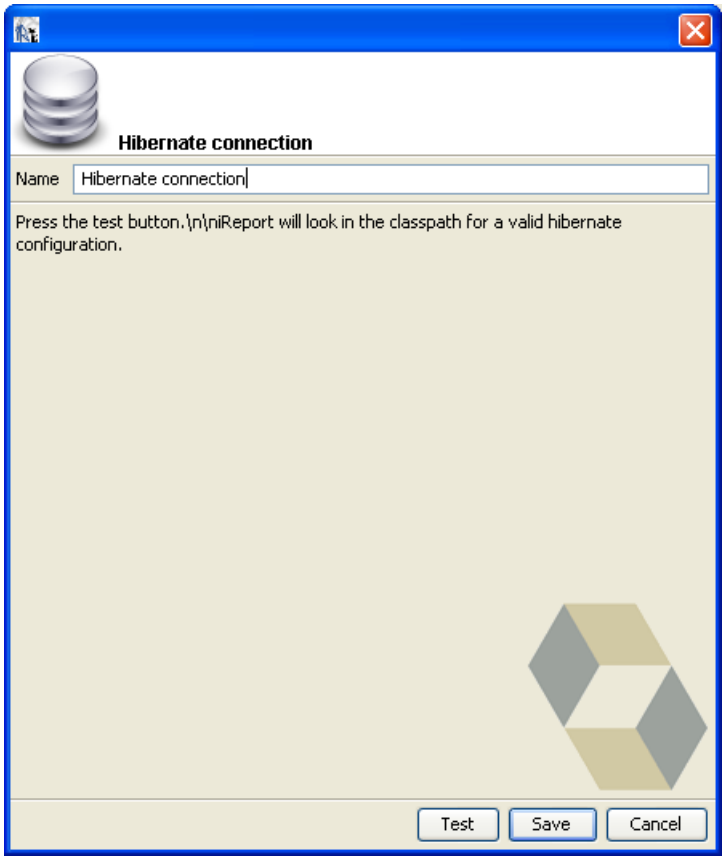



**Figure 10-39 Hibernate connection**

Click the **Test** button to check the path resolution, so that you are certain that `hibernate.cfg.xml` is in the classpath.

Currently, iReport works only with a single Hibernate configuration (that is, the first hibernate.cfg.xml file found in the classpath).

If you use the Spring framework, you can use a Spring configuration file to define your connection. In this case, you'll need to set the configuration file name and the Session Factory Bean ID (see **Figure 10-40**).

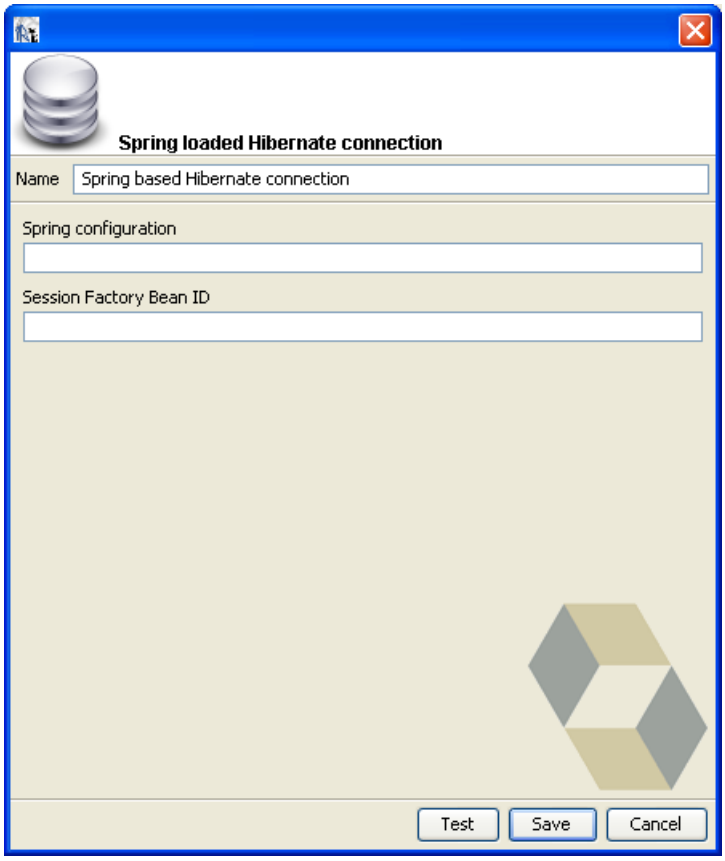



**Figure 10-40 Spring based Hibernate connection**

Now that a Hibernate connection is available, use an HQL query to select the data to print. You can use HQL in the same way you use SQL: open the Report query dialog box and choose HQL as the query language from the combo box on top (see **Figure 10-41**).

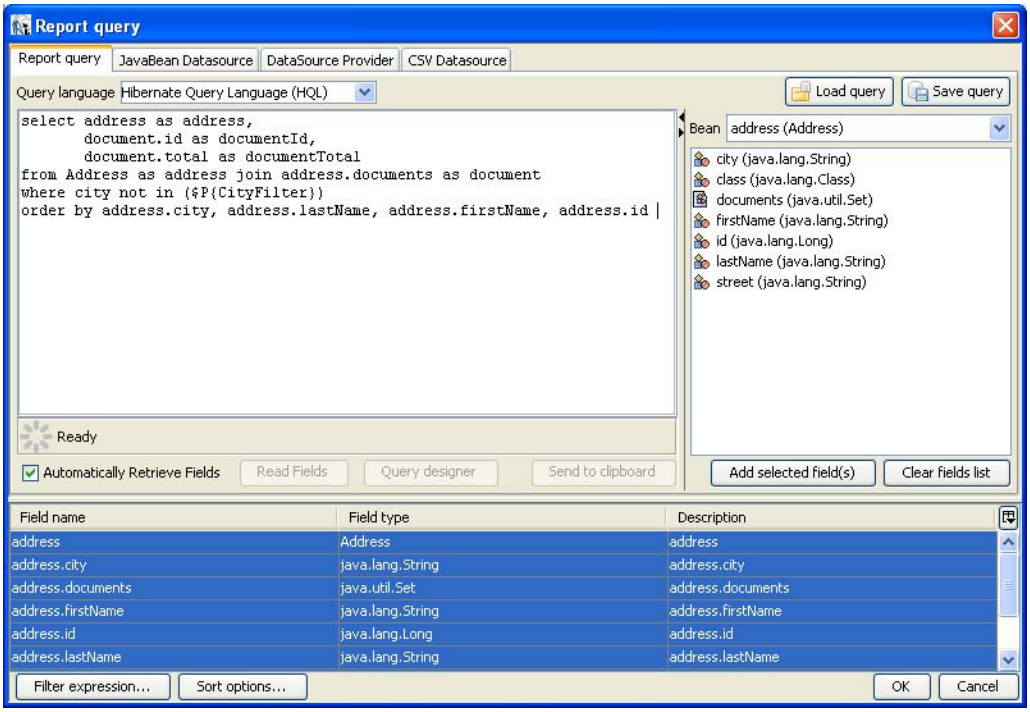



**Figure 10-41 The HQL and the HQL mapping tool**

When you enter an HQL query, iReport tries to retrieve the available fields. According to the JasperReports documentation, the field mappings are resolved as follows:

- If the query returns one object per row, a field mapping can be one of the following:
  - If the object's type is a Hibernate entity or component type, the field mappings are resolved as the property names of the entity/component. If a select alias is present, it can be used to map a field to the whole entity/component object.
  - Otherwise, the object type is considered scalar, and only one field can be mapped to its value.
- If the query returns a tuple (object array) per row, a field mapping can be one of the following:
  - A select alias: The field will be mapped to the value corresponding to the alias.
  - A property name prefixed by a select alias and a ".": The field will be mapped to the value of the property for the object corresponding to the alias. The type corresponding to the select alias has to be an entity or component.

If you don't understand what this means, simply accept the fields listed by iReport when the query is parsed.

iReport provides a mapping tool to map objects and attributes to report fields. The objects (or JavaBeans) available in each record are listed in the combo box on top of the object tree.

To add a field from the tree, select the corresponding node and click the *Add selected field(s)* button.

## 10.16 How to Implement a New JRDataSource

Sometimes the `JRDataSource` supplied with JasperReports cannot satisfy completely your needs. In these cases, it is possible to write a new `JRDataSource`. This operation is not complex: in fact, all you have to do is create a class that implements the `JRDataSource` interface (see Listing 10-6) that exposes two simple methods: `next` and `getFieldValue`.

```
package net.sf.jasperreports.engine;
```

```
public interface JRDataSource
{
      public boolean next() throws JRException;
      public Object  getFieldValue(JRField jrField) throws JRException;
}
```

**Figure 10-42 The JRDataSource Interface**

The `next` method is used to set the current record into the data source. It has to return true if a new record to elaborate exists and false otherwise.

If the `next` method has been called positively, the `getFieldValue` method has to return the value of the requested field (or `null` if the requested value is not found or does not exist). In particular, the requested field name is contained in the `JRField` object passed as a parameter. Also, `JRField` is an interface through which it is possible to get the information associated to a field: the name, the description, and the Java type that represents it (as mentioned previously in the "Understanding the JRDataSource Interface" section earlier).

Now try writing your personalized data source. The idea is a little original: you have to write a data source that explores the directory of a file system and returns the found objects (files or directories). The fields you will make to manage your data source will be the file name, which you will name FILENAME; a flag that indicates whether the object is a file or a directory, which you will name IS_DIRECTORY; and the file size, if available, which you will name SIZE.

There will be two constructors for your data source: the former will receive as a parameter the directory to scan, the latter will not have parameters (and will use the current directory to scan).

Just instantiated, the data source will look for the files and the directories present in the way you indicate, filled the array files.

The next method will increase the index variable that you will use to keep track of the position reached in the array files and it will return true until you reach the end of the array.

```
import net.sf.jasperreports.engine.*;
import java.io.*;

public class JRFileSystemDataSource implements JRDataSource
{
File[] files = null;
int    index = -1;

public JRFileSystemDataSource(String path)
{
File dir = new File(path);
if (dir.exists() && dir.isDirectory())
{
files = dir.listFiles();
}
}

public JRFileSystemDataSource()
{
this(".");
}
```

```
public boolean next() throws JRException
{
index++;
if (files != null && index < files.length)
{
return true;
}
return false;
}



        public Object  getFieldValue(JRField jrField) throws JRException
        {
        File f = files[index];
        if (f == null) return null;
        if (jrField.getName().equals("FILENAME"))
        {
        return f.getName();
        }
        else if (jrField.getName().equals("IS_DIRECTORY"))
        {
        return new Boolean(f.isDirectory());
        }
        else if (jrField.getName().equals("SIZE"))
        {
        return new Long(f.length());
        }
        // Field not found...
        return null;
        }
}
```

The `getFieldValue` method will return the requested file information. Your implementation does not use the information regarding the return type expected by the caller of the method, but it assumes that the name has to be returned as a string, the flag IS_DIRECTORY as a Boolean object, and the file size as a Long object. In the next section, you will learn how to use your personalized data source in iReport and test it.

## 10.17 Using a Personalized JRDataSource with iReport

iReport provides support for almost all the data sources provided by JasperReports like `JRXmlDataSource`, `JRBeanArrayDataSource` and `JRBeanCollectionDataSource`.

To use your personalized data sources, a special connection is provided; it is useful for employing whichever `JRDataSource` you want to use through some kind of factory class that provides an instance of that JRDataSource implementation. These factory is just a simple Java class useful to test your data source and use it to fill a report in iReport.

The idea is the same as what you have seen for the JavaBeans set data source: it is necessary to write a Java class that creates the data source through a static method and returns it.

For example, if you want to test the `JRFileSystemDataSource` in the previous section, you need to create a simple class like that shown in Listing 10-8.

```
import net.sf.jasperreports.engine.*;


public class FileSystemDataSourceFactory {


      public static JRDataSource createDatasource()
{
return new JRFileSystemDataSource("/");


}
}
```

**Figure 10-43 Listing 9-8. Class for Testing a Personalized Data Source**

This class, and in particular the static method that will be called, will execute all the necessary code for instancing correctly the data source. In this case, you create a new `JRFileSystemDataSource` object by specifying a way to scan the directory root (“/”).

Now that you have defined the way to obtain a `JRDataSource` that you have prepared and is ready to be used, create the connection through which it will be used.

Create a new connection as you normally would, select Custom JRDataSource from the data source type list, and specify a data source name such as `TestFileSystemDataSource` (or whatever name you wish), as shown in **Figure 10-44**.

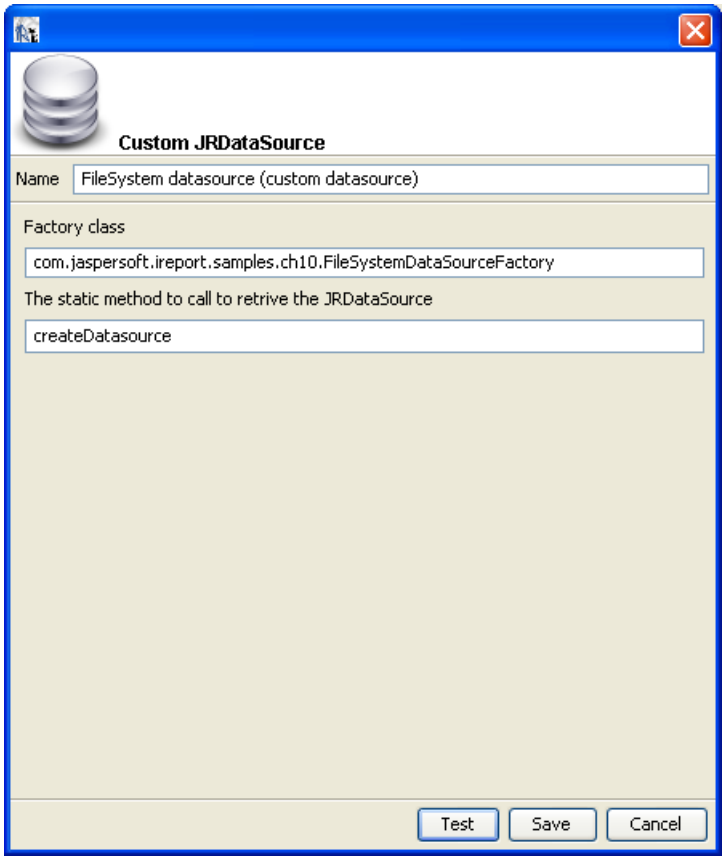



**Figure 10-44 Configuration of the custom data source**

Next, specify the class and the method to use to obtain an instance of your JRFileSystemDataSource: TestFileSystemDataSource and test.

Prepare a new report with fields managed by the data source. No method to find the fields managed by a data source exists. In this case, you know that the `JRFileSystemDataSource` provides three fields and their names and types are: FILENAME (String), IS_DIRECTORY (Boolean), and SIZE (Long). After you have created these fields, insert them in the report's detail band as shown in **Figure 10-45**.


**Figure 10-45 The layout for the file list**

Divide the report into two columns, and in the column header band insert File name and Size tags. Then add two images, one representing a document and the other an open folder. In the Print when expression setting of the Image element that is placed in the foreground, insert the expression `$F{IS_DIRECTORY}` or use as image expression a condition like:

```
($F{IS_DIRECTORY}) ? “folder.png” : “file.png”
```

The final report is shown in **Figure 10-46**.

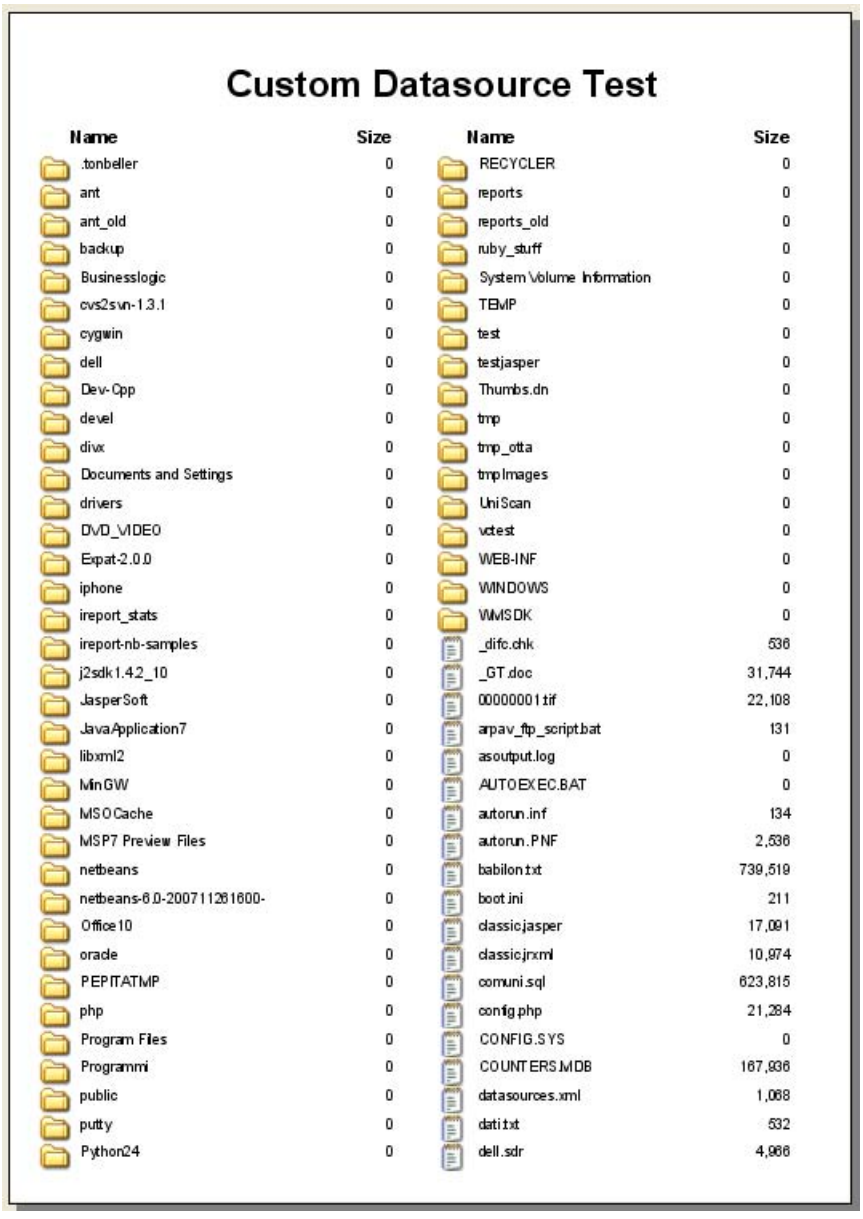



**Figure 10-46 The result produced with the custom data source**

In this example, the class that instantiated the `JRFileSystemDataSource` was very simple. However, you can use more complex classes, such as one that obtains the data source by calling an Enterprise JavaBean, or by calling a web service.

## 10.18 Importing and Exporting Data Sources

To simplify the process of sharing data source configurations, iReport provides a mechanism to export and import data source definitions.

To export one or more data sources, select from the **Connections/Datasources** window the items to export and click the **Export** button (see **Figure 10-47**). iReport will ask you to name the file and indicate the destination where to store the exported information. The created file is a simple XML file and can be edited with a common text editor, if needed. A file exported with iReport can be imported by clicking Import. Since an exported file can contain more than one data source or connection definition, the import process will add all the data sources found in the specified file to the current list.

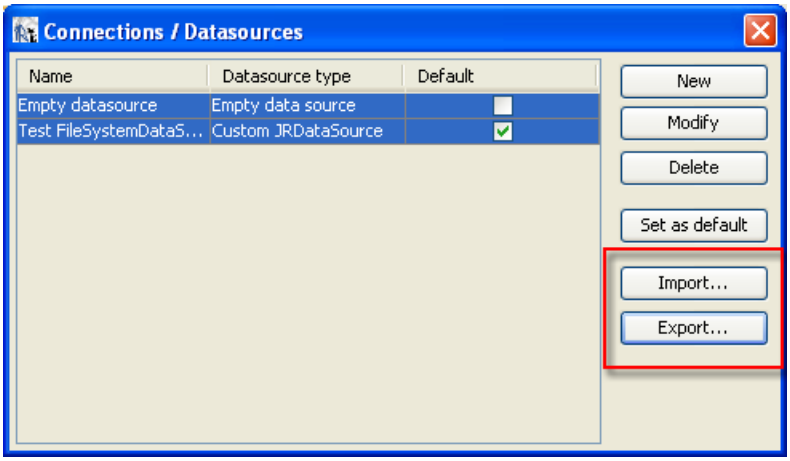



**Figure 10-47 Export connection and data source definitions**

If a duplicated data source name is found during the import, iReport will append a number to the imported data source name, as shown in **Figure 10-48**.

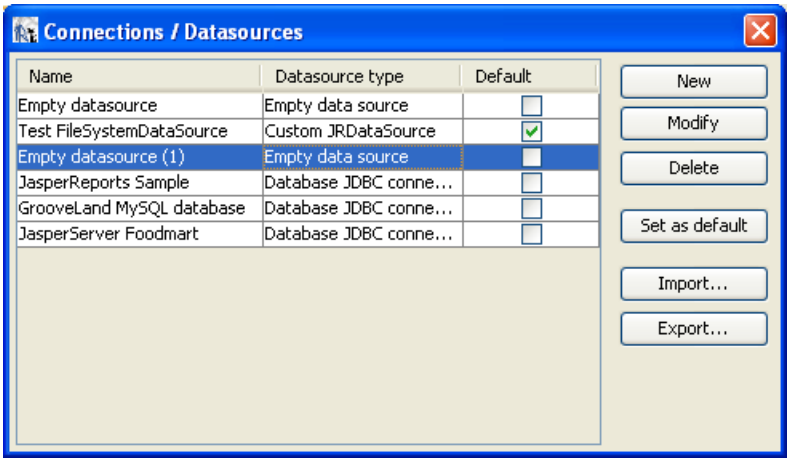



**Figure 10-48 Imported data source (Empty data source is duplicated)**

# 11 CHARTS

JasperReports provides the ability to render charts inside a report by using `JFreeChart`, a powerful open source chart-generation library.

In a chart is possible to print the data coming from the main dataset or using a subdataset (we will deal with subdatasets in**Chapter 12**). This allows you to include many different charts in one document without using subreports.

JasperReports supports a wide variety of chart types: Area, Bar, Bar 3D, Bubble, Line, Pie, Pie 3D, Scatter Plot, Stacked Bar, Stacked Bar 3D, Time Series, XY Area, Stacked Area, XY Bar, XY Line, Meter, Thermometer, Candlestick and High Low Open Close charts. A special chart called MultiAxis can be used to aggregate multiple charts into a single one.

## 11.1 Creating a Simple Chart

In this section, you will learn how to use the **Chart** tool to build a report containing a Pie 3D chart, then you will explore all the details regarding chart management.We will use the JasperReports sample database for this example:

1. Create a new empty document. Open the Report query dialog box by clicking the button representing a cylinder in the designer tool bar.
2. The idea is to produce a chart to display the count of orders in different countries using a simple query:

```
select COUNT(*) as orders, shipcountry from orders group by shipcountry
```

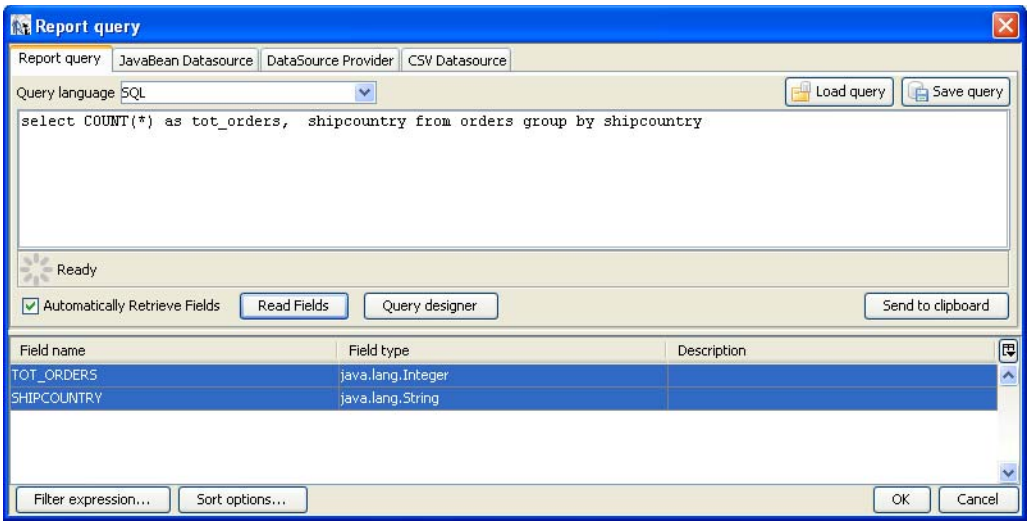



**Figure 11-1 Query Dialog**

Confirm your query by clicking **OK** (see Figure 11-1):

3. iReport will register the query-selected fields. Place the fields in the detail band by dragging them from the outline view (**Figure 11-2**).

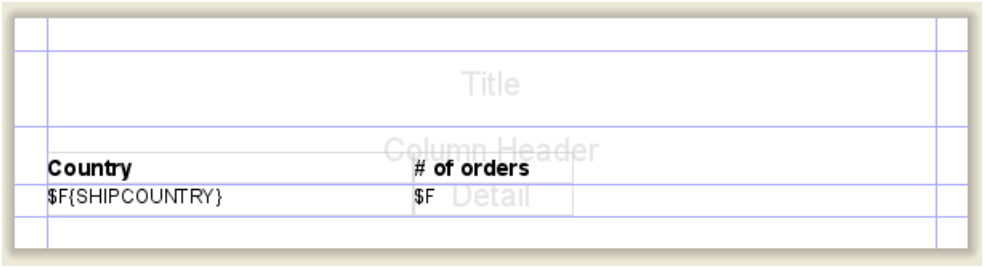



**Figure 11-2 The initial design**

4. Rearrange the bands, and expand the summary; this is where we will place our new chart.

5. Select the **Chart** tool and drag it into the summary band. When you add a new chart element to a report, iReport shows the chart selection window from which you can pick the chart type (**Figure 11-3**).


**Figure 11-3 Chart Selection Window**

6. Select the Pie 3D icon and click **OK**. **Figure 11-4** shows what your report should now look like.

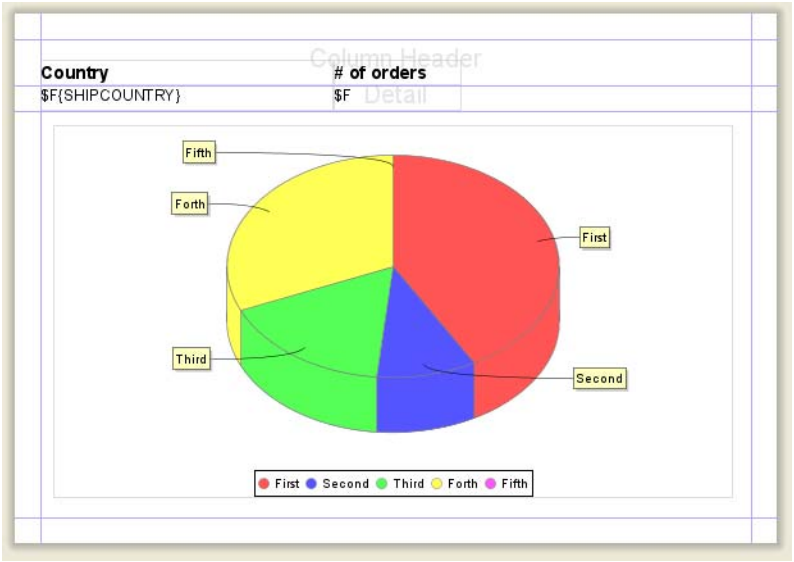



**Figure 11-4 The chart is placed in the summary band**

At this point, we have to configure the chart. Right-click the chart element and select the menu item **Chart Data** (see **Figure 11-5**).

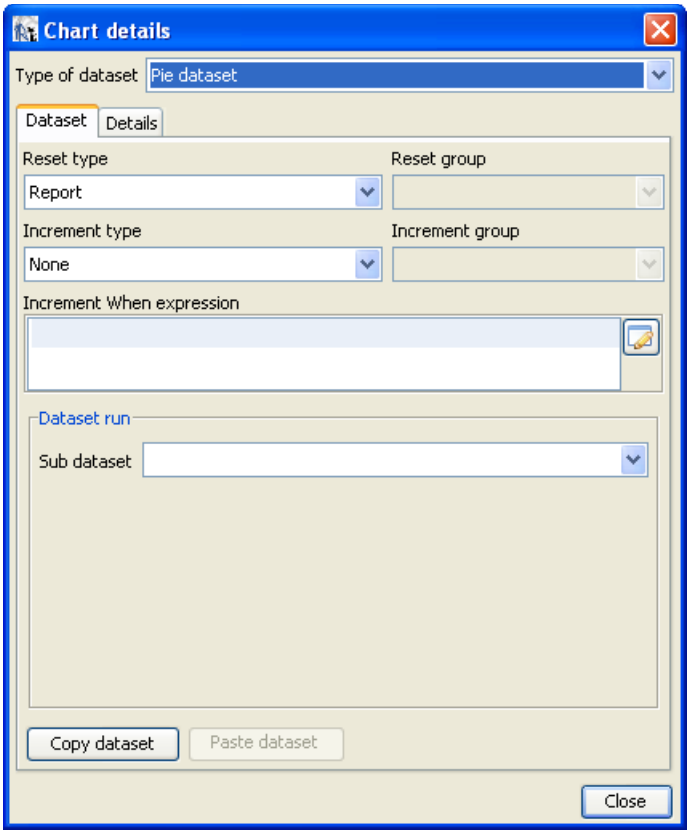



**Figure 11-5 The chart data window**

In this window you can select the data to use in order to create the chart. The **Type of Dataset** combo box allows you to specify the dataset types to generate the graph. Usually one dataset type is available, except when generating an XY Bar chart.

In the **Dataset** tab, you can define the dataset within the context of the report. Specifically, *Reset Type* and *Reset Group* allow you to periodically reset the dataset. This is useful, for example, when summarizing data relative to a special grouping. *Increment Type* and *Increment Group* specify the events that determine when new values must be added to the dataset: by default each record of the dataset used to fill the chart corresponds to a value printed in the chart. This behavior can be changed forcing the engine to collect the data for the chart only at specific time (like for instance every time the end of a group is reached).

The *Increment When expression* area allows you to add a flag to determine whether to add a record to the record set designed to feed the chart. This expression must return a Boolean value. iReport considers a blank string to mean “add all the records.”

For the purposes of this example, set the *Reset Type* to *Report* since you don't want the data to be reset, and leave the *Increment Type* set to *None* so that each record will be appended to your dataset.

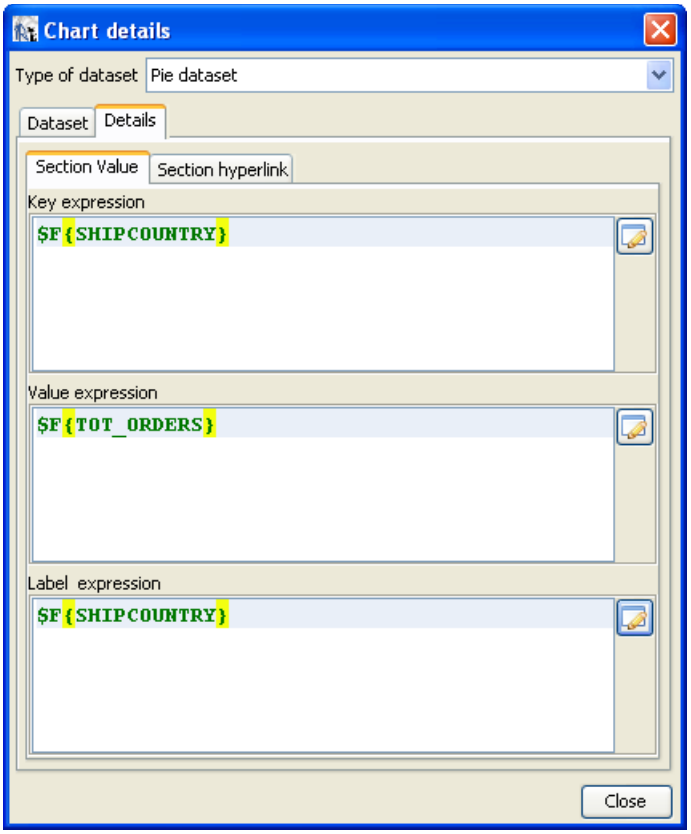



**Figure 11-6 Dataset Configuration**

The Details tab allows you to enter an expression to associate with every single value in the data source. For the Pie 3D chart type, three expressions can be entered: *key*, *value*, and *label* (see **Figure 11-6**).

The *key expression* must be a unique value to identify a slice of the pie chart. If a key value is repeated, the label and value values previously associated with that key are overwritten. A key can never be *null*.

The *value expression* specifies the numeric value associated with the key.

The label expression allows you to specify a label for each slice of the pie chart. This expression is optional, and the default value is the key value.

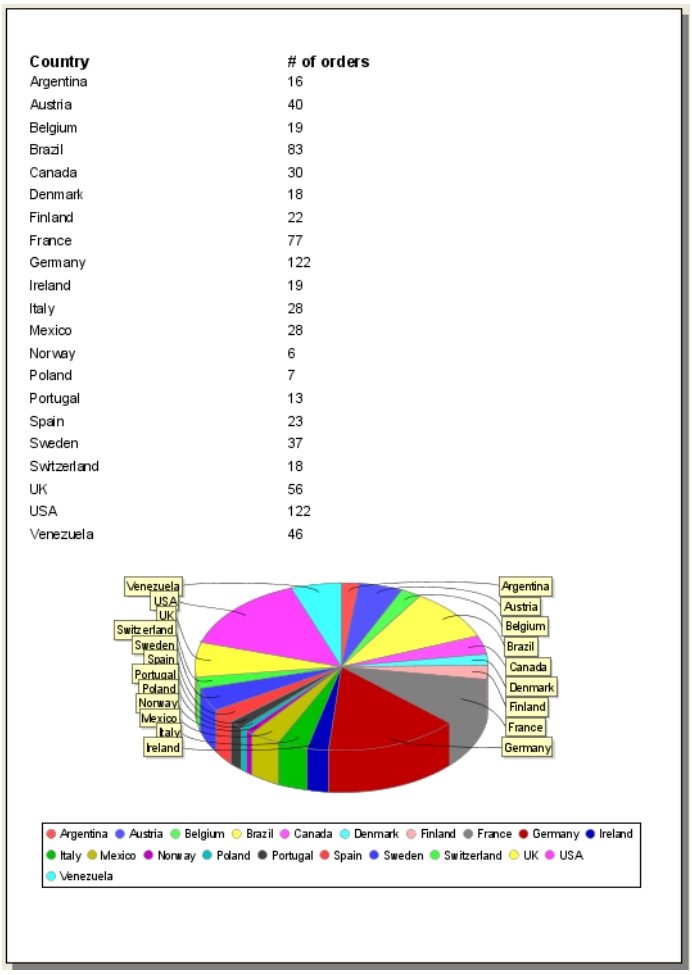



**Figure 11-7 The final report**

Previewing the report, you should get a result similar to the one in **Figure 11-7**.

## 11.2 Using Datasets

The data represented within charts is collected when the report is generated and then stored within the associated dataset. The dataset types are as follows:

- Pie
- Category
- Time period
- Time series
- XY
- XYZ
- High low
- Value

Think of a dataset as a table. Each dataset has different columns (fields). When a new record is inserted into the dataset, values are added to the fields.

Figure 11-7, shown earlier, demonstrates the options that you can select in JasperReports to indicate when and how to acquire data for the dataset. Specifically, you can indicate whether and when the dataset should be emptied (through *Reset Type* and *Reset Group* settings), and when to append a new record to the dataset (through *Increment Type* and *Increment Group* settings). These four fields have the same effect as the corresponding fields used for report variables (see the discussion of variables in Chapter 6).

Depending on the dataset type that you have selected, the **Chart Data** tab shows the fields within the specified dataset.

Detailed descriptions of the various field types and their functionality are available in *The Ultimate Guide to JasperReports* by Teodor Daniciu, Lucian Chirita, Sanda Zaharia, and Ionut Nedelcu.

## 11.3 Value Hyperlinks

Some types of datasets provide a way to assign a *hyperlink* to the value represented in the chart, enhancing the user experience by allowing the user to click the chart to open a web page or navigate through the report.

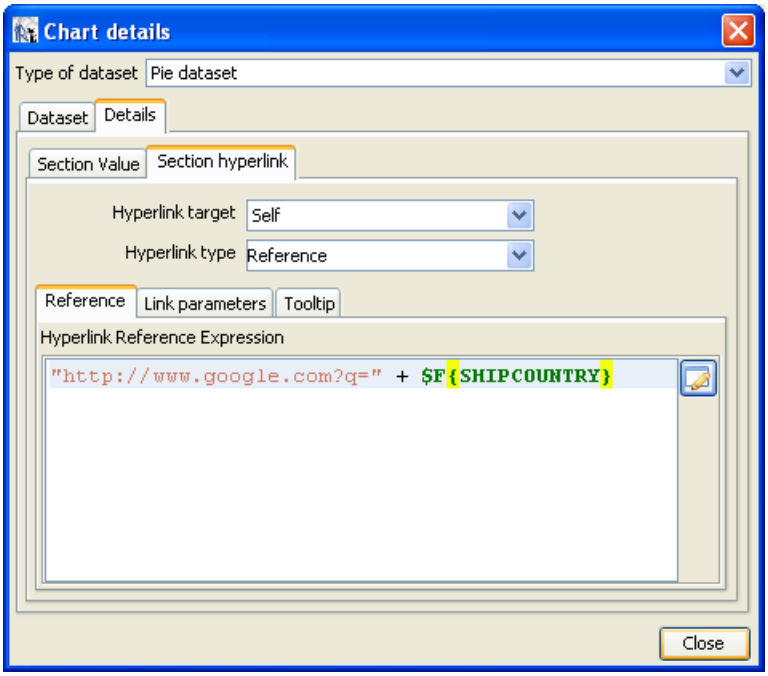



**Figure 11-8 Hyperlink for a slice in a pie dataset**

The click-enabled area depends on the chart type. For pie charts, the hyperlink is linked to each slice of the pie; for the bar charts, the click-enabled areas are the bars themselves (see Figure 11-8).

Adding hyperlinks to elements is described in Chapter 4, "Report Structure," on page 43 chapter. Recall from that discussion that hyperlinks utilize expressions to include all the fields, variables, and parameters of the dataset used by a chart.

## 11.4 Properties of Charts

The appearance of a chart can be configured using the chart element property sheet (see **Figure 11-9**). You can see and edit properties common to all charts and graphs (such as the title and the visibility of the legend) as well as other properties specific to the chart or graph that is being created. Properties that differ among chart types are known as plot properties.

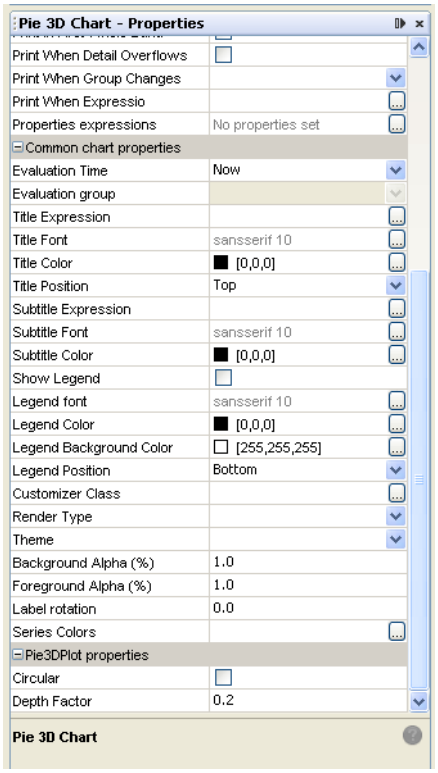



**Figure 11-9 Chart element property sheet**

Currently, JasperReports takes advantage of only a small portion of the capabilities of the `JFreeChart` library. To customize a graph, a class must be written that implements the following interface:

```
net.sf.jasperreports.engine.JRChartCustomizer
```

The only method available from this interface is the following:

```
public void customize(JFreeChart chart, JRChart jasperChart);
```

which takes a `JFreeChart` object and a `JRChart` object as its arguments. The first object is used to actually produce the image, while the second contains all the features you specify during the design phase that are relevant to the customize method.

Another way is by creating a new chart **Theme** that gives you full control over you customize the style of the chart.

## 11.5 Using Chart Themes

Chart themes allow you to customize the design style of a chart. There are several techniques to create a Chart theme, but the most simple one for the end user is to create a `JRCTX` file (JasperReports Chart Theme XML) using iReport. In this section we will see how to create such a file and how to use it in a report.

### 11.5.1 Creating a Chart Theme

To create a Chart Theme XML:

1. Select **New...** → **Chart Theme** from the **File** menu.
2. Specify the location where to store the new file (which has extension `.jrctx`).
3. Press **Finish**.

### 11.5.2 Using the Chart Theme Designer

iReport automatically opens the new chart theme in the Chart Theme designer. This designer comprises three main views:

- Template Inspector — Tree view showing the several sections of the chart that you can customize
- Main view — Displays a real time preview of the new chart theme
- Property sheet— Lists the properties of each section that you can modify

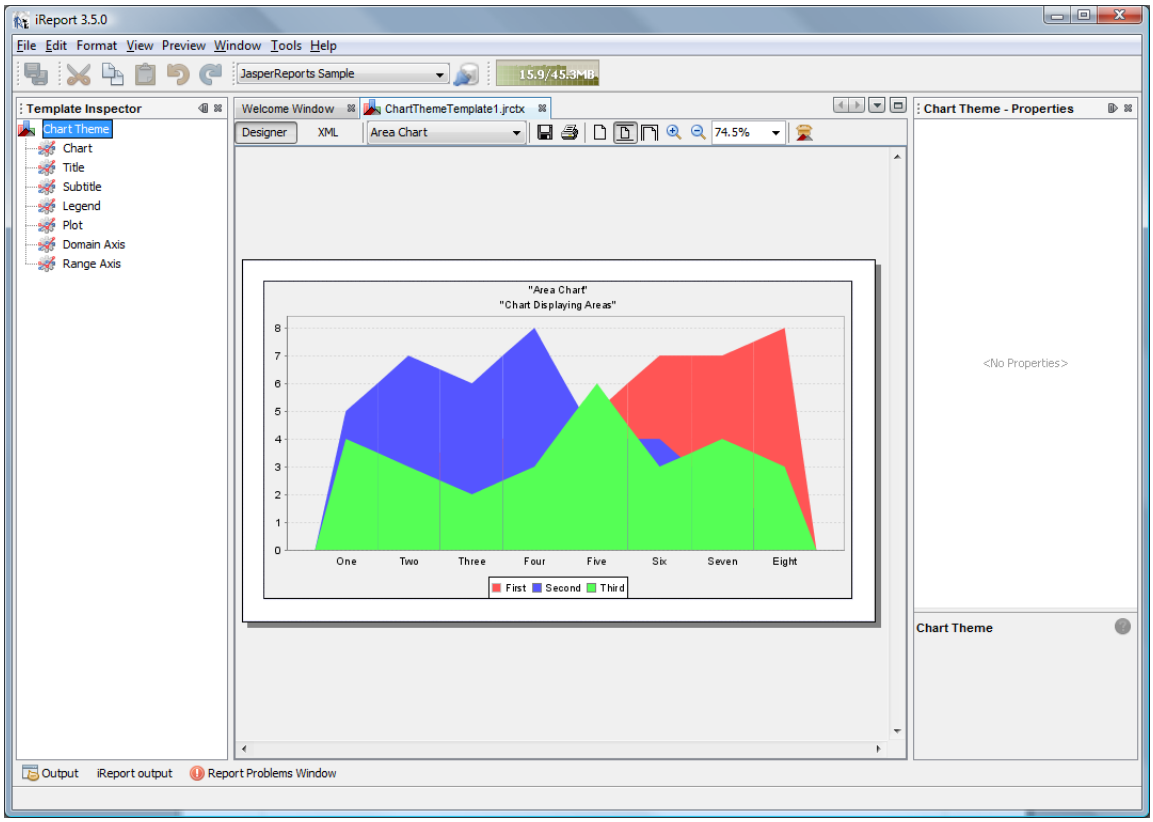



**Figure 11-10 Chart Theme Designer**

JasperReports organizes a chart theme into seven sub-sections:

- Chart
- Title
- Subtitle
- Legend
- Plot
- Domain Axis
- Range Axis

iReport allows you to design the properties for each part of the chart theme. **Figure 11-11** shows which part the chart these sections affect in an Area.

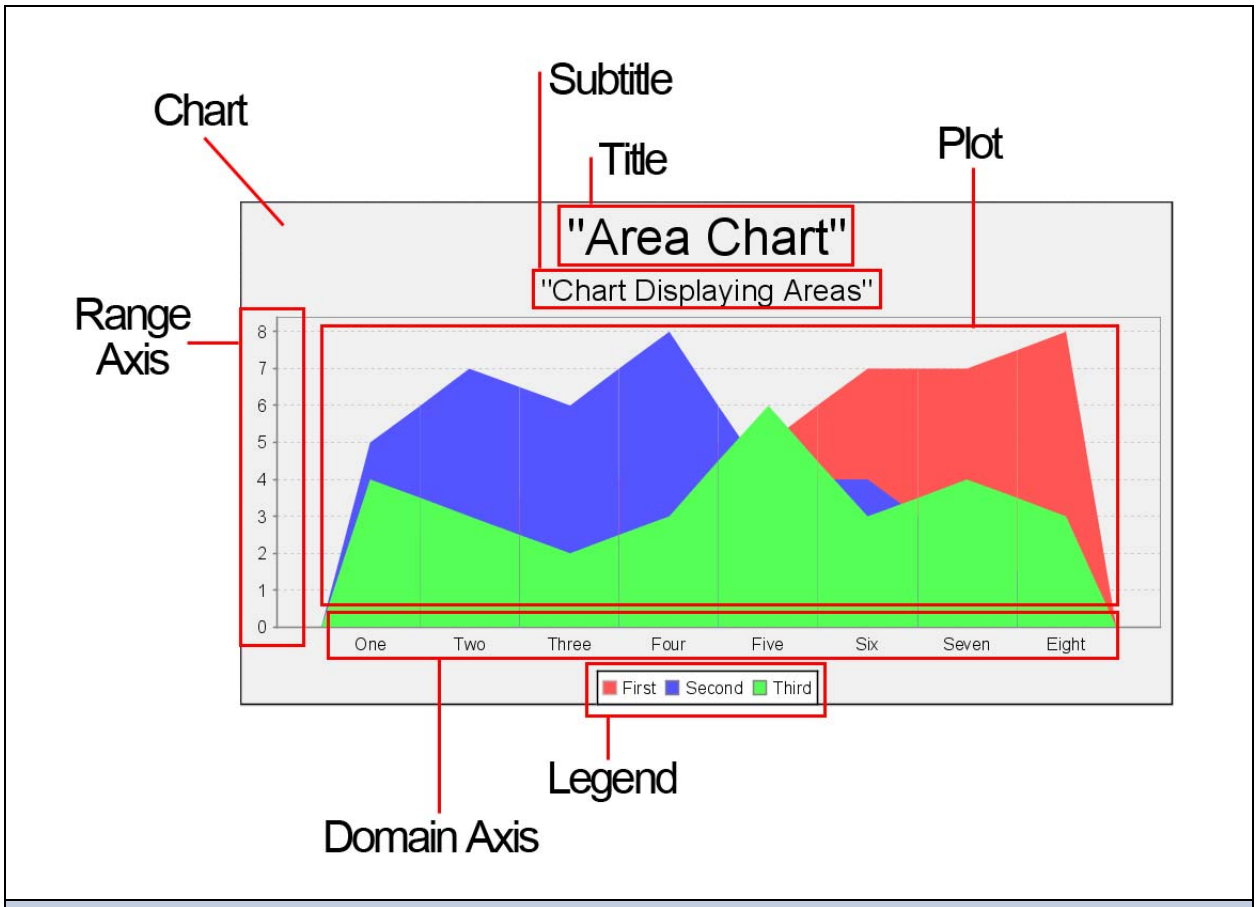



**Figure 11-11 Chart Sections**

By selecting a section node in the tree view, the properties available for that section are displayed in the property sheet. These properties are applied to all the chart types, not just to the chart shown in the preview area. To preview a different type of chart, select it from the combo box in the toolbar of the preview window.

The properties are fairly self-explanatory so I will not describe them any further.

#### 11.5.2.1 Editing Chart Theme XML Source

The Chart Theme designer allows you to view and edit the XML source code for your chart theme via the **XML** tab in the preview window. I suggest, however, that only those users quite experienced with XML architecture should modify a chart theme with direct edits of the source file.

Below is a variation of the default **Area Chart** chart theme, shown as it would appear in the **Designer** tab view; the corresponding source code that would be displayed in the **XML** tab view can be found in **Appendix A, "Chart Theme Example," on page 253**.

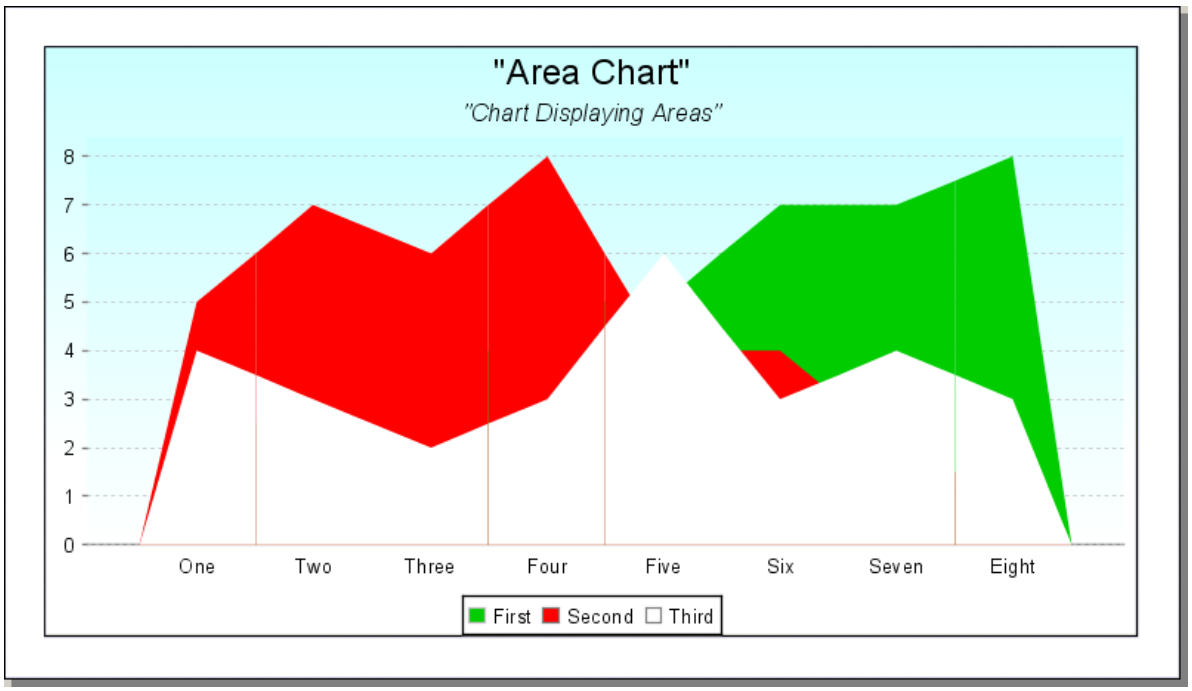



**Figure 11-12 Example Chart Theme - TricolorAreaChart**

### 11.5.3 Creating a JasperReports Extension for a Chart Theme

The JRCTX file we have created cannot yet be used in a report. It just describes some properties of the chart, but it needs to be wrapped into a JasperReports extension jar first: we need to set a name for the new theme, produce a `jasperreports_extension.properties` file to describe the JasperReports extension and package the `.jrctx` and the `.properties` files in a jar.

This can be done in a single step by right clicking the Chart Theme root node in the tree view, and selecting the **Export as a Jar...** menu item or optionally clicking the button showing a little package situated in the preview window toolbar (see figure).


**Figure 11-13 Export As a Jar ... Button**

The export tool window pops up (**Figure 11-14**).

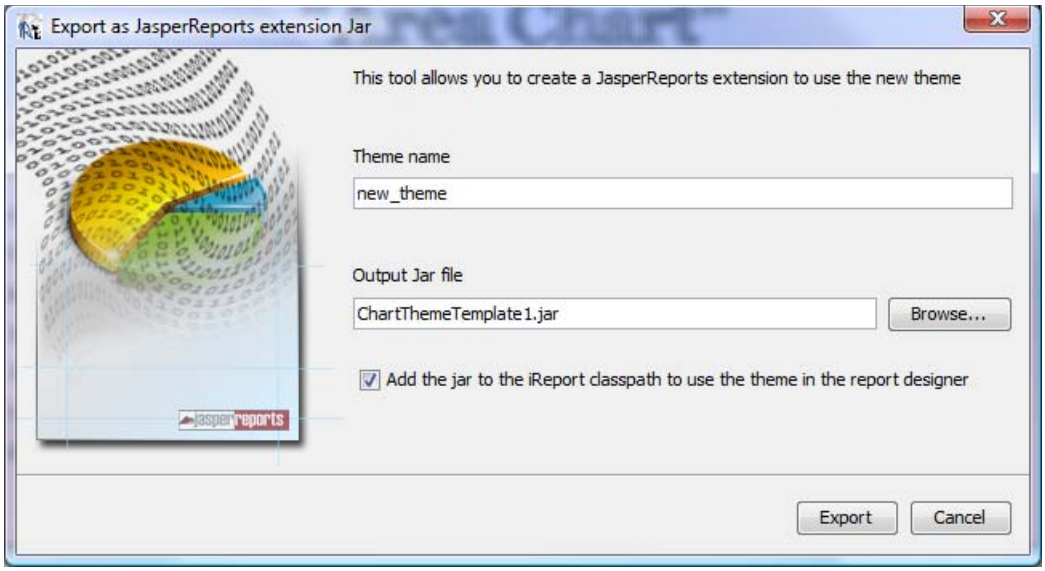



**Figure 11-14 Chart Theme Exporter**

You can set a name for your new theme, this is the name that will be used by JasperReports to identify your theme. Select the jar to create. Optionally, you can automatically add the jar to the iReport classpath, but remember to add that jar to your application too when you deploy the reports.

### 11.5.4 Using a Chart Theme in the Report Designer

Once you have the chart theme extension in the classpath, you should see it in the theme property combo box (the chart properties are shown in the property sheet when a chart element is selected).

Select the new chart theme. In the chart theme as been just added to the classpath, the real time preview could not be available (if you need it, just restart iReport).

Run your report. iReport should display a new report reflecting your customized chart theme, similar to the one shown in **Figure 11-15**.

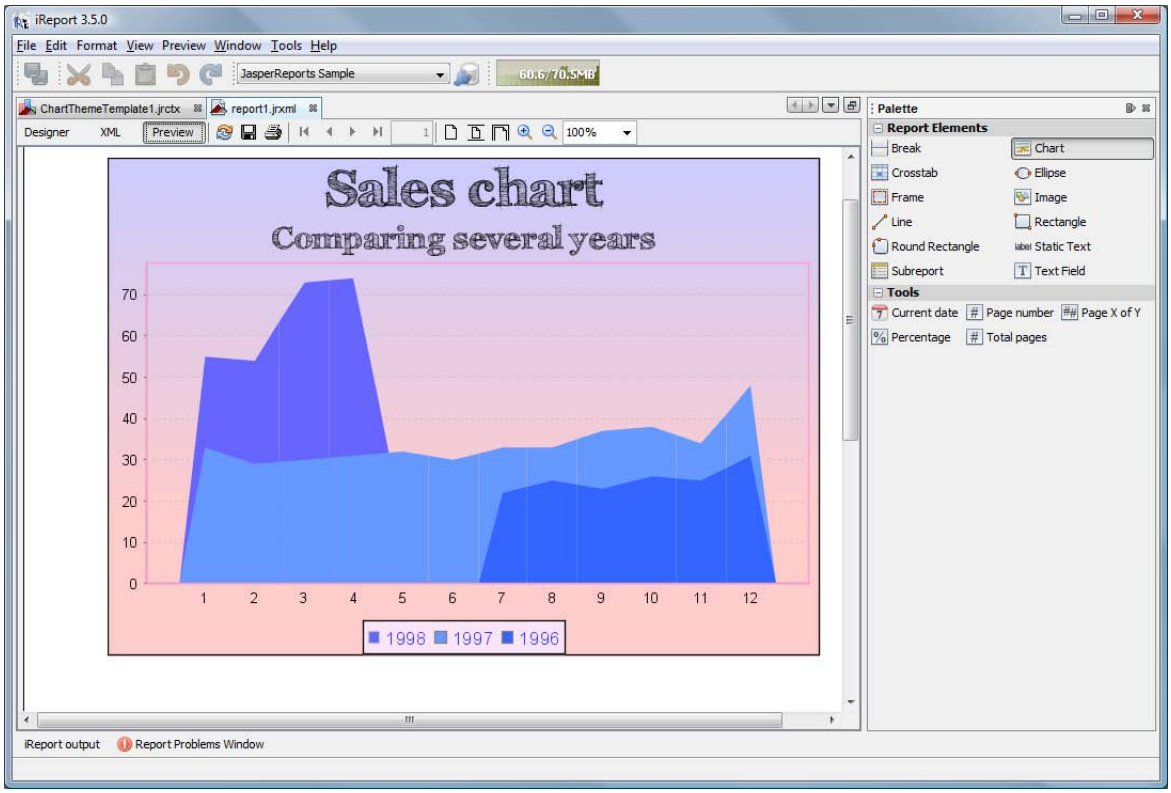



**Figure 11-15 Area chart with a chart theme**

# 12 SUBDATASETS

Report generation is based on a single data source (such as a query, a collection of JavaBeans, or an XML file). When you deal with a chart or a crosstab, this might not be sufficient, or it might simply be easier to retrieve the chart or crosstab data using a specific query or in general using another dataset. You can use a subdataset to provide a secondary record nested within a report (performing an additional query using a new data source or for instance the same database connection used to fill the master report). Currently, you can use a subdataset to fill only a chart or a crosstab, but a developer may be able to use it by creating a custom component. You can have an arbitrary number of subdatasets in a report.

A subdataset has its own fields, variables, and parameters and can have a query executed as needed. The dataset records can be grouped in one or more groups (like in a main report); these groups are used in subdataset variables.

A subdataset is linked to a chart or crosstab by means of a *dataset run*. The dataset run specifies all the information needed by the subdataset to retrieve and filter data and process the rows used to fill the chart or crosstab.

## 12.1 Creating a Subdataset

To create a new subdataset, right-click the root node in the outline view and select **Add Sub dataset** from the context menu (see Figure 12-1).

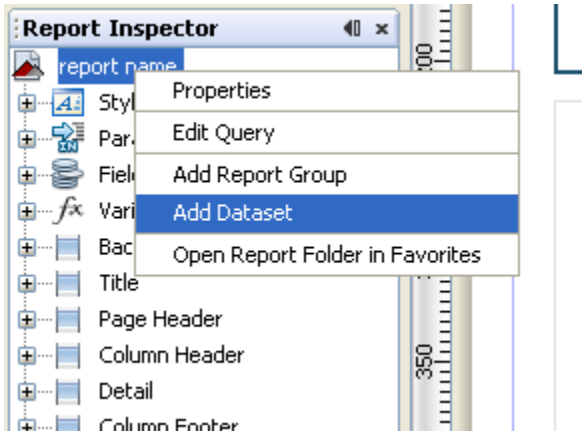



**Figure 12-1 Creating a New Subdataset**

The new subdatset appears in the outline view (see **Figure 12-2**).

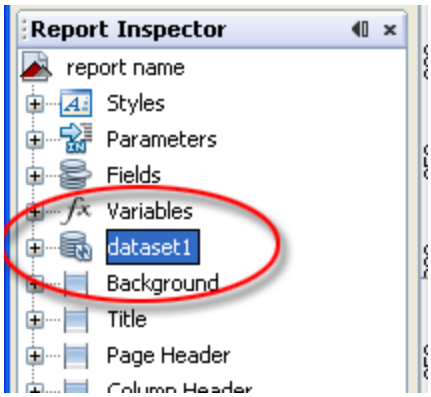



**Figure 12-2 The new subdataset in the outline view**

The property sheet for a node allows you to specify all the subdataset details (see **Figure 12-3**). You can set the desired name for the dataset, which must be unique within your report.

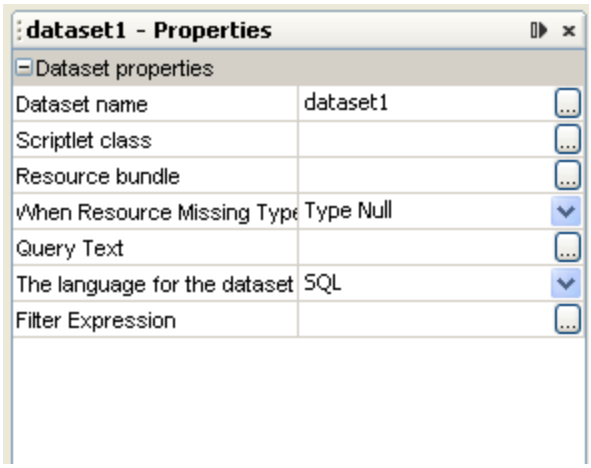



**Figure 12-3 Subdataset properties**

JasperReports permits you to use a scriptlet to perform special calculations on the records of a subdataset records in a manner similar to that provided for the main report. If you need it, you can set the name of your scriptlet class when you create your new subdataset. You can also set the name of a specific resource bundle to be used with this dataset and set the appropriate policy to apply in case of a missing key.

iReport allows you to edit the query, ordering and filter options for the subdataset from the query dialog. To open it, select the subdataset node in the outline view and click **Edit query** (see **Figure 12-4**).

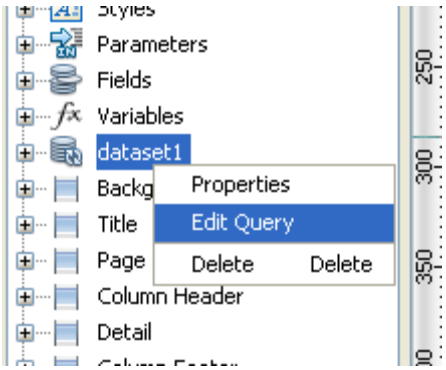



**Figure 12-4 Subdataset Context Menu - Edit Query**

The fields, variables, parameters and groups for a dataset can be managed directly from the outline view (see **Figure 12-5**).

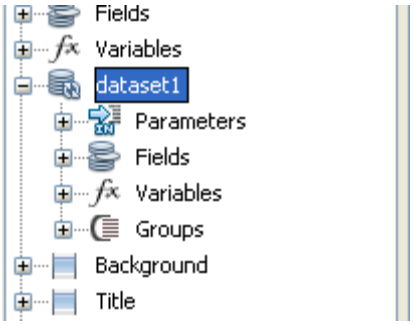



**Figure 12-5 Subdataset Content Tree**

The query dialog can be used to automatically register fields in the subdataset in the same way you do that for the main report (i.e. getting the fields from an SQL query).

In the context of a dataset, groups are only used to group records, but there is no discrete portion of the report tied to them (e.g., like the group header and footer bands associated with groups). Primarily, dataset groups are used in conjunction with variable calculations.

## 12.2 Creating Dataset Runs

As mentioned previously, you can use a subdataset in a chart or in a crosstab. To provide data to the subdataset, JasperReports needs some extra information, such as which JDBC connection to access for the subdataset SQL query, or how to set the value of a specific subdataset parameter. All this information is provided using a dataset run.

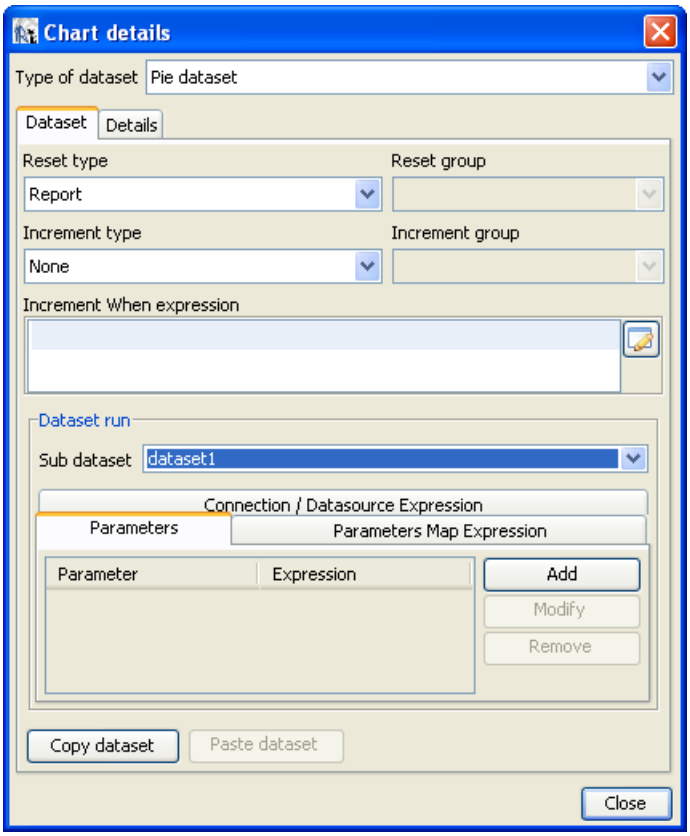



**Figure 12-6 Dataset Run Definition for a Chart**

Figure 12-6 shows a dataset run for a chart. The dataset run definition is similar to what you use to set up a subreport element. A subreport itself can be seen as a sub-dataset, and you need to set a value for its parameters and specify a connection to use in order to get the data: you can set the value of the subdataset parameters using expressions containing main report objects (like fields, variables, and parameters), define a parameters map to set values for the subdataset parameters at runtime, and finally define the connection or data source that will be used by the subdataset.

## 12.3 Working Through an Example Subdataset

The following step-by-step example shows how to use a subdataset to fill a chart.:

1. Create a basic report using a simple SQL query:

```
(select count(*) as tot_orders from orders)
```

The resulting main report will have just a single record containing the total number of orders (see **Figure 12-7**).

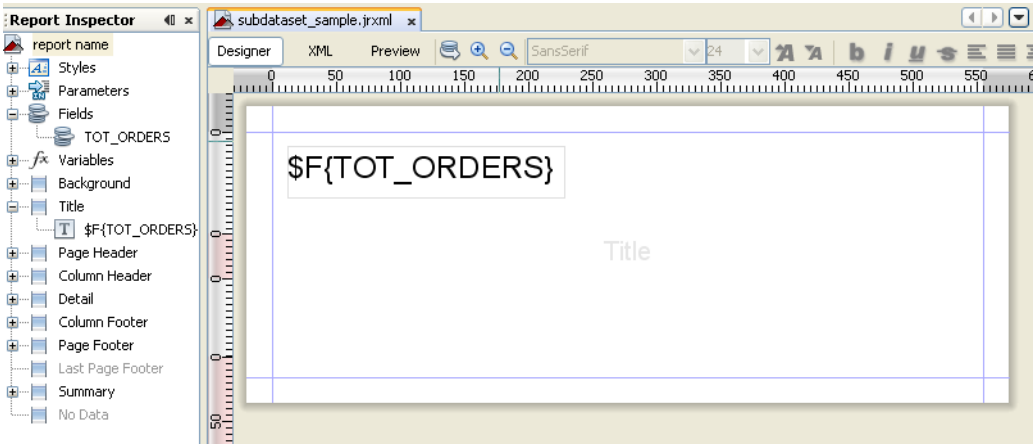



**Figure 12-7 Initial Layout**

2. Create a subdataset as explained above in this chapter. Edit the subdataset query and set it to:

```
select SHIPCOUNTRY, COUNT(*) country_orders from ORDERS group by SHIPCOUNTRY
```

The fields will be registered in the subdataset (see **Figure 12-8**).

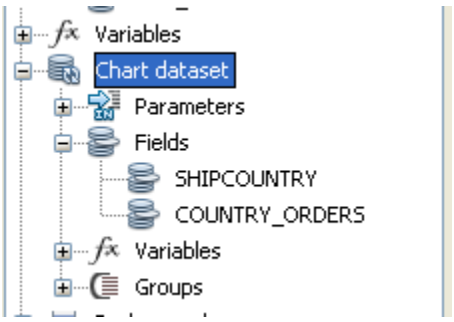



**Figure 12-8 The subdataset to fill the chart**

3. Now create a chart element in the title, for instance a Pie 3D (like in **Figure 12-9**). Right-click in the chart, and select the chart data menu item to open the chart definition dialog.

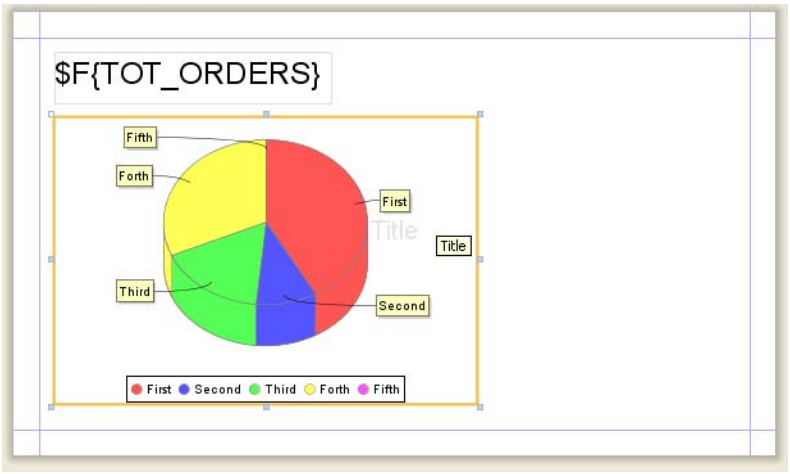



**Figure 12-9 The subdataset will be used to fill the chart**

4. In the dataset run section, select the dataset we have created. Click one **Connection/Datasource Expression** tab and select the **Use connection expression**. In our sample we will share with the subdataset the same database connection used by the report: selecting **Use connection expression** causes the expression to be set automatically to `$P{REPORT_CONNECTION}`. If you wanted to use a different connection type, you can refer to the subreport chapter where is explained in depth how a connection or data source can be specified using an expression.

In order to allow the expression context to update the fields, parameters and values, after the dataset run configuration close the dialog, then reopen it again. When editing the chart dataset details (in the details tab), you should now be able to use the subdataset fields (see **Figure 12-10**).

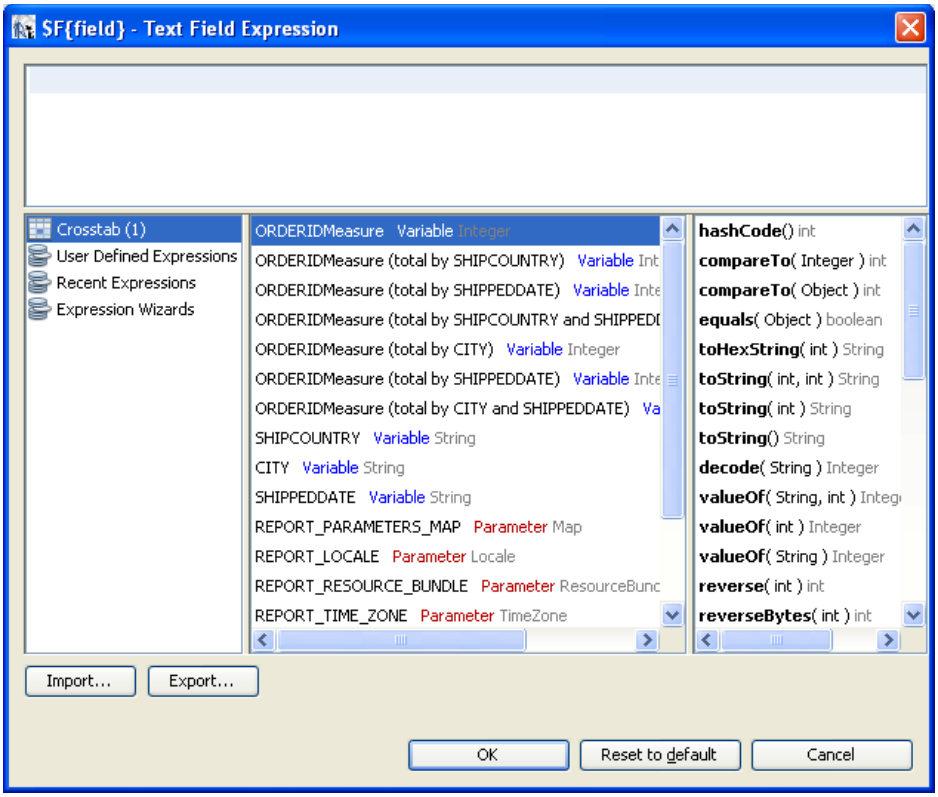



**Figure 12-10 Expression editor showing the subdataset fields**

Set the chart dataset (meaning the values to fill the chart) as shown in **Figure 12-11**.

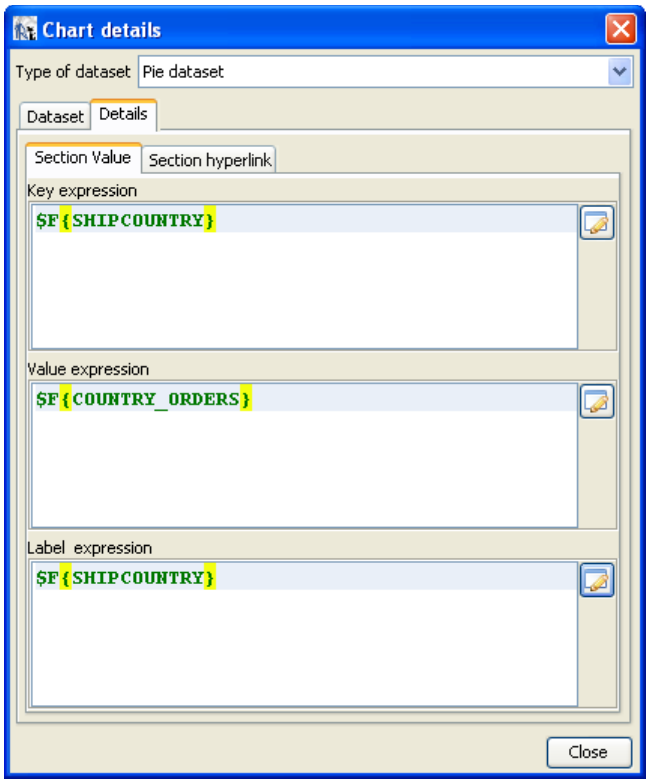



**Figure 12-11 Pie dataset configuration**

Remember that you can not use objects coming from the master report dataset in a chart or a crosstab that uses a sub-dataset. Only sub-dataset objects can be used in that case.

When done, run the report. If everything has been done as explained, you should get a result similar to the one presented in **Figure 12-12**.

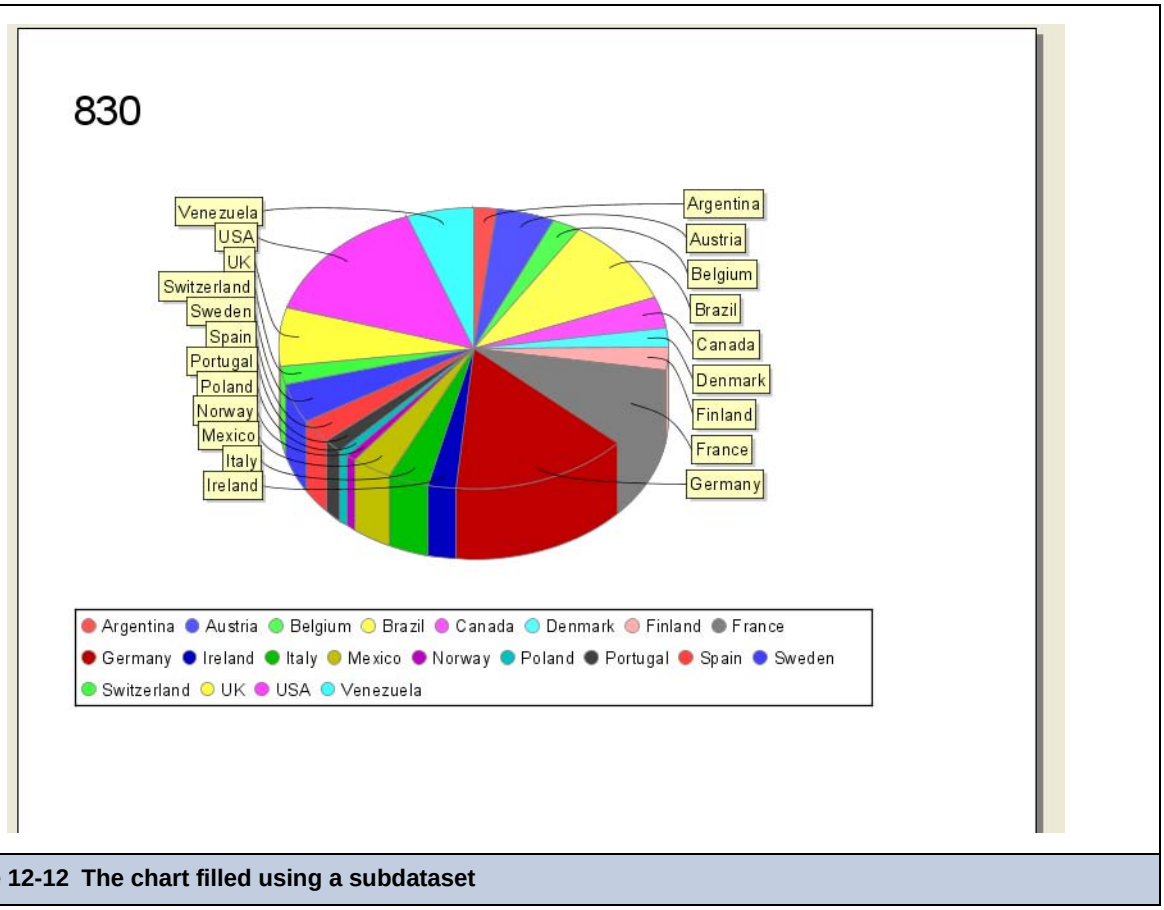



**Figure 12-12 The chart filled using a subdataset**

# 13 CROSSTABS

A crosstab is a kind of table where the exact number of rows and columns remains undefined at design time, like a table that shows the sales of some products (rows) during different years (columns):

| Fruit / Year | 2004 | 2005 | 2006 | ... |
|---|---|---|---|---|
| Strawberry | | | | |
| Wild Cherry | | | | |
| Big Banana | | | | |
| ... | | | | |

The implementation of crosstabs in JasperReports allows the grouping of columns and rows, the calculation of totals and individual format configuration of every cell. For each row or column group, you have a detail row/column and an optional total row/column. Data to fill the crosstab can come from the main report dataset or from a subdataset. Thanks to an intuitive wizard, iReport makes it easy to create and use this powerful reporting component.

## 13.1 Using the Crosstab Wizard

To understand how a crosstab works, I will walk you through creating one using the **Crosstab Wizard**. Ireport displays the wizard automatically when you add a Crosstab element to a report.

Start with a blank report containing this query:

```
select * from orders
```

You will include the crosstab at the end of the report, in the summary band. To do so:

1. Drag the **Crosstab** tool into the summary band. The first screen of the **Crosstab Wizard** appears.

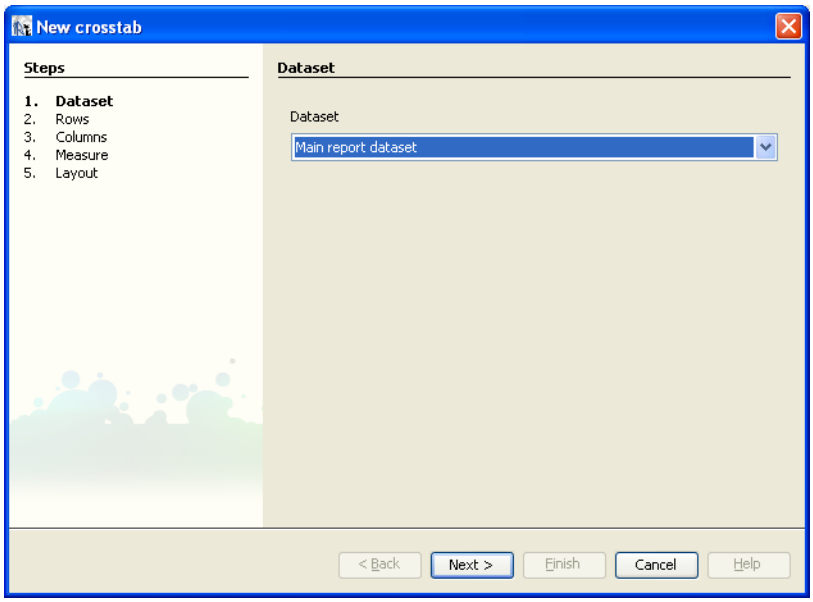



**Figure 13-1 The first step of the Crosstab Wizard**

Choose the dataset to fill the crosstab. Specify the dataset of the main report (as shown in Figure 13-1). Click **Next** to go to the next step.

2. In the second screen, you have to define at least one row group. For the purposes of this example, you will choose to group all records by `SHIPCOUNTRY`, as shown in **Figure 13-2**.

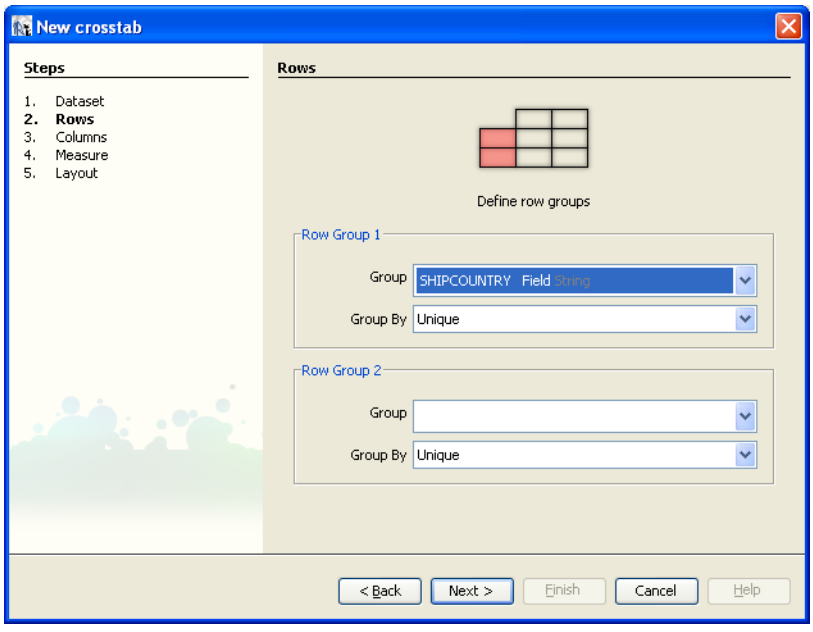



**Figure 13-2 Second step: row groups definition**

This means that each row in the crosstab will refer to a specific country. Unlike what happens in the main report, JasperReports will sort the data for you, although you can disable this function to speed up the fill process if your data is already sorted. Using the **Crosstab Wizard**, you can define one or two row groups. This is only a limitation of the wizard;

you can define as many row and column groups as you need in the outline view (I'll talk more about that later in the **Working with Columns, Rows, and Measures** on **page 220**.) Click **Next** to move to the third step.

3. As with the rows, from the third screen you can define up to two column groups. For this example, you will use only one column group, grouping the data by the `SHIPPEDDATE` field, as shown in**Figure 13-3**. Specifically, you will use a function that returns only the year of the date, thus grouping the orders by year.

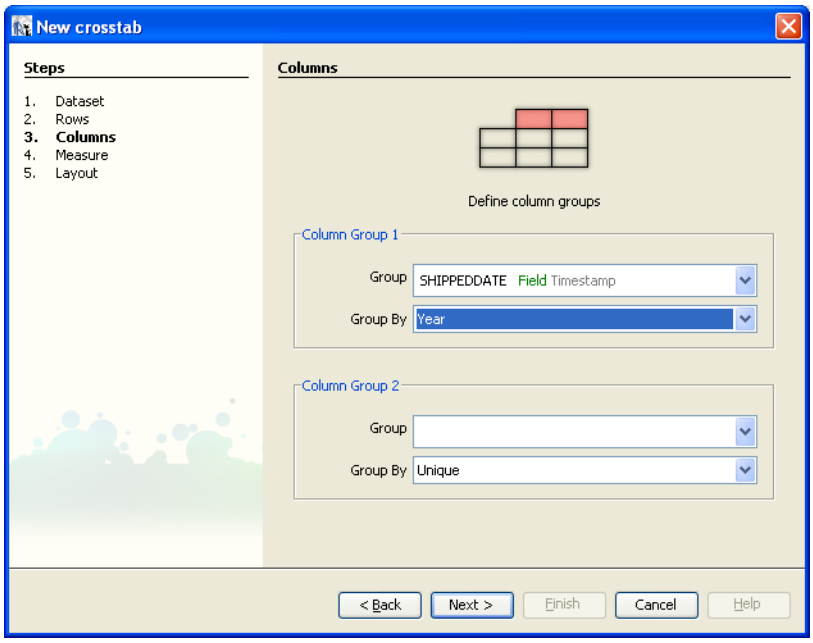



**Figure 13-3 Definition of the column groups**

As you can see in **Figure 13-3**, whenever you have a time field (time stamp, date, etc.), you can use a time-based aggregation function (such as `Year`, `Month`, `Week`, or `Day`), or treat it as a plain value (in which case you can use the `Unique` aggregation function to group records having the same value). Click **Next** to move to the next step.

4. It's time to define the detail data. Normally, the detail (or measure) is the result of an aggregation function like the count of orders by country by year, or the sum of freight for the same combination (country/ year). You will choose to print the

number of orders placed by specifying `ORDERID` (field) in the **Measure** combo box and **Count** in the Function combo box (see Figure 13-4). Once again, click **Next** to continue.

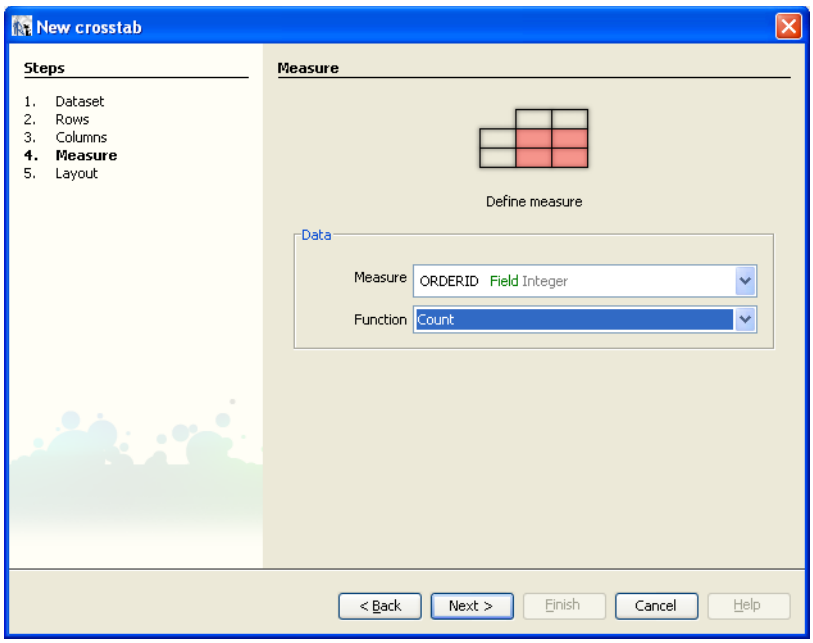



**Figure 13-4 Definition of the measure**

5. In the last step, you can set a few options for the crosstab layout. You can indicate whether you want to see grid lines or not, choose a color set to easily distinguish totals, headers and detail cells, and whether you want to include the totals for rows and columns.

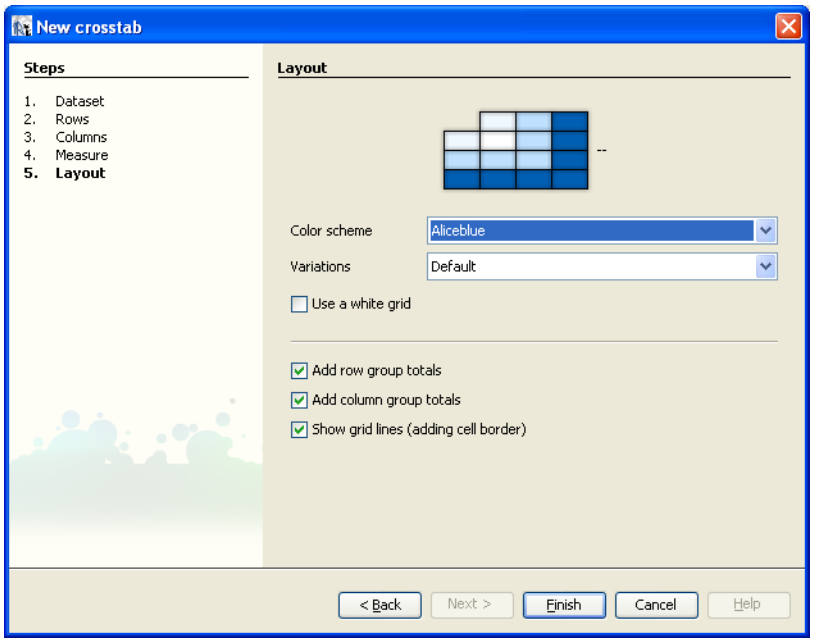



**Figure 13-5 Some crosstab options**

For this example, select all the check box options, as shown in Figure 13-5, and click **Finish**.

Note that when you add a crosstab to the report, iReport includes a corresponding active tab in the design window. This tab is the crosstab designer for the new **Crosstab** element.

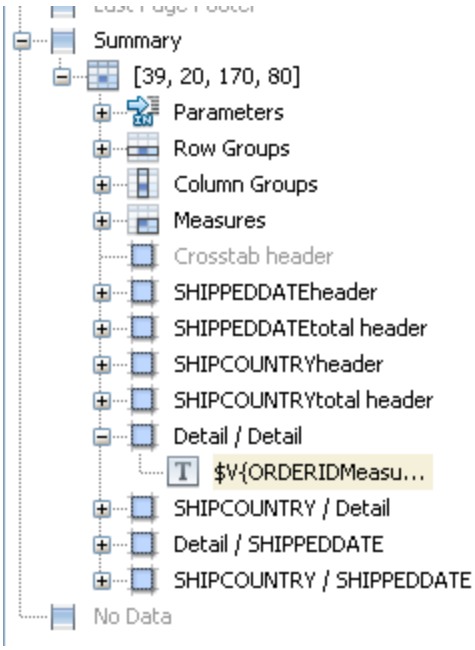



**Figure 13-6 Outline Tree View - Crosstab Details**

The crosstab element in the outline view shows the whole crosstab structure, included the crosstab parameters, row and column groups, measures and each single cell (see **Figure 13-6**).

| | 1996 | 1997 | 1998 | null | Total SHIPPED |
|---|---|---|---|---|---|
| Argentina | 0 | 6 | 8 | 2 | 16 |
| Austria | 7 | 20 | 11 | 2 | 40 |
| Belgium | 2 | 7 | 10 | 0 | 19 |
| Brazil | 13 | 39 | 29 | 2 | 83 |
| Canada | 4 | 17 | 8 | 1 | 30 |
| Denmark | 2 | 11 | 4 | 1 | 18 |
| Finland | 4 | 13 | 5 | 0 | 22 |
| France | 15 | 38 | 22 | 2 | 77 |
| Germany | 23 | 60 | 37 | 2 | 122 |
| Ireland | 4 | 11 | 4 | 0 | 19 |
| Italy | 3 | 14 | 10 | 1 | 28 |
| Mexico | 9 | 12 | 6 | 1 | 28 |
| Norway | 1 | 2 | 3 | 0 | 6 |
| Poland | 1 | 2 | 4 | 0 | 7 |
| Portugal | 3 | 8 | 2 | 0 | 13 |
| Spain | 6 | 5 | 12 | 0 | 23 |
| Sweden | 6 | 17 | 14 | 0 | 37 |
| Switzerland | 3 | 8 | 6 | 1 | 18 |
| UK | 10 | 26 | 20 | 0 | 56 |
| USA | 20 | 62 | 37 | 3 | 122 |
| Venezuela | 7 | 20 | 16 | 3 | 46 |
| Total | 143 | 398 | 268 | 21 | 830 |

**Figure 13-7 Our first crosstab**

When you execute the new report you should get a result similar to the one shown in **Figure 13-7**.

The last column contains the total for each row, across all columns. The last row contains the total for each column, across all rows. Finally, the last cell (in the corner on the bottom right) contains the combined total for all orders (830).

## 13.2 Working with Columns, Rows, and Measures

A crosstab must have at least one row group and one column group. Each row or column group can have an optional total row/column. The rows and columns are defined by these groups. The following is a basic crosstab with one column group and one row group:

| Crosstab header cell | Column group 1 header | Column group 1 total header |
|---|---|---|
| Row group 1 header | Detail | Col group 1 total |
| Row group 1 total header | Row group 1 total | Row group 1 total/ Column group 1 total |

When you include another row group, the crosstab appears as follows:

| Crosstab header cell | Column group 1 header | Column group 1 total header | |
|---|---|---|---|
| Row group 1 header | Row group 2 header<br>Detail<br>Col group 1 total<br><br>Row group 2 total header<br>Row group 2 total<br>Row Grp 2 total/ Col Grp 1 total<br><br>Row group 1 total header | Row Group 1 total | Row Group 1 total/ Col Group 1 total |

When you add a row group, iReport splits the current row into two rows, adding two subtotals, and creates a new nested row group header is. Adding a column group results in a similar design with two new columns and a column group header.

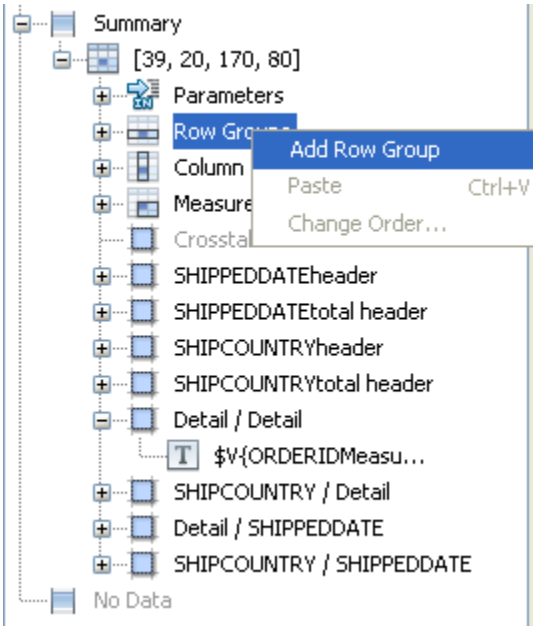



**Figure 13-8 Adding a Row Group**

Row and column groups are displayed in the outline view. To add a row group, right-click the rows node and select **Add Row Group** (see **Figure 13-8**).

The new group appears in the outline view and the relative cells are created in the crosstab designer. You need to set a Bucket Expression—that's an expression used to group the rows. For example, we can add a group row to show the cities of each country. In that case a valid expression could be the field `SHIPCITY` (the expression would look like `$F{SHIPCITY}`). The expression must be set in the row group properties.

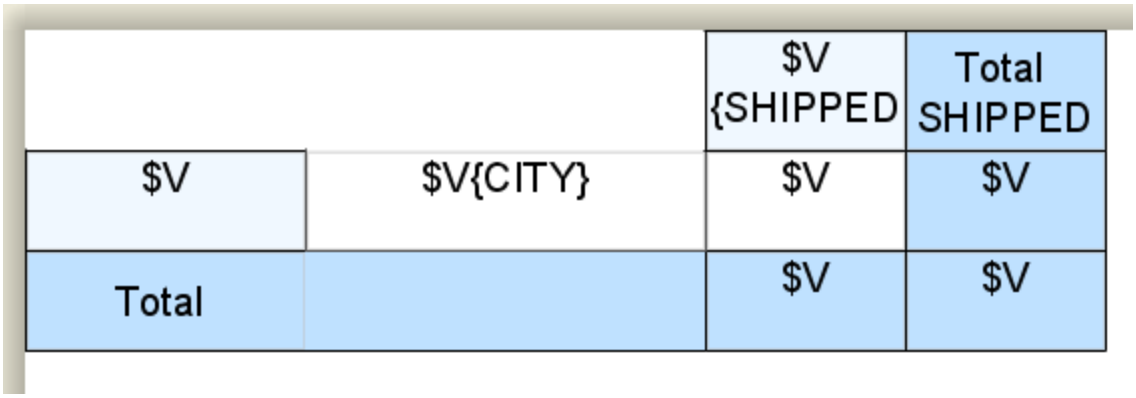



**Figure 13-9 The layout after the new row**

The expression is the only information that must be set after the group creation. Other settings include:

| | |
|---|---|
| *Total position* | Defines the presence of a row to show subtotals |
| *Order* | Order of the values in the group (Ascending or Descending) |
| *Comparator expression* | Returns an instance of `java.util.Comparator` that must be used to order the values |

Column and row sizes can be modified directly using the designer by dragging the cells sides. The content of each cell must be completely contained in the cell (more or less like it happens with the bands in the master report).

When you add a row or column to a crosstab, iReport creates a special variable that refers to the value of the bucket expression. It has the same name as the new group. All the objects that can be displayed in a crosstab cell are shown in the expression editor that will pop-up when editing a textfield expression of elements put in a cell (see **Figure 13-10**).

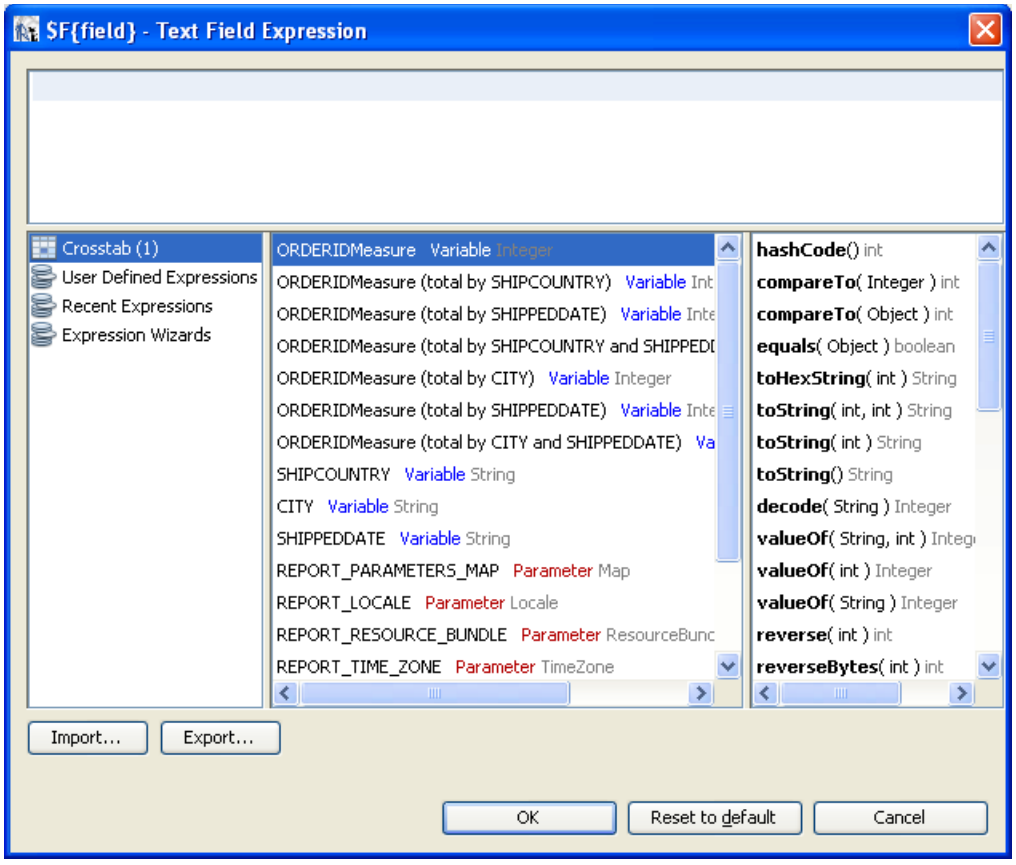



**Figure 13-10 Objects that can be used in a crosstab textfield expression**

When you create a new group, iReport simply creates the new header cell for the group in which it puts a new textfield to display the group value (using a built-in variable having the same name of the group) and fills the new measure cell with a textfield to display the first measure available in the crosstab.

No extra cell display options are applied; in particular, iReport does not set borders for the new cells.

If the `Total Position` is set to a value different that `None` (usually `End`), iReport inserts other cells to host the subtotals. Those cell are created empty, so again you must drag a textfield element into each cell and set a proper expression for the data to display (see **Figure 13-11**).

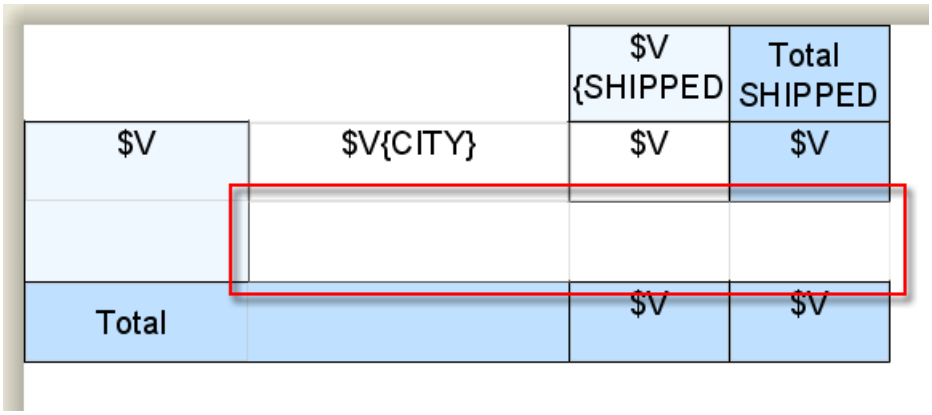



**Figure 13-11 Empty row total cells**

The order of the groups can be changed just be dragging them in the outline view. Please note that the crosstab layout is strictly tied to the group order settings.

## 13.3 Modifying Cells

Each intersection between a row and a column defines a cell. You have header cells, total cells, a detail cell, and an optional when-no-data cell. Cells can contain a set of elements like text fields, static texts, rectangles, images, but they can't contain a subreport, a chart, or another crosstab. **Figure 13-11** shows a crosstab with some colored cells and several text fields.

You can modify the background color and borders of each single cell: right-click the cell you want to change to display the context menu and choose **Padding And Borders** to modify cell borders (see **Figure 13-12**).

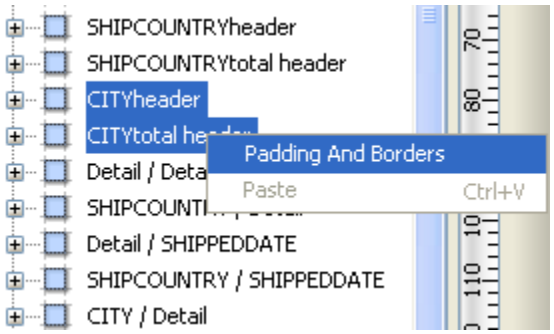



**Figure 13-12 Modifying Cell Borders**

The cell background and style can be modified in the property sheet when you select a cell node in the outline view (see **Figure 13-13**).

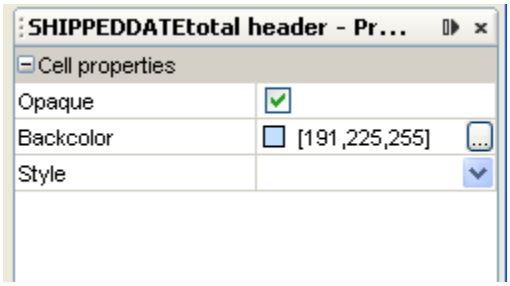



**Figure 13-13 Cell Background and Style**

## 13.4 Understanding Measures

Expressions for elements in a crosstab, such as print-when expressions and text field expressions, can only contain measures. In this context, you cannot use fields, variables, or parameters directly: you always have to use a measure.

A measure is an object similar to a variable. To create a measure, right-click the measures node in the outline view and select **Add Measure** (see **Figure 13-14**).

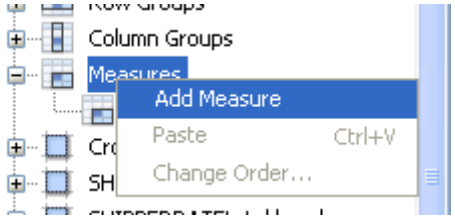



**Figure 13-14 Adding a Measure**

iReport adds the new measure to the outline view. Just as when you create a new group, you'll need to define an expression for the measure, and the easiest way to display a new measure is to use a textfield. Drag a textfield element into a cell and set the proper textfield expression (e.g., with a measure name like `$V{Average_freight}`) and the proper expression class for the textfield that must be consistent with the measure type.

There are several options you can use to set a measure. Besides the name, its class and expression, you can set the calculation type (remember that a measure is always, in some way, the result of a calculation performed on a value for each row and column group interested by the cell in which the measure is displayed). If the available calculation types are not enough, you

can provide your custom `Incrementer` class by means of a Factory that returns an instance of that class (the factory must implement the interface `net.sf.jasperreports.engine.fil.JRIncrementerFactory`).

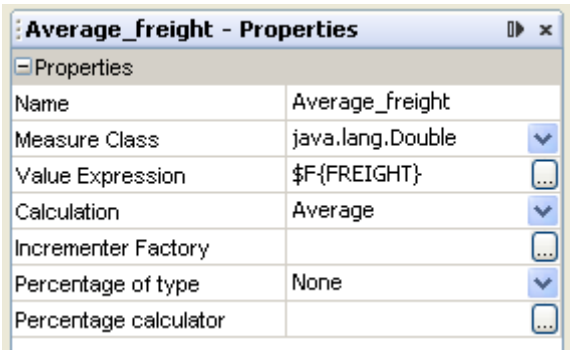



**Figure 13-15 Measure Properties**

If you want to display your measure as a percentage of the Grand Total, you can set the property **Percentage of type** to **Grand Total**. Finally, you can specify a custom calculator class to perform the percentage calculation (the class must use the interface `net.sf.jasperreports.crosstabs.fill.JRPercentageCalculator`).

## 13.5 Modifying Crosstab Element Properties

Select the crosstab node in the outline view to see the crosstab properties in the property sheet (see **Figure 13-16**).

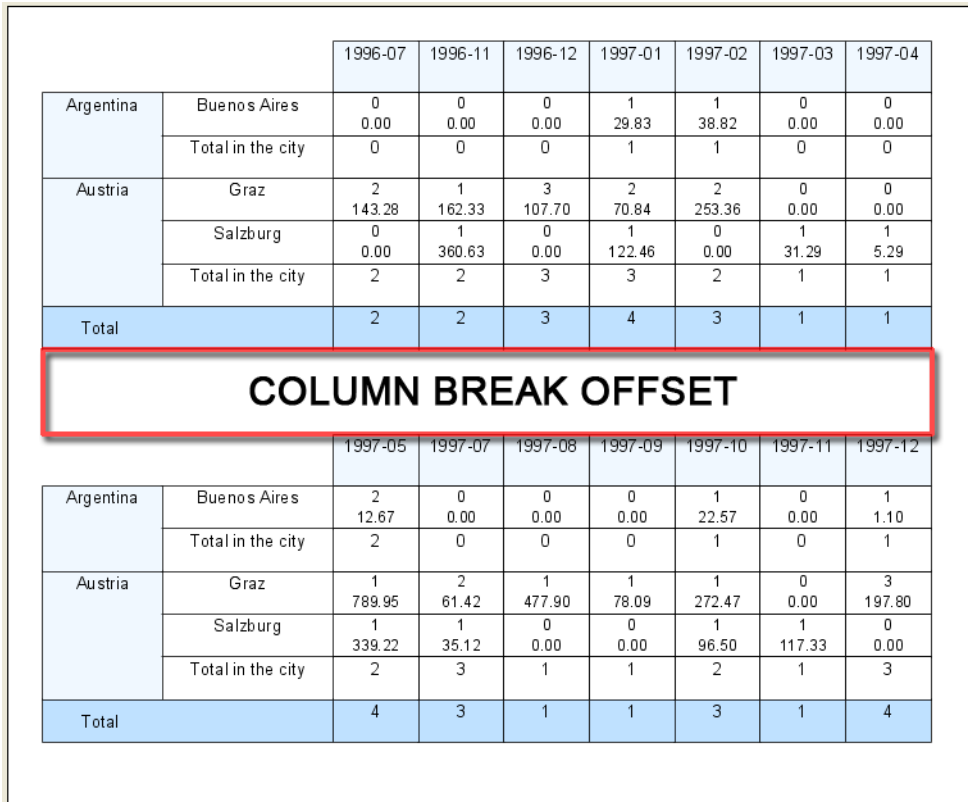

**Figure 13-16 Crosstab Properties**

Following is a brief rundown of some of the options in this dialog box:

*Repeat Column Headers* — If selected, the column headers will be printed again when the crosstab spans additional pages.

*Repeat Row Headers* — If selected, the row headers will be printed again when the crosstab spans additional pages.

*Column Break Offset* — This specifies the space between two pieces of a crosstab when the crosstab exceeds the page width (see **Figure 13-17**).



| | | 1996-07 | 1996-11 | 1996-12 | 1997-01 | 1997-02 | 1997-03 | 1997-04 |
|---|---|---|---|---|---|---|---|---|
| Argentina | Buenos Aires | 0<br>0.00 | 0<br>0.00 | 0<br>0.00 | 1<br>29.83 | 1<br>38.82 | 0<br>0.00 | 0<br>0.00 |
| | Total in the city | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| Austria | Graz | 2<br>143.28 | 1<br>162.33 | 3<br>107.70 | 2<br>70.84 | 2<br>253.36 | 0<br>0.00 | 0<br>0.00 |
| | Salzburg | 0<br>0.00 | 1<br>360.63 | 0<br>0.00 | 1<br>122.46 | 0<br>0.00 | 1<br>31.29 | 1<br>5.29 |
| | Total in the city | 2 | 2 | 3 | 3 | 2 | 1 | 1 |
| Total | | 2 | 2 | 3 | 4 | 3 | 1 | 1 |

COLUMN BREAK OFFSET

| | | 1997-05 | 1997-07 | 1997-08 | 1997-09 | 1997-10 | 1997-11 | 1997-12 |
|---|---|---|---|---|---|---|---|---|
| Argentina | Buenos Aires | 2<br>12.67 | 0<br>0.00 | 0<br>0.00 | 0<br>0.00 | 1<br>22.57 | 0<br>0.00 | 1<br>1.10 |
| | Total in the city | 2 | 0 | 0 | 0 | 1 | 0 | 1 |
| Austria | Graz | 1<br>789.95 | 2<br>61.42 | 1<br>477.90 | 1<br>78.09 | 1<br>272.47 | 0<br>0.00 | 3<br>197.80 |
| | Salzburg | 1<br>339.22 | 1<br>35.12 | 0<br>0.00 | 0<br>0.00 | 1<br>96.50 | 1<br>117.33 | 0<br>0.00 |
| | Total in the city | 2 | 3 | 1 | 1 | 2 | 1 | 3 |
| Total | | 4 | 3 | 1 | 1 | 3 | 1 | 4 |

**Figure 13-17 Column Break Offset**

## 13.6 Crosstab Parameters

Crosstab parameters may be used in the expressions of elements displayed in the crosstab. They can be defined and managed through the outline view (see **Figure 13-18**).

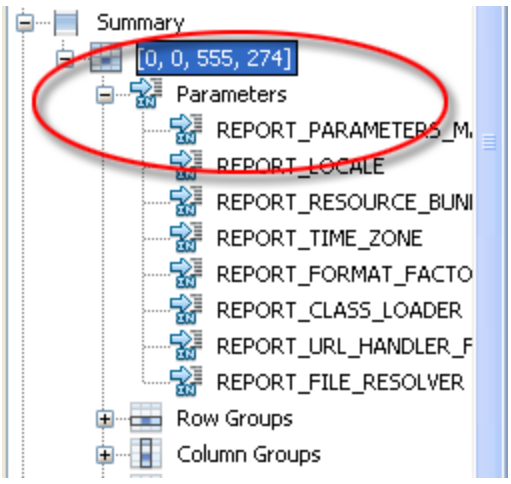



**Figure 13-18 Crosstab Parameters**

To add a parameter, right-click the Parameters node in the Crosstab element and select **Add Crosstab Parameter**. The parameter expression for a crosstab parameter can use only objects from main report, not from an optional sub dataset used to feed the crosstab. Again, these parameters are designed to be used in the crosstab elements, and they are not the same as the

dataset parameters that are used in the crosstab context to filter a query and to calculate the value for measures and group when included in their expressions.

You can use a map to set the value of declared crosstab parameters at run-time. In this case, you'll need to provide a valid parameters map expression in the crosstab properties.

## 13.7 Working with Crosstab Data

As mentioned previously, you can fill a crosstab using data from the main report or from a subdataset. In the latter case, you must specify the dataset run in the Crosstab data accessible right-clicking the crosstab node (or element in the designer) and selecting the **Crosstab Data** menu item (see **Figure 13-19**).

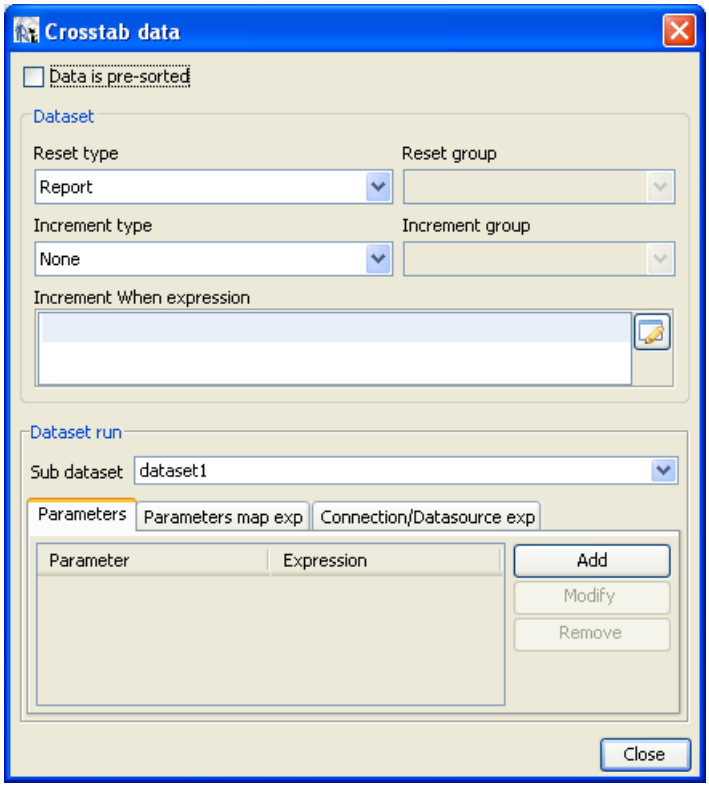



**Figure 13-19 Crosstab Data**

This interface is pretty similar to the one used to provide data to a chart. Effectively, the mechanism to provide the data to a crosstab is very similar.

If your data is presorted, you can select the **Data is Presorted** check box option to speed up the filling process.

You can use the Reset Type/Reset Group and Increment Type/Increment Group options to define when to reset the collected data or when to add a record to your dataset.

The **Increment When** expression is a flag to determine whether to add a record to the record set designed to feed the chart. This expression must return a Boolean value. iReport considers a blank string to mean "add all the records."

See **Chapter 12** for details on how to set the dataset run properties.

## 13.8 Using Crosstab Total Variables

Crosstab total variables (see **Figure 13-20**) are built-in objects that you can use inside crosstab text field expressions to combine data at different aggregation levels (e.g., to calculate a percentage).

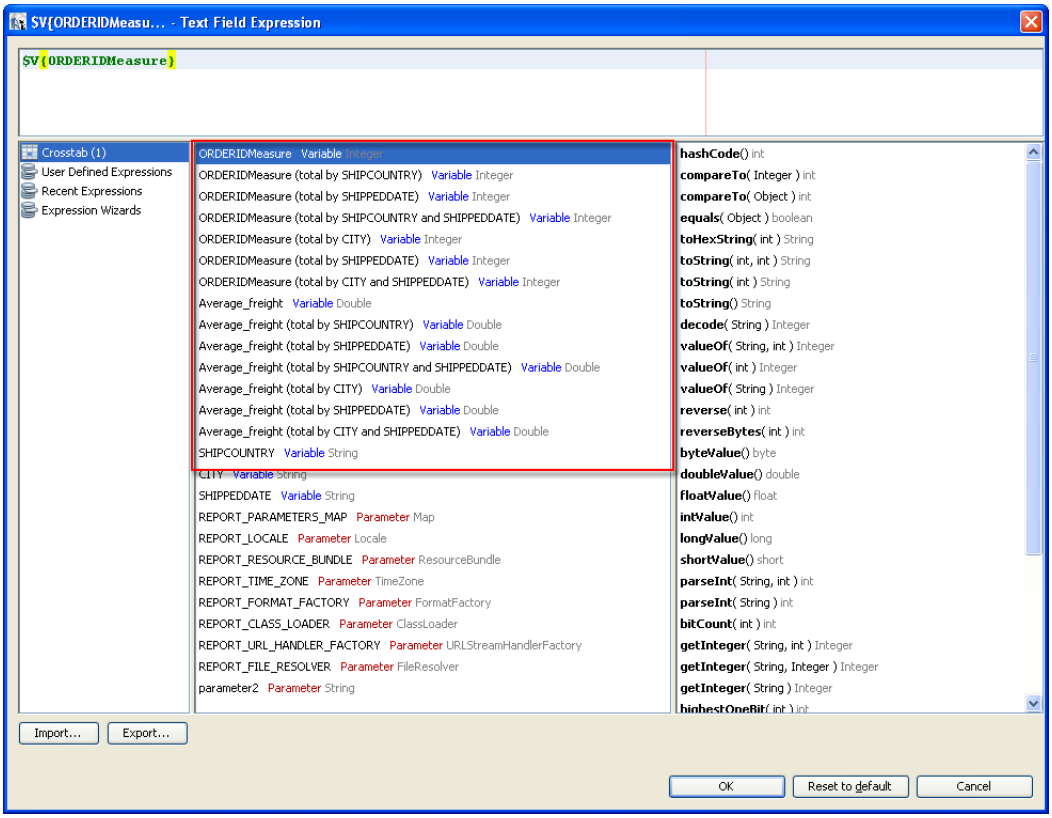



**Figure 13-20 Total variables are suggested in the expression editor**

For each measure, JasperReports creates variables that store the total value of the measure by each row/column group.

The following example is a simple report that shows the number of orders shipped in several regions for several years. **Figure 13-21** shows the simple crosstab for this example, and **Figure 13-22** shows the printed results of this crosstab.

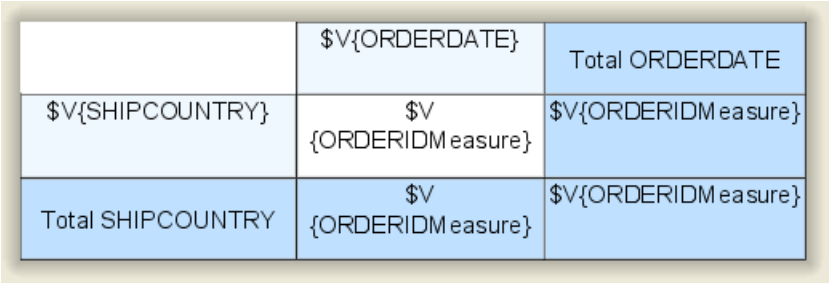
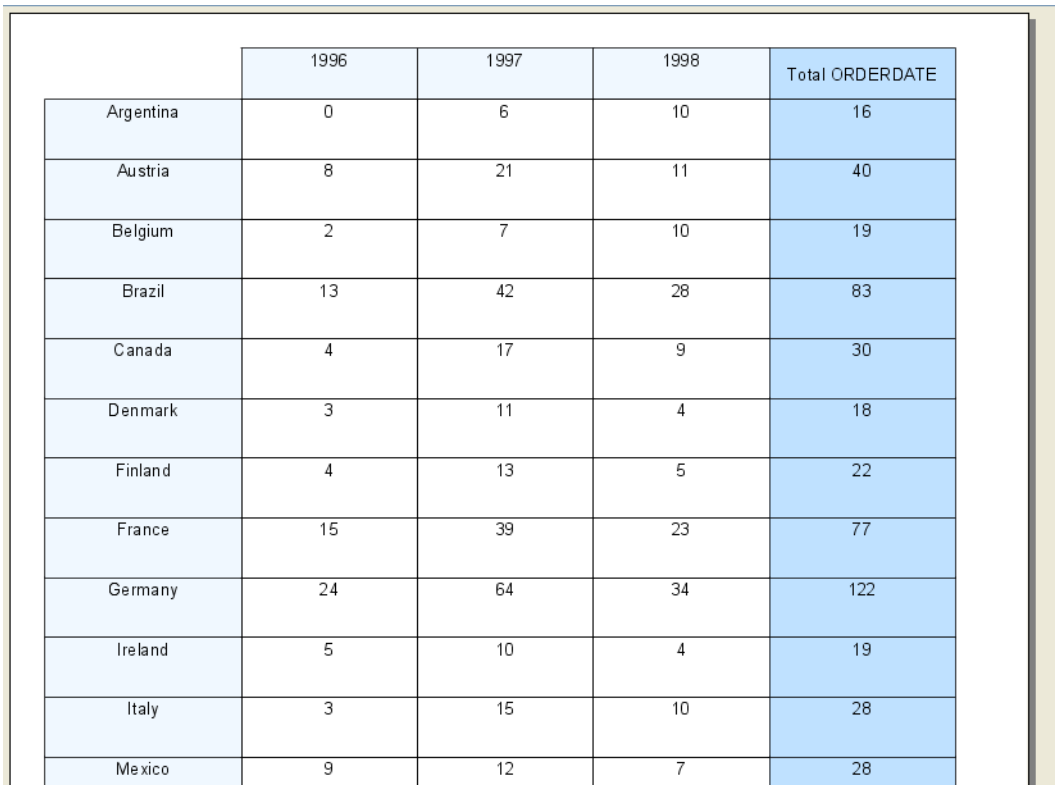



**Figure 13-21 Simple crosstab layout**



| | 1996 | 1997 | 1998 | Total ORDERDATE |
|---|---|---|---|---|
| Argentina | 0 | 6 | 10 | 16 |
| Austria | 8 | 21 | 11 | 40 |
| Belgium | 2 | 7 | 10 | 19 |
| Brazil | 13 | 42 | 28 | 83 |
| Canada | 4 | 17 | 9 | 30 |
| Denmark | 3 | 11 | 4 | 18 |
| Finland | 4 | 13 | 5 | 22 |
| France | 15 | 39 | 23 | 77 |
| Germany | 24 | 64 | 34 | 122 |
| Ireland | 5 | 10 | 4 | 19 |
| Italy | 3 | 15 | 10 | 28 |
| Mexico | 9 | 12 | 7 | 28 |

**Figure 13-22 The result of the simple layout**

To calculate the percentage of orders placed in each region per year, add a text field with the following Java expression:

```
new Double(
    $V{ORDERIDMeasure}.doubleValue()
    /
    $V{ORDERIDMeasure_ORDERDATE_ALL.doubleValue()
)
```

Or, if you use Groovy as suggested, simply:

```
(double)$V{ORDERIDMeasure} / (double)$V{ORDERIDMeasure_ORDERDATE_ALL
```

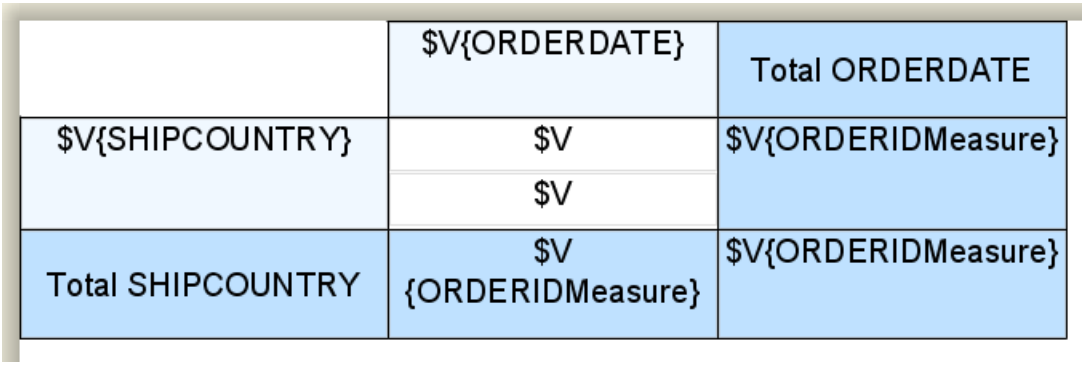
The basic formula that we are implementing is:

(Number of orders placed in this region and in this year) / (All orders shipped in this region)

A percentage must be treated as a floating-point number. For this reason, extract the double-scalar value from the `ORDERIDMeasure` and `ORDERIDMeasure_ORDERDATE_ALL` objects even if there are objects of class-type Integer (actually, a percentage derives from a number between 0 and 1 multiplied by 100).

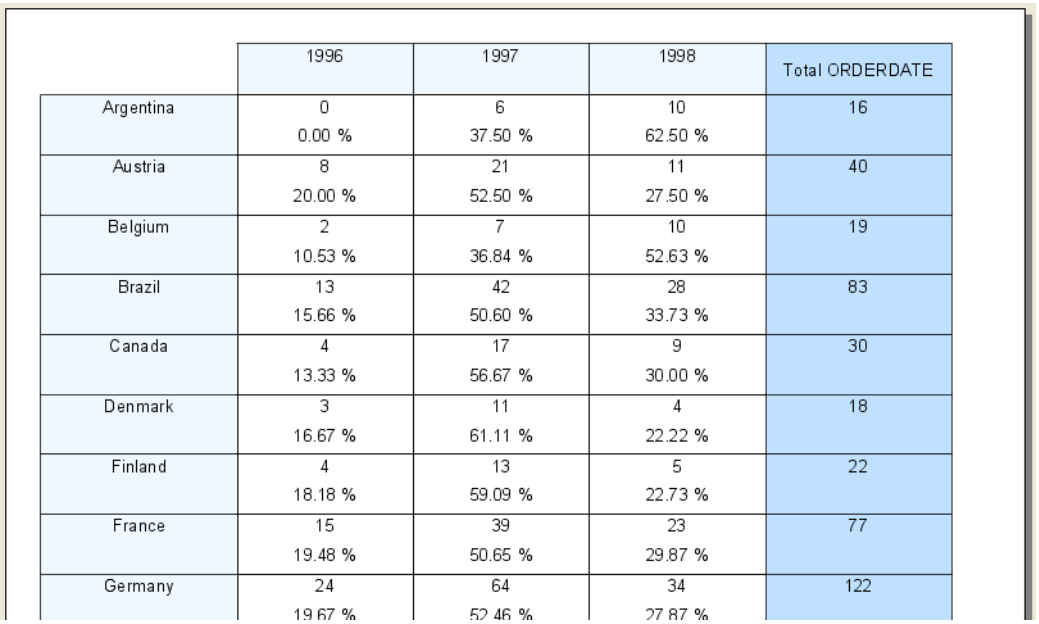
To include the value of the expression as a percentage, set the value of the pattern text field attribute to `#,##0.00 %`.

**Figure 13-23** shows the modified crosstab in the design window, and **Figure 13-24** shows the final printed results.



| | $V{ORDERDATE} | Total ORDERDATE |
|---|---|---|
| $V{SHIPCOUNTRY} | $V | $V{ORDERIDMeasure} |
| | $V | |
| Total SHIPCOUNTRY | $V {ORDERIDMeasure} | $V{ORDERIDMeasure} |

**Figure 13-23 A second field has been added in the measure cell to show the percentage**



| | 1996 | 1997 | 1998 | Total ORDERDATE |
|---|---|---|---|---|
| Argentina | 0<br>0.00 % | 6<br>37.50 % | 10<br>62.50 % | 16 |
| Austria | 8<br>20.00 % | 21<br>52.50 % | 11<br>27.50 % | 40 |
| Belgium | 2<br>10.53 % | 7<br>36.84 % | 10<br>52.63 % | 19 |
| Brazil | 13<br>15.66 % | 42<br>50.60 % | 28<br>33.73 % | 83 |
| Canada | 4<br>13.33 % | 17<br>56.67 % | 9<br>30.00 % | 30 |
| Denmark | 3<br>16.67 % | 11<br>61.11 % | 4<br>22.22 % | 18 |
| Finland | 4<br>18.18 % | 13<br>59.09 % | 5<br>22.73 % | 22 |
| France | 15<br>19.48 % | 39<br>50.65 % | 23<br>29.87 % | 77 |
| Germany | 24<br>19.67 % | 64<br>52.46 % | 34<br>27.87 % | 122 |

**Figure 13-24 The final report with percentages included**

# 14 INTERNATIONALIZATION

Internationalizing a report means making all static text set at design time (such as labels and messages) adaptable to locale options used to generate the report: the report engine will print the text using the most appropriate available translation. The text translations in the different languages supported by the report are stored in particular resource files named the Resource Bundles.

Moreover, this chapter covers the built-in function `msg()` and how it's possible to “localize” very complex sentences created dynamically.

## 14.1 Using a Resource Bundle Base Name

When you internationalize a report, it's necessary to find all the display text included in the report design that needs to be customized for a location, such as labels and static strings. A key (a name) is associated with every text fragment and is used to recall these fragments. These keys and the relative text translation are written in special files (one per language). Below is an example of a text localization mapping file.

```
Title_GeneralData=General Data
Title_Address=Address
Title_Name=Name
Title_Phone=Phone
```

All files containing this information have to be saved with the `.properties` file extension. The effective file name (i.e., the file name without the file extension and the language/country code, which you will see later in this section) represents the report Resource Bundle Base Name (e.g., the Resource Bundle Base Name for the resource file `i18nReport.properties` is `i18nReport`). When you generate an instance of the report, the report engine will look in the classpath for a file that has as a name the Resource Bundle Base Name plus the .properties extension (so for the previous example, it will look for a file named

exactly `i18nReport.properties`). If the report engine cannot find the file, it uses the default mapping resource defined for the report. The Resource Bundle Base Name is specified using the report property sheet as shown in Figure 14-1.

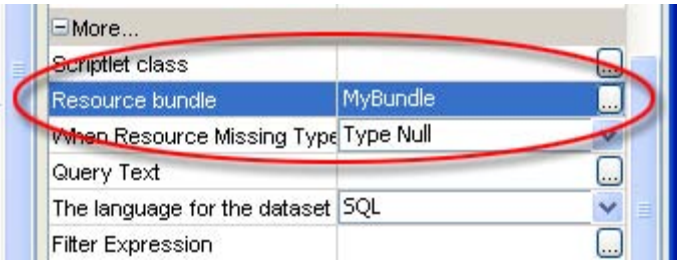



**Figure 14-1 The Resource bundle base name property**

When you need to generate a report using a specific locale, JasperReports looks for a file starting with the Resource Bundle Base Name string, followed by the language and country code relative to the requested locale. For example, `i18nReport_it_IT.properties` is a file that contains all locale strings to print in Italian; in contrast, `i18nReport_en_US.properties` contains the translations in American English.*

So it's important to always create a default resource file that will contain all the strings in the most widely used languages and a set of language-specific files for other languages.

The default resource file does not have a language/country code after the Resource Bundle Base Name, and the contained values are used only if there is no resource file that matches the requested locale, or if the file does not include the key for a translated string.

The complete resource file name is composed as follows:

```
<resource bundle base name>[_language code[_country code[_other code]]].properties
```

Here are some examples of valid resource file names:

```
i18nReport_fr_CA_UNIX
i18nReport_fr_CA
i18nReport_fr
i18nReport_en_US
i18nReport_en
i18nReport
```

The “other” code (or alternative code) is usually not used for reports, but is included to identify very specific resource files. The alternative code is appended after the language and the country code ( `_UNIX` in the preceding example).

If a resource key is not found in any of the suitable resource bundles, you can choose what to do by setting the report property `When Resource Missing Type`. The possible options are:

| | |
|---|---|
| *Type Null* | the null value is used in the expression (resulting in the string “null”) |
| *Type Empty* | the empty string is used |
| *Rise an error* | this will stop the filling process throwing a Java exception |
| *Type the key* | the value of the key is used as value |

iReport provides built-in support for editing the resource bundle files used for report localization.

* The language codes are the lower-case, two-letter codes as defined by ISO-639 (a list of this codes is available i.e. at this site: http://www.loc.gov/standards/iso639-2/php/English_list.php), the country codes are the upper-case, two-letter codes as defined by ISO-3166 (a list of this codes is available i.e. at this site: http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html).

To create a new resource bundle, select **New→Resource Bundle** (see **Figure 14-2**).

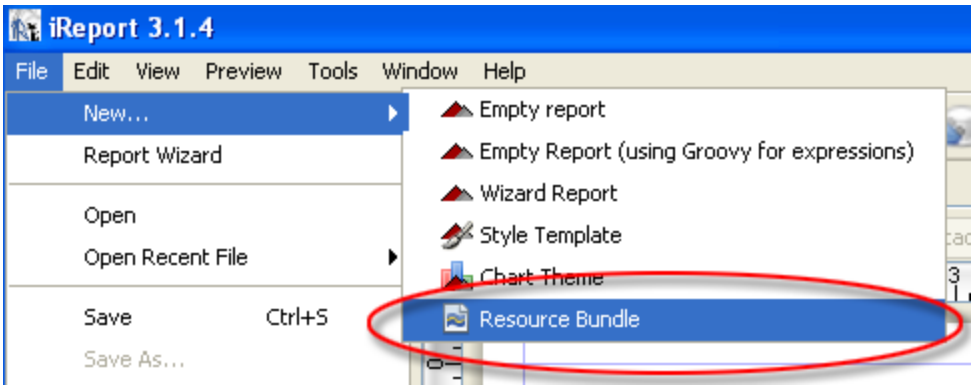



**Figure 14-2 Creating a new resource bundle**

Select the file name and where to save it. When done, iReport will open the file as simple text file (see **Figure 14-3**).

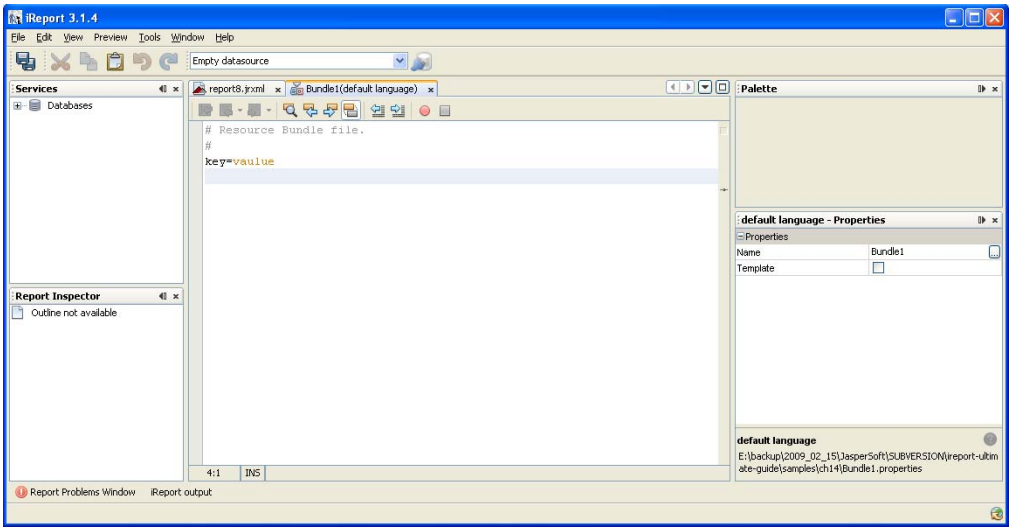



**Figure 14-3 The new resource bundle**

This is just the default bundle. To add new languages and/or switch to the visual editor for resource bundles, you need to locate the file using the favorites window. Select **Window→Favorites** to open the Favorites view, and locate the resource bundle file

directory. It is important you add the containing directory, not the file itself, otherwise you'll be not able to see the language specific bundles (see **Figure 14-4**).

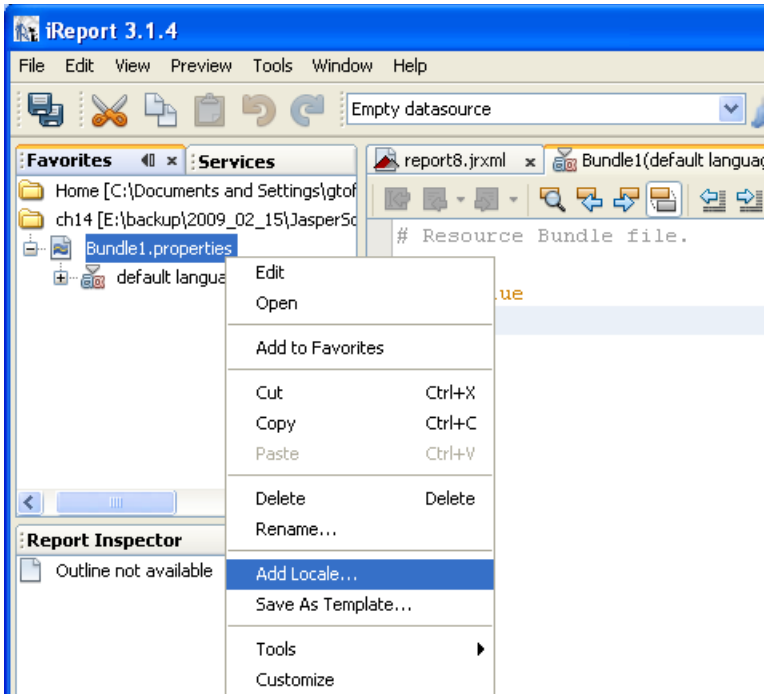



**Figure 14-4 The resource bundle in the favorites view**

The bundle node in favorites provides several tool features. Right-click the file node to open one of the editing tools for the resource bundle:

- **Edit** - Open a resource bundle as a text file (see **Figure 14-3**)
- **Open** - Opens the visual resource bundle editor that shows at the same time the translations for all the language you are supporting (see **Figure 14-5**).

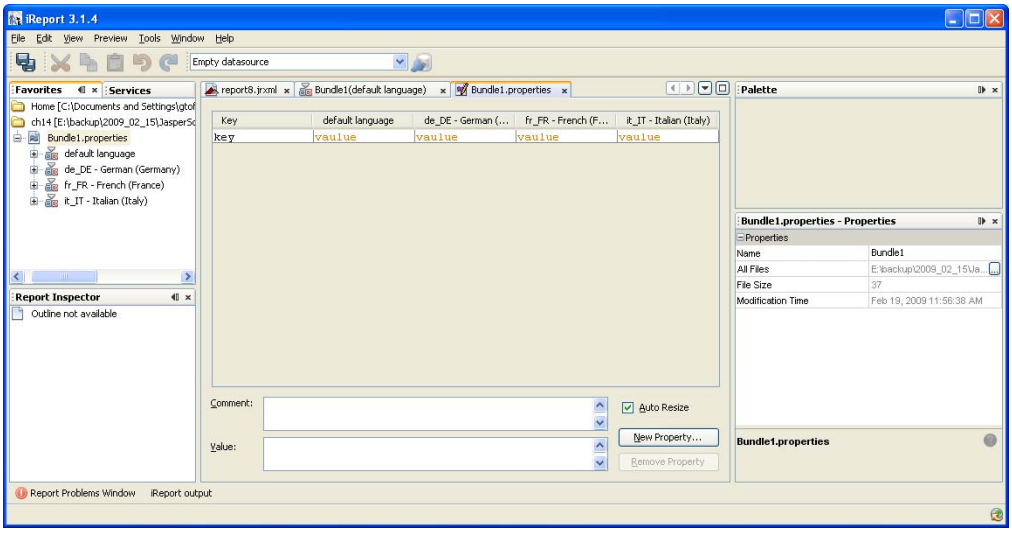



**Figure 14-5 The Visual Bundle Editor**

To add a new locale (meaning the support for a new language), right-click the resource bundle node and select the menu item **Add locale...**. This will pop up the window shown in **Figure 14-6**.

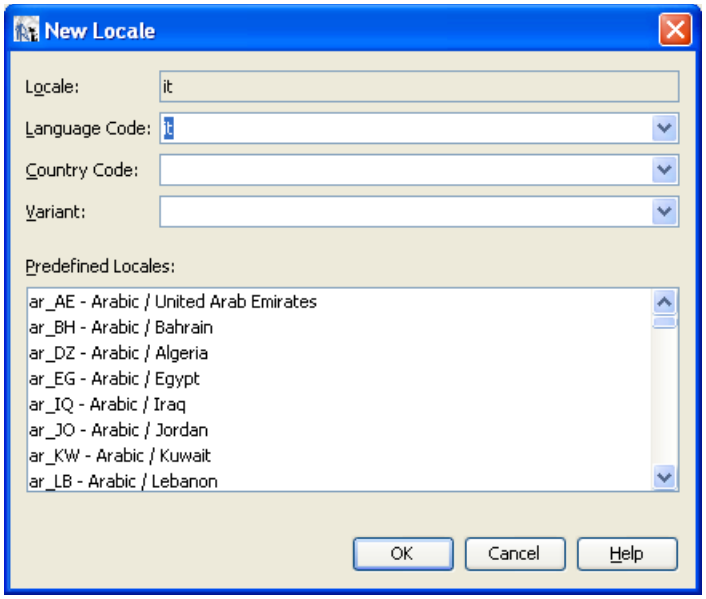



**Figure 14-6 New locale**

It is used to set the correct language specifications (language, country and optionally a variant code).

By confirming the choice, a new file will be created in the same directory as the default one with the proper localization abbreviation appended to the file name, and it will be visible as child of the bundle node in the favorites view.

## 14.2 Retrieving Localized Strings

There are two ways to retrieve the localized string for a particular key inside a JasperReports expression:

- Use the built-in `str("key name")` function
- Use the special syntax `$R{key name}`

Here is an example expression to retrieve a localized string:

```
$R{hello.world}
```

JasperReports converts the text associated with the key `hello.world` using the most appropriate available translation for the selected locale.

## 14.3 Formatting Messages

The internationalization features included with JasperReports are based on the support provided by Java. One of the most useful features is the `msg` function, which you can use to dynamically build messages using arguments. In fact, `msg` uses strings as patterns. These patterns define where arguments, passed as parameters to the `msg` function, must be placed. The position of an argument is expressed using numbers between braces, as in this example:

```
“The report contains {0} records.”
```

The zero specifies where to place the value of the first argument passed to the `msg` function. The expression:

```
msg($R{text.message}, $P{number})
```

uses the string referred to by the key `text.message` as the pattern for the call to `msg`. The second parameter is the first argument to be replaced in the pattern string. If `text.message` is the string `"The report contains {0} records."` and the value for the report parameter number is 100,JasperReports displays the interpreted text string as:

```
The report contains 100 records.
```

The reason for using patterns instead of building messages like this by dividing them into substrings translated separately (for example, `[The report contains] {0} [records]`), is that sometimes the second approach is not possible. Localization modules may not be able to create grammatically correct translations for all languages (e.g., for languages in which the verb appears at the end of the sentence).

It's possible to call the `msg` function in three ways, as shown below:

```
public String msg(String pattern, Object arg0)
public String msg(String pattern, Object arg0, Object arg1)
public String msg(String pattern, Object arg0, Object arg1, Object arg2)
```

The only difference between the three calls is the number of arguments passed to the function.

## 14.4 Deploying Localized Reports

To deploy a localized report, you must make sure that all `.properties` files containing the translated strings are present in the classpath.

`JasperReports` looks for resource files using the `getBundle` method of the `ResourceBundle` Java class. To learn more about how this class works, visit http://java.sun.com/docs/books/tutorial/i18n/, where you will find all the main concepts about how Java supports internationalization fully explained.

## 14.5 Generating a Report Using a Specific Locale and Time Zone

If you wish to use a specific locale or to generate a report using a particular time zone, go to the iReport options dialog (**Tools→Options**) and specify the preferred locale and time zone in the Report execution options section (see **Figure 14-7**).

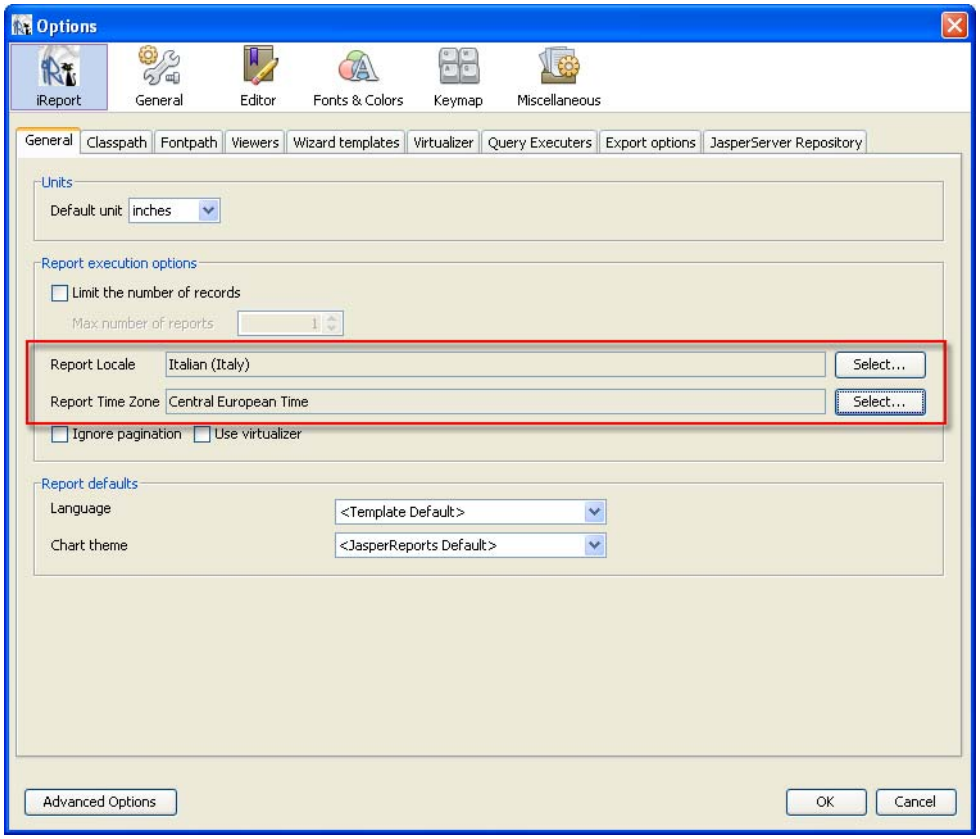



**Figure 14-7 Locale and Time Zone Options**

You will see the current settings in the log window each time you run a report, as shown in **Figure 14-8**.

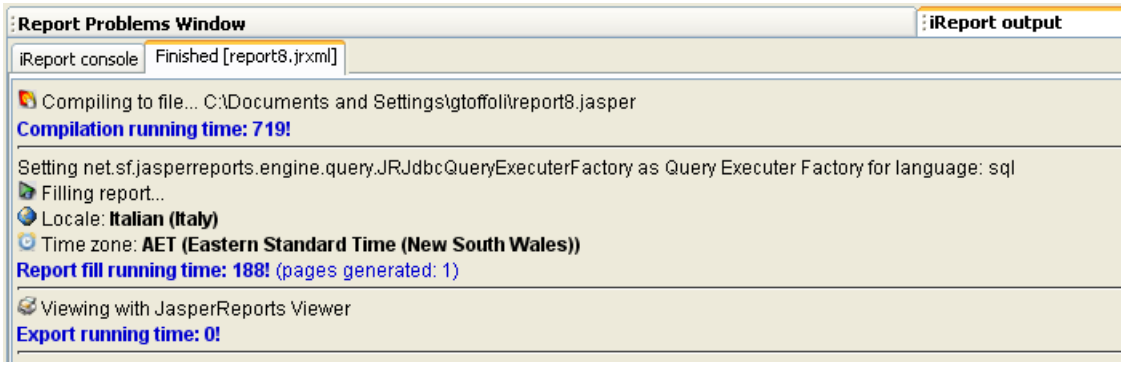



**Figure 14-8 Locale and time zone used are specified in the execution log**

# 15 SCRIPTLETS

A scriptlet is a Java class used to execute special elaborations during report generation. A scriptlet exposes a set of methods that are invoked by the reporting engine when some particular events, like the creation of a new page or the end of processing a detail row, occur.

In this chapter, you will see how to write a simple scriptlet and how to use it in a report. You will also see how iReport handles scriptlets and what methods are useful when deploying a report using this kind of functionality.

## 15.1 Understanding the `JRAbstractScriptlet` Class

To implement a scriptlet, you have to extend the Java class `net.sf.jasperreports.engine.JRAbstractScriptlet`. This class exposes all the abstract methods to handle the events that occur during report generation and provides data structures to access all variables, fields, and parameters present in the report.

The simplest scriptlet implementation is provided directly by JasperReports: it is the class `JRDefaultScriptlet`, shown in Figure 15-1, which extends the class `JRAbstractScriptlet` and implements all the required abstract methods with a void function body.

**Figure 15-1** `JRDefaultScriptlet`

```
package net.sf.jasperreports.engine;

/**
 * @author Teodor Danciu (teodord@users.sourceforge.net)
 * @version $Id: JRDefaultScriptlet.java,v 1.3 2004/06/01 20:28:22 teodord Exp $
 */
public class JRDefaultScriptlet extends JRAbstractScriptlet
{
    public JRDefaultScriptlet() {    }

    public void beforeReportInit() throws JRScriptletException
    {
    }

    public void afterReportInit() throws JRScriptletException
    {
    }

    public void beforePageInit() throws JRScriptletException
    {
    }

    public void afterPageInit() throws JRScriptletException
    {
    }

    public void beforeColumnInit() throws JRScriptletException
    {
    }

    public void afterColumnInit() throws JRScriptletException
    {
    }

    public void beforeGroupInit(String groupName) throws JRScriptletException
    {
    }

    public void afterGroupInit(String groupName) throws JRScriptletException
    {
    }

    public void beforeDetailEval() throws JRScriptletException
    {
    }

    public void afterDetailEval() throws JRScriptletException
    {
    }
}
```

As you can see, the class is formed by a set of methods with a name composed using the keyword `after` or `before` followed by an event or action name (e.g., `DetailEval` and `PageInit`). These methods map all of the events that can be handled by a scriptlet, which are summarized in Table 15-1.

**Table 15-1Report Events**

| Event/Method | Description |
|---|---|
| Before Report Init | This is called before the report initialization (i.e., before all variables are initialized). |
| After Report Init | This is called after all variables are initialized. |
| Before Page Init | This is called when a new page is created, before all variables having reset type Page are initialized. |
| After Page Init | This is called when a new page is created and after all variables having reset type Page are initialized. |
| Before Column Init | This is called when a new column is created, before all variables having reset type Column are initialized; this event is not generated if the columns are filled horizontally. |
| After Column Init | This is called when a new column is created, after all variables having reset type Column are initialized; this event is not generated if the columns are filled horizontally. |
| Before Group x Init | This is called when a new group x is created, and before all variables having reset type Group and group name x are initialized. |
| After Group x Init | This is called when a new group x is created, and after all variables having reset type Group and group name x are initialized. |
| Before Detail Eval | This is called before a detail band is printed and all variables are newly evaluated. |
| After Detail Eval | This is called after a detail band is printed and all variables are evaluated for the current record. |

Inside the scriptlet, you can refer to all of the fields, variables, and parameters using the following maps (`java.util.HashMap`) defined as class attributes:

- `fieldsMap`
- `variablesMap`
- `parametersMap`

The groups (if present in the report) can be accessed through the attribute `groups`, an array of `JRFillGroup`.

## 15.2 Creating a Simple Scriptlet

Like all the Java classes, to create a scriptlet you just need a simple text editor and a java compiler. But we are not all hardcore developers, so let's assume we are using an IDE (Integrated Development Environment) to do this. My favorite IDE is NetBeans (http://www.netbeans.org), but the instructions shown here should be generic enough to understand how to do the same using another IDE like Eclipse or whatever.

The scriptlet we want to write will calculate the time taken to fill each page, and print it in the page footer. We could store the system time when a new page is created using the method `afterPageInit` and provide a method to get the milliseconds past from that start time. We can then show how long the page took to be rendered by printing this value in a textfield with evaluation time *Page*. This is a good example of a scriptlet that can be used to profile a report execution too.

We will start with a new Java project (if you are working on a Java application, the project could be the same), anyway let's be generic. In NetBeans this is done selecting **File → New Project**. The window in **Figure 15-2** pops up.

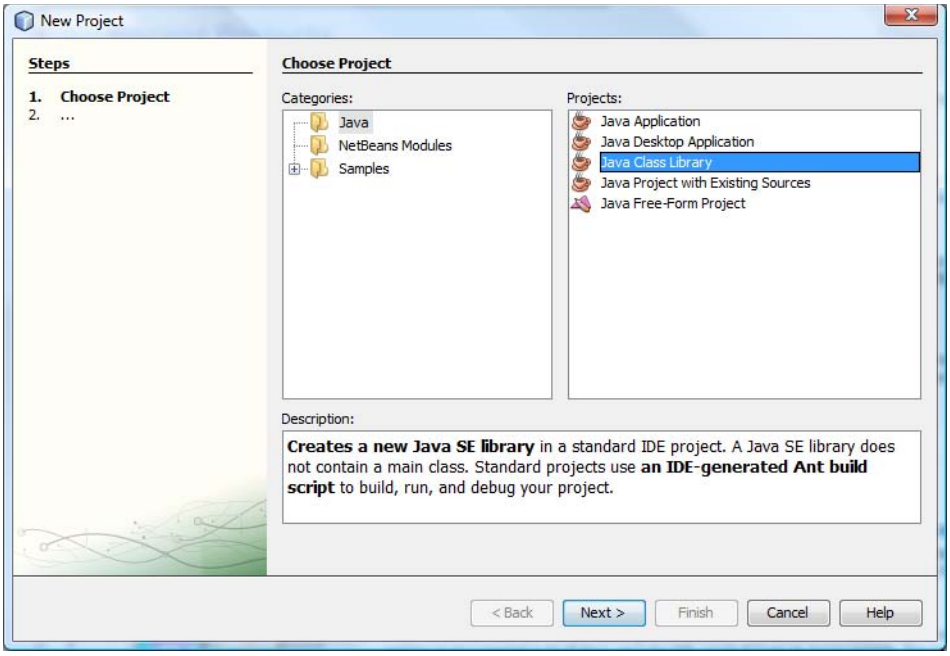



**Figure 15-2 Creation of a new project in NetBeans IDE**

The best type of project is a Java Class Library; after all, we just have to write a simple class, not a real application.

In the second step give a name to the project (i.e. Scriptlet) and choose a location for it (**Figure 15-3**).

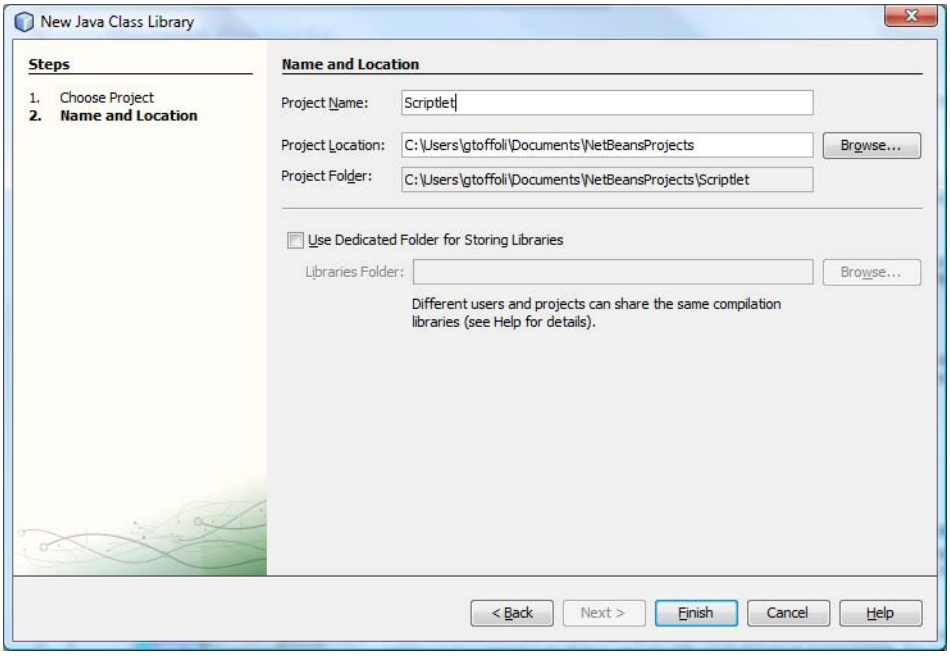



**Figure 15-3 Project Details**

The project is now created. The next step is to add to the project the jars required to write your scriptlet. Technically the only one required is `jasperreports.jar`, but you could add other jars if required by your particular scriptlet. To do it in NetBeans, right click the **Libraries** node in the Projects view, and select `Add JAR/Folder` (**Figure 15-4**).

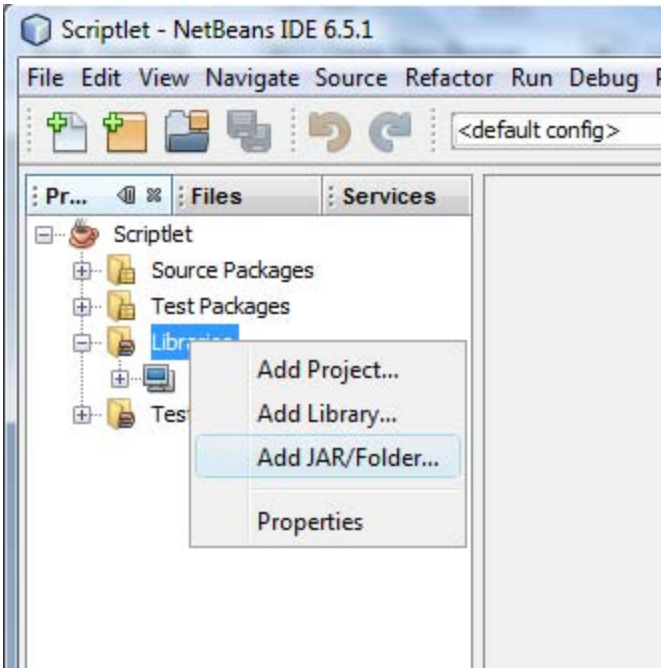



**Figure 15-4 Adding a Jar to the project**

Locate the jasperreports jar on your computer. If you have never downloaded a distribution of JasperReports, you can find a copy of this jar in your iReport installation directory, more precisely at this location:

```
<ireport installation directory>/ireport/modules/ext/jasperreports-x.y.z.jar
```

Now that we have in the project classpath all the required classes, let's move on creating a package for the scriptlet and implementing it. Right click the **Source Packages** node in Projects and select **New > Java Class**. The window to create the

new class pops up (**Figure 15-5**). In the sample I'll set as class name `MyScriptlet` and I'll set the package name to `com.mycompany`.

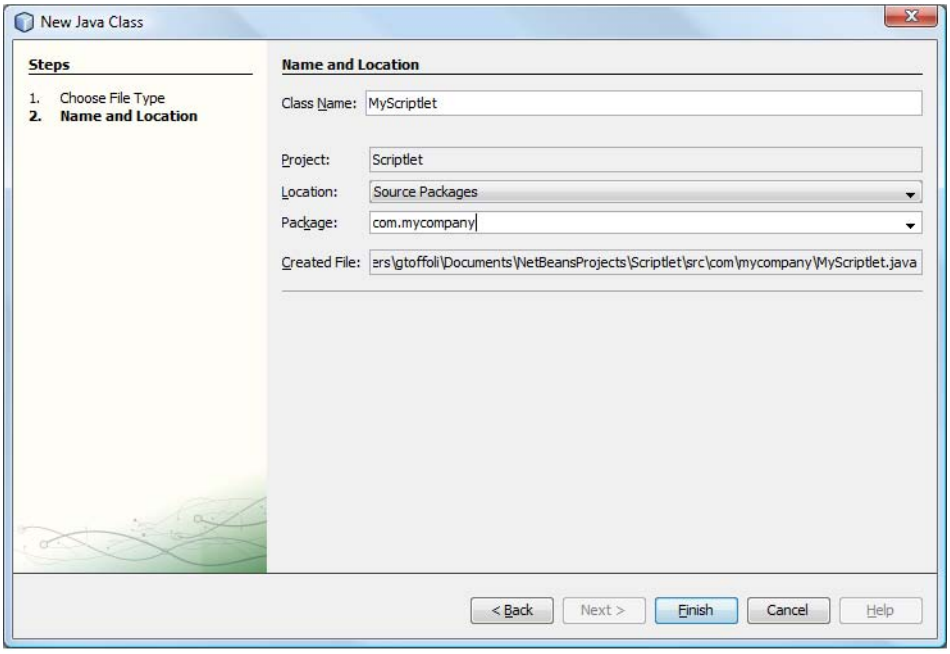



**Figure 15-5 New Java Class**

When finished, the class is opened in the Java editor. The class must extend the class `net.sf.jasperreports.engine.JRDefaultScriptlet`.

Figure 15-6 shows the source code of the scriptlet.

**Figure 15-6 Example Scriptlet Source Code**

```
package com.mycompany;

import net.sf.jasperreports.engine.JRDefaultScriptlet;
import net.sf.jasperreports.engine.JRScriptletException;


public class MyScriptlet extends JRDefaultScriptlet {

    long pageInitTime = 0;

    @Override
    public void beforePageInit() throws JRScriptletException {

        pageInitTime = new java.util.Date().getTime();
    }

    /**
     * @return the time past from the last page init
     */
    public Long getLastPageTime() {

        long now = new java.util.Date().getTime();
        return new Long(now - pageInitTime);
    }

}
```

We just overrode the method `beforeInitPage`, and added the method `getLastPageTime`. This last method returns a Long object. It is always good to return Objects, since that is what we use in expressions.

Building the project, NetBeans will create a jar, that is exactly what we need. Press the build button, in the output window we will find the exact location of the created jar containing the scriptlet (**Figure 15-7**).

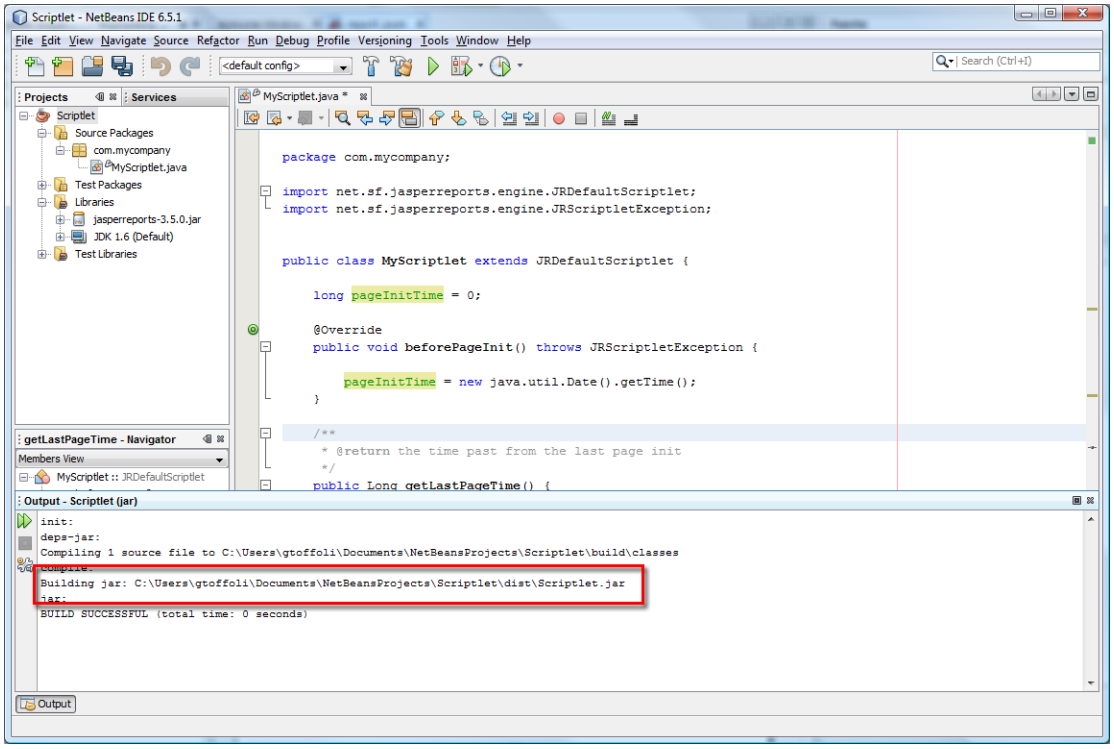



**Figure 15-7 Build Output**

## 15.3 Testing a Scriptlet in iReport

One of the most interesting feature of iReport is the ability to dynamically load jars. This allow the developers to rebuild their jars and test fresh versions in iReport without having to restart it.

The first step to test the scriptlet (in other words to use it), is to add the jar we just created in the IDE to the iReport classpath, which allows it to be reloaded.

Open the iReport options dialog (**Tools → Options**), move to the iReport section and select the **Classpath** tab (**Figure 15-8**). Add to the list of classpath entries the scriptlet jar and check the Reloadable flag.

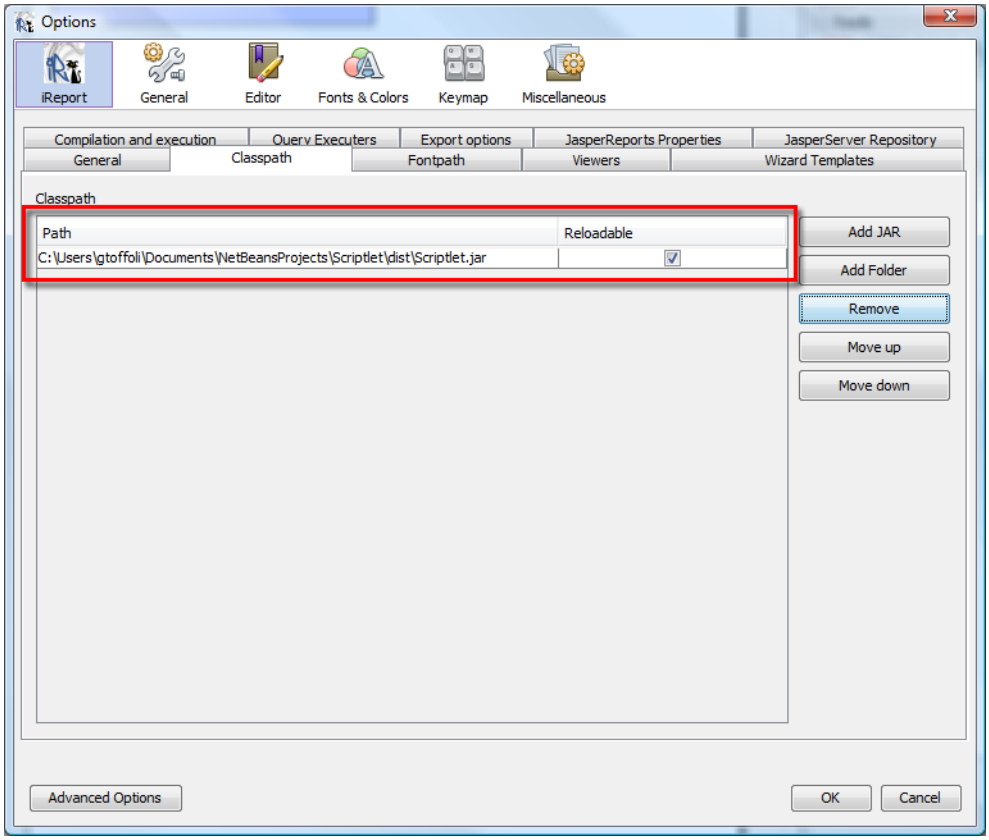



**Figure 15-8 Adding the jar to the iReport classpath**

Now we can use the class `com.mycompany.MyScriptlet` in any report. If you already have a report to modify, just open it, otherwise create a new report that can produce several pages or reuse one of the samples shown in the previous chapters.

A report can use one or more scriptlets. If you just use one, set the Scritplet property of the report with the full qualified name of your scriptlet class (in this case `com.mycompany.MyScriptlet`), as shown in **Figure 15-9**.

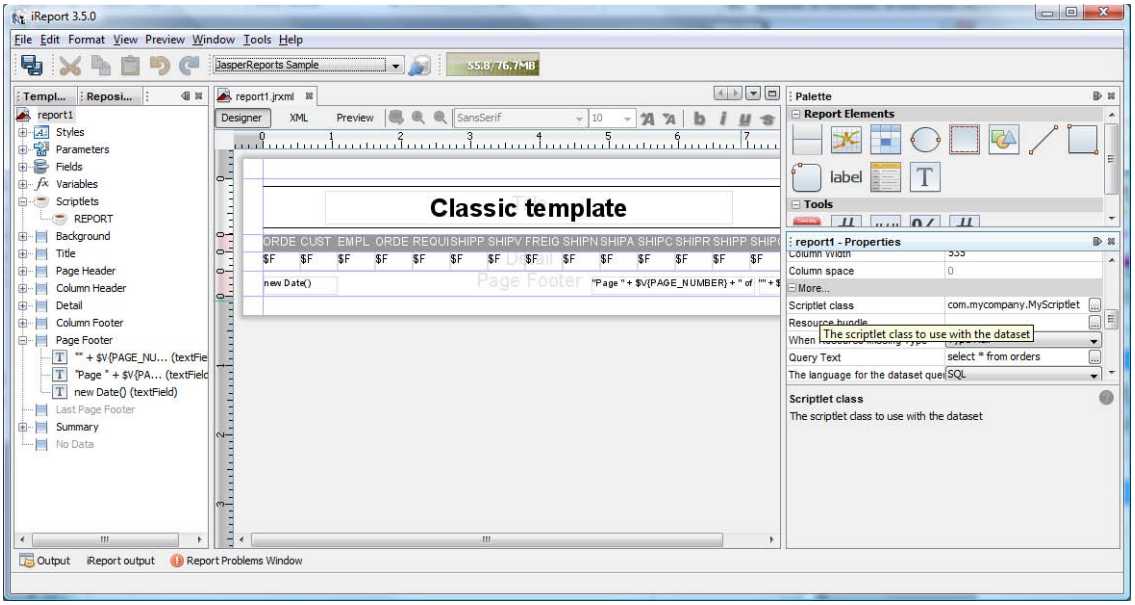



**Figure 15-9 Setting the Scriptlet property**

If you want to add more scriptlets, use the Report Inspector view, in which you can find a node called Scriptlets. The subnode REPORT can not be removed, and identifies always the first scriptlet of the report (the one set in the Scritplet property). To add another scriptlet right click the Scriptlets node and select *Add Scriptlet*, finally set the correct class name for that scriptlet in the property sheet view (which is set by default to `net.sf.jasperreports.engine.JRDefaultScriptlet`).

It's time to use the scriptlet we just created. Add a textfield to the page footer band. The expression of the textfield will be:

```
$P{REPORT_SCRIPTLET}.getLastPageTime()
```

`REPORT_SCRIPTLET` is the built-in parameter to reference the scriptlet in the report. The other scritplets can be references with the name <scriptlet name>_SCRIPTLET (i.e. scritplet1_SCRIPTLET). In this report, REPORT_SCRIPTLET has type `MyScriptlet` so we can call the method getLastPageTime() that returns the number of milliseconds past from the last page initialization. This method returns a Long object, so we need to adjust the textfield class type (setting it to java.lang.Long). Finally, since we want to get the past time only when the page is complete, set the evaluation time of the textfield to *Page* (see **Figure 15-10**).

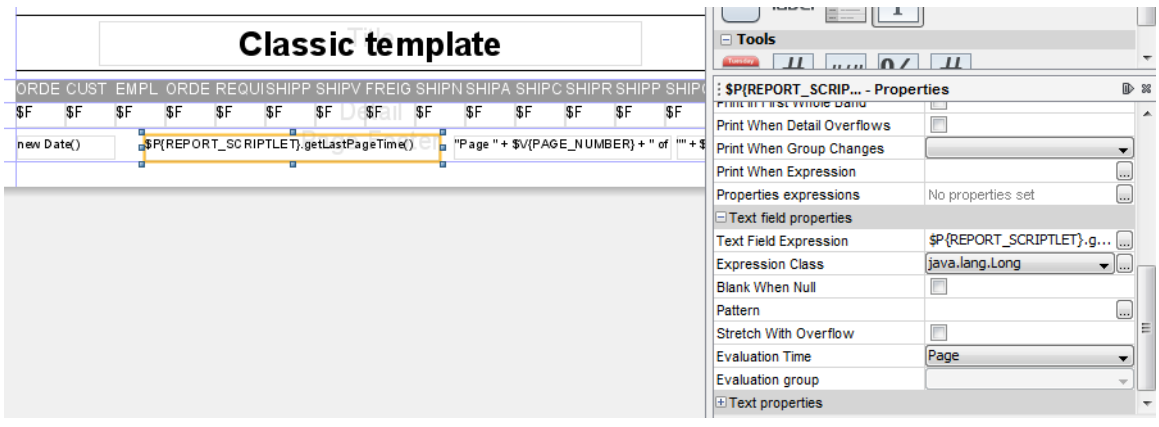



**Figure 15-10 The textfield details**

Preview the report to see the result. On each page you can read the exact number of millisecond required to fill the particular page. It is interesting to see how the first page usually takes much more than the other ones, while the last one is often most fast.

If you want to change something in the scriptlet, just back to the IDE, rebuild it and preview the report again (maybe using the *Run again* button in the toolbar of the preview tab in iReport).

## 15.4 Accessing report objects

The scriptet class has access to all the object of the dataset it belongs to (a subdataset can have its own scriptlet). All the scriptlets inherit from `JRAbstractScriptlet` class three `java.util.Map`'s (`parametersMap`, `fieldsMap` and `variablesMap`) and an array of `JRFillGroup` called `groups`. The maps use the object names as keys, and some special objects (JRFillParameter, JRFillField and JRFillVariable) as values. Some convenient functions are provided to get the value of the objects and to set the value for variables:

```
Object getParameterValue(String parameterName)
Object getParameterValue(String parameterName)
Object getParameterValue(String parameterName, boolean mustBeDeclared)
Object getFieldValue(String fieldName)
Object getVariableValue(String variableName)
void setVariableValue(String variableName, Object value)
```

All of them throw a `JRScriptletException` in case of error.

Variables are the only objects for which a scriptlet can change the value. Pay attention to the fact that when a scriptlet set a value for a variable, it could be in concurrence with the reporting engine (in general when a calculation type has been set for the variable). For this reason, variable that are supposed to be used by a scriptlet should always have *System* as calculation type.

Be able to access the report objects, not just their value is a great advantage, especially when a scriptlet is thought to be reusable (since you can identify for instance a parameter or a field having a certain custom property). In particular the `JRFillParameter` and `JRFillField` provide a way to read the their properties set at design time, and both `JRFillField` and `JRFillVariable` expose the previous value, useful for differential calculations.

## 15.5 Debugging a Scriptlet

Unfortunately there is no way to run a step by step debugger to debug a scriptlet setting breakpoints in the code, but there are several techniques to monitor the scriptlet execution

One of them is by using a simple `System.out.println(<msg>)` to print informations that will appear in the iReport output console. It is good practice to follow the println call with a flush to be sure the printed message is shows as soon as possible in the output view. Here is an example:

```
public void beforePageInit() throws JRScriptletException {

        pageInitTime = new java.util.Date().getTime();

        System.out.println("I have set the pageInitTime to: " + pageInitTime);
        System.out.flush();
    }
```

This is what you get in the output console:

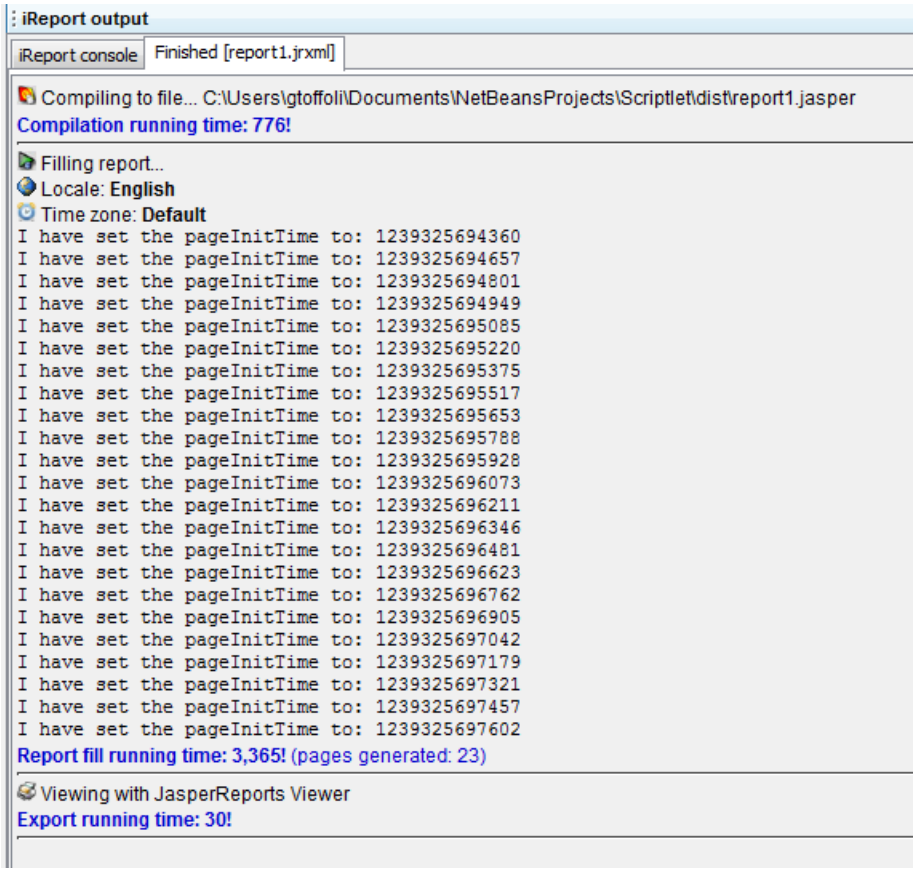



**Figure 15-11 Debug messages coming from the scriptlet**

A more sophisticated scriptlet (suitable only in a design environment) can pop up a dialog displaying some informations like the current status of the fields and the option to stop the execution, as shown in **Figure 15-12**.

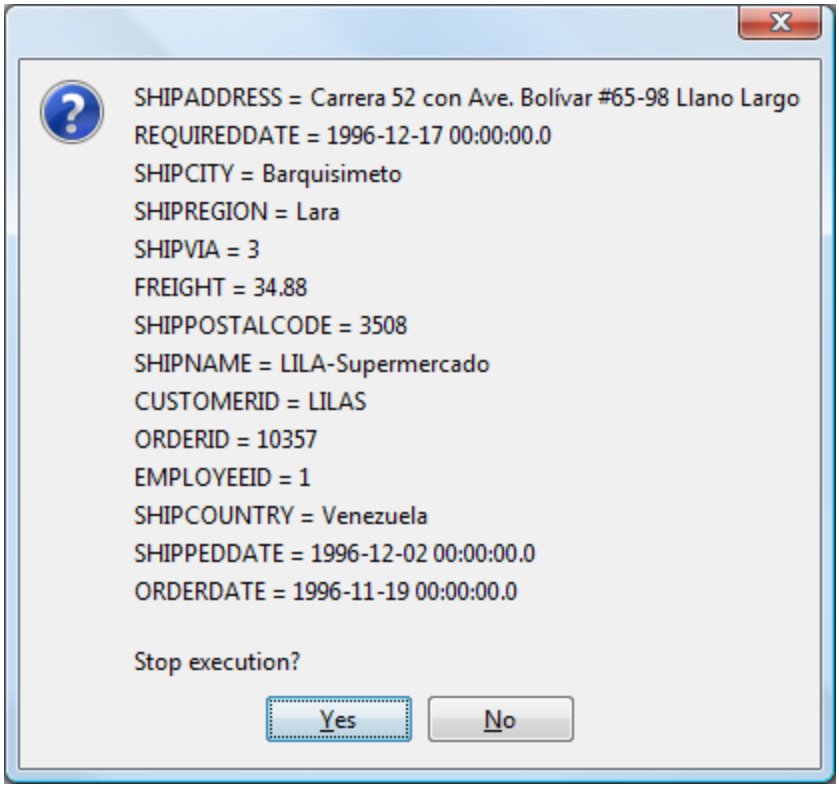



**Figure 15-12 Simple fields debug window**

The code of the scriptlet is the following:

**Figure 15-13 Debugging Scriptlet Example Source**

```
public void beforePageInit() throws JRScriptletException {

        pageInitTime = new java.util.Date().getTime();

        String fieldValuesMsg = "";

        Iterator i = fieldsMap.keySet().iterator();
        while (i.hasNext())
        {
            String fieldName = (String)i.next();
            fieldValuesMsg += fieldName + " = " + getFieldValue(fieldName) + "\n";
        }
        fieldValuesMsg += "\nStop execution?";

        if ( JOptionPane.showConfirmDialog(null, fieldValuesMsg,"",
                    JOptionPane.YES_NO_OPTION) == JOptionPane.OK_OPTION)
        {
            // Stop the execution
            throw new JRScriptletException("Execution interrupted by the user");
        }
    }
```

The dialog will pop up every time a new page is initialized (just because we are implementing the method `beforePageInit` pausing the report execution. If the user select Yes (stop the execution), the scriptlet throws a `JRScriptletException` that will terminate the report execution with the message “Execution interrupted by the user”. This technique can be used to automatically terminate a process that is taking too much time (until one of the scriptlet events is actually invoked), or when we are producing too much pages, and so on.

## 15.6 Deploying Reports that use Scriptlets

Sometime you may create a report works well in iReport, but that does not work when deployed in an external application. One of things to check is if the report is using a scriptlet and be sure that the scriptlet classes are available in the classpath. Finally, a scriptlet can be used in more than a single report: consider to create your own library of scriptlets and put all of them in a single jar.

# APPENDIX A CHART THEME EXAMPLE

Below is the XML source code for the custom chart theme “TricolorAreaChart” shown in **Figure 11-12 on page 201**.

```
<?xml version="1.0" encoding="UTF-8"?>
<chart-theme>
 <chart-settings
  background-image-alignment="Align.BOTTOM"
  border-visible="true"
 >
  <background-paint
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   color1="#CCFFFF"
   color2="#FFFFFF"
   xsi:type="gradient-paint"
  />
  <font/>
 </chart-settings>

 <title-settings>
  <font font-size="18"/>
 </title-settings>

 <subtitle-settings>
  <font font-size="12" italic="true"/>
 </subtitle-settings>

 <legend-settings>
  <font/>
 </legend-settings>

 <plot-settings
  background-image-alignment="Align.BOTTOM"
  outline-visible="false"
  domain-gridline-visible="true"
  range-gridline-visible="true"
 >
```

```
   <background-paint
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    color1="#FFFFFF" color2="#CCFFFF" xsi:type="gradient-paint"
   />
   <outline-paint
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    color="#000000" xsi:type="color"
   />
   <stroke
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" width="1.0"
    xsi:type="stroke"
   />
   <series-color-sequence
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    color="#00CC00" xsi:type="color"
   />
   <series-color-sequence
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    color="#FF0000" xsi:type="color"
   />
   <series-color-sequence
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    color="#FFFFFF" xsi:type="color"
   />
  </plot-settings>
  <domain-axis-settings
   line-visible="true"
  >
   <label-font
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="font"
   />
   <tick-label-font
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="font"
   />
  </domain-axis-settings>
  <range-axis-settings
   line-visible="false"
  >
   <label-font
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="font"
   />
   <tick-label-font
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="font"
   />
  </range-axis-settings>
</chart-theme>
```

**Figure 16-1 XML Source for TricolorAreaChart**

# INDEX