

Foreword by Mike Milinkovich,
Executive Director, Eclipse Foundation



Integrating and Extending **BIRT**

Jason Weathersby • Tom Bondur
Iana Chatalbasheva • Don French

Second Edition
Revised and Updated

Integrating and Extending BIRT

Second Edition

SERIES EDITORS Erich Gamma ■ Lee Nackman ■ John Wiegand

Eclipse is a universal tool platform, an open extensible integrated development environment (IDE) for anything and nothing in particular. Eclipse represents one of the most exciting initiatives hatched from the world of application development in a long time, and it has the considerable support of the leading companies and organizations in the technology sector. Eclipse is gaining widespread acceptance in both the commercial and academic arenas.

The Eclipse Series from Addison-Wesley is the definitive series of books dedicated to the Eclipse platform. Books in the series promise to bring you the key technical information you need to analyze Eclipse, high-quality insight into this powerful technology, and the practical advice you need to build tools to support this evolutionary Open Source platform. Leading experts Erich Gamma, Lee Nackman, and John Wiegand are the series editors.

Titles in the Eclipse Series

John Arthorne and Chris Laffra

Official Eclipse 3.0 FAQs

0-321-26838-5

David Carlson

Eclipse Distilled

0-321-28815-7

Eric Clayberg and Dan Rubel

Eclipse: Building Commercial-Quality Plug-Ins, Second Edition

0-321-42672-X

Adrian Colyer, Andy Clement, George Harley, and Matthew Webster

Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools

0-321-24587-3

Naci Dai, Lawrence Mandel, and Arthur Ryman

Eclipse Web Tools Platform: Developing Java™ Web Applications

0-321-39685-5

Erich Gamma and Kent Beck

Contributing to Eclipse: Principles, Patterns, and Plug-Ins

0-321-20575-8

Jeff McAffer and Jean-Michel Lemieux

Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications

0-321-33461-2

Diana Peh, Alethea Hannemann, Paul Reeves, and Nola Hague

BIRT: A Field Guide to Reporting

0-321-44259-8

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks

EMF: Eclipse Modeling Framework

0-321-33188-5

Jason Weathersby, Don French, Tom Bondur, Jane Tatchell, and Iana Chatalbasheva

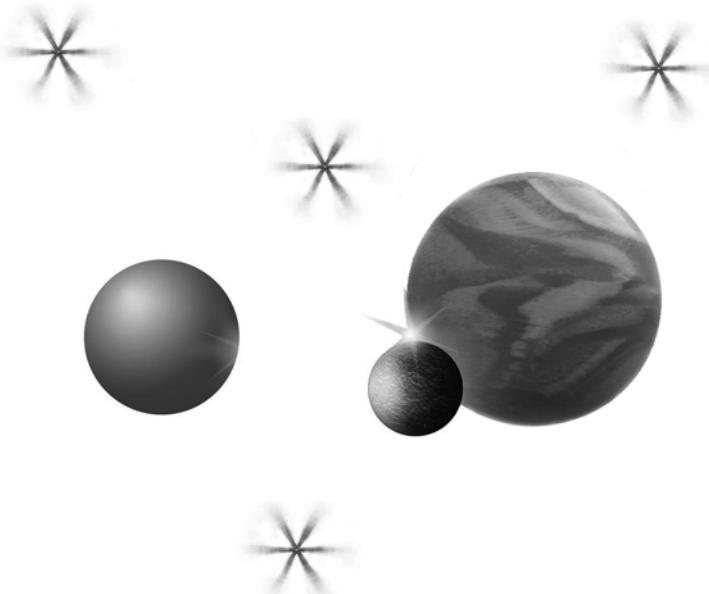
Integrating and Extending BIRT

0-321-44385-3



Integrating and Extending BIRT

Second Edition



Jason Weathersby • Tom Bondur • Iana Chatalbasheva
Don French

◆ Addison-Wesley

*Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City*

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Copyright© 2008 by Actuate Corporation

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-58030-6
ISBN-10: 0-321-58030-3

Text printed in the United States at OPM in Laflin, Pennsylvania.
First printing, June 2008

Foreword.....	xix
Preface	xxi
About this book	xxi
Who should read this book	xxii
Contents of this book	xxii
Typographical conventions	xxvi
Syntax conventions	xxvii
Acknowledgments	xxix
Part I Installing and Deploying BIRT.....	1
Chapter 1 Prerequisites for BIRT	3
Downloading Eclipse BIRT components	3
BIRT Report Designer software requirements	5
About types of BIRT builds	7
Chapter 2 Installing a BIRT Report Designer	9
Installing BIRT Report Designer Full Eclipse Install	9
Installing BIRT RCP Report Designer	10
Troubleshooting installation problems	11
Avoiding cache conflicts after you install a BIRT report designer	12
Specifying a Java Virtual Machine when starting BIRT report designer	12
Installing a language pack	13
Updating a BIRT Report Designer installation	14
Updating BIRT RCP Report Designer installation	15
Chapter 3 Installing Other BIRT Packages	17
Installing Chart Engine	17
Installing BIRT Data Tools Platform Integration	19
Installing BIRT Demo Database	19
Installing Report Engine	21
Installing BIRT Samples	23
Installing BIRT Source Code	23

Installing BIRT Web Tools Integration	25
Chapter 4 Deploying a BIRT Report to an Application Server ..	27
About application servers	27
About deploying to Tomcat	27
About deploying to other application servers	28
Placing the BIRT report viewer on an application server	28
Installing the BIRT report viewer as a web application	28
Testing the BIRT report viewer installation	30
Using a different context root for the BIRT report viewer	30
Placing the BIRT report viewer in a different location	30
Understanding the BIRT report viewer context parameters	31
Verifying that Apache Tomcat is running BIRT report viewer	32
Placing fonts on the application server	33
Viewing a report using a browser	33
Using connection pooling on Tomcat	34
Setting up a report to use connection pooling	34
Using a jndi.properties file	34
Configuring a JNDI connection object on Tomcat	35
Chapter 5 Using Eclipse BIRT Web Viewer ..	39
Understanding Eclipse BIRT Web Viewer	39
Understanding Web Viewer architecture	44
Web Viewer servlets	45
Web Viewer URL parameters	46
Web Viewer fragments	50
Web Viewer web.xml settings	52
Web Viewer directory structure	54
Web Viewer AJAX operation	55
Using the Web Viewer Deployment wizard	61
Web Viewer tag library	63
Web Viewer RCP deployment	76
Passing a web context object to the Web Viewer	78
Building the Web Viewer	78
Part II Understanding the BIRT Framework ..	81
Chapter 6 Understanding the BIRT Architecture ..	83
Understanding the BIRT integration	83
About the BIRT applications	87
About BIRT Report Designer and BIRT RCP Report Designer	87
About the BIRT Viewer	88
About the BIRT engines and services	88
About the report design engine	89
About the report engine	89
About the generation services	89
About the presentation services	89

About the chart engine	90
About the data services	90
About data services components	90
About the ODA framework	90
About the types of BIRT report items	90
About standard report items	90
About custom report items	91
About chart report items	91
About the Report Object Model (ROM)	91
About the types of BIRT files	91
About report design files	92
About report document files	92
About report library files	92
About report template files	92
About custom Java applications	93
About custom report designers	93
About custom Java report generators	93
About extensions to BIRT	93

Chapter 7 Understanding the Report Object Model 95

About the ROM specification	95
ROM properties	96
ROM slots	97
ROM methods	97
ROM styles	97
About the ROM schema	98
About the rom.def file	98
About the primary ROM elements	101
About the report item elements	102
About the report items	102
Understanding the report item element properties	103
About the data elements	103

Part III Scripting in a Report Design 105

Chapter 8 Using Scripting in a Report Design 107

Overview of BIRT scripting	107
Choosing between JavaScript and Java	107
Using both JavaScript and Java to write event handlers	108
Events overview	108
Engine task processes	108
BIRT Web Viewer	109
BIRT Report Designer	109
BIRT processing phases	110
BIRT event types	110
Parameter events	110
Report design events	111

Data source and data set events	112
ReportItem Events	113
Event order sequence	115
Preparation phase operation	115
Generation phase operation	116
About data source and data set events	118
About data binding	119
About the page break event	120
About chart event order	121
About table and list event order	121
Completion of the generation phase	124
Presentation phase operation	124
Event order summary	125
Chapter 9 Using JavaScript to Write an Event Handler	127
Using BIRT Report Designer to enter a JavaScript event handler	127
Creating and using a global variable	128
Understanding execution phases and processes	129
Using the reportContext object	130
Using getOutputFormat	131
Using reportContext to retrieve the report design handle	132
Passing a variable between processes	133
Using getAppContext	134
Getting information from an HTTP request object	135
Using the this object	135
Using this object methods	136
Using the this object to set the property of a report item	136
Using the row object	138
Getting column information	139
Getting and altering the query string	140
Changing the connection properties of a data source	140
Getting a parameter value	141
Determining method execution sequence	142
Providing the ReportDesign.initialize code	143
Providing code for the methods you want to track	144
Providing the ReportDesign.afterFactory code	144
Tutorial 1: Writing an event handler in JavaScript	144
Task 1: Open the report design	145
Task 2: Create and initialize a counter in the Table.onCreate() method	145
Task 3: Conditionally increment the counter in the Row.onCreate() method	146
Task 4: Display the result using the ReportDesign.afterFactory() method	148
JavaScript event handler examples	149
JavaScript onPrepare examples	149
JavaScript onCreate examples	149
JavaScript onRender examples	151
Calling Java from JavaScript	151
Understanding the Packages object	151
Understanding the importPackage method	151

Using a Java class	152
Placing Java classes	152
Issues with using Java in JavaScript code	153
Calling the method of a class that resides in a plug-in	153
Chapter 10 Using Java to Write an Event Handler	155
Writing a Java event handler class	155
Locating the JAR files that an event handler requires	156
Extending an adapter class	156
Making the Java class visible to BIRT	159
Associating the Java event handler class with a report element	160
BIRT Java interface and class naming conventions	161
Writing a Java event handler	161
Using event handler adapter classes	161
Using event handler interfaces	162
About the Java event handlers for report items	162
Using Java event handlers for the DataSource element	163
Using Java event handlers for the DataSet element	164
Using Java event handlers for the ScriptedDataSource element	164
Using Java event handlers for the ScriptedDataSet element	165
Using Java event handlers for the ReportDesign	166
Understanding the BIRT interfaces	166
About the element design interfaces	167
About the methods for each report element	167
About the IReportElement interface	168
About the element instance interfaces	168
Using the IReportContext interface	169
Using the IColumnMetaData interface	171
Using the IDataSetInstance interface	172
Using the IDataSetRow interface	172
Using the IRowData interface	173
Java event handler example	173
Report level events	173
Report item events	175
Debugging a Java event handler	180
Chapter 11 Working with Chart Event Handlers	183
Chart events overview	183
Understanding when chart events trigger	185
Prepare phase	185
Data binding phase	186
Static data	186
Dynamic data	186
Binding phase script events	187
Building phase	189
Rendering phase	191
Rendering phase script events	193
Rendering blocks	194

Rendering data points	195
Rendering legend items	196
Rendering axes	198
Chart script context	200
Chart instance object	201
Chart instance getter methods	201
Chart instance setter methods	202
Miscellaneous chart instance methods	203
Writing a Java chart event handler	204
Setting up the chart event handler project	204
Chart Java event handler examples	205
Writing a JavaScript chart event handler	208
Using the simplified charting API	211
Getting an instance of a chart item	212
Understanding the sub-interfaces of IChart	213
Chapter 12 Accessing Data Programmatically	217
Using a Scripted Data Source	217
Tutorial 2: Creating and scripting a scripted data source	219
Task 1: Create a new report design	219
Task 2: Create a scripted data source	219
Task 3: Create a scripted data set	220
Task 4: Supply code for the open() and close() methods of the data source	221
Task 5: Supply code for the open() method of the data set	221
Task 6: Define output columns	221
Task 7: Place the columns on the report layout	222
Task 8: Supply code for the fetch() method of the data set	223
Writing the scripted data set in Java	224
Using a Java object to access a data source	226
Performing initialization in the data set open() method	226
Getting a new row of data in the data set fetch() method	226
Cleaning up in the data set close() method	227
Deciding where to locate your Java class	227
Deploying your Java class	227
Using input and output parameters with a scripted data set	227
Creating a web services data source using a custom connection class	228
Part IV Integrating BIRT Functionality into Applications	233
Chapter 13 Understanding the BIRT APIs	235
Package hierarchy diagrams	236
About the BIRT Report Engine API	237
Creating the BIRT ReportEngine instance	238
Using the BIRT Report Engine API	238
EngineConfig class	238
ReportEngine class	239

IReportRunnable interface	239
IReportDocument interface	239
IEngineTask interface	240
IGetParameterDefinitionTask interface	240
IDataExtractionTask interface	240
IRunTask interface	240
IRenderTask interface	241
IRunAndRenderTask interface	241
Report engine class hierarchy	241
Report engine interface hierarchy	242
About the Design Engine API	244
Using the BIRT Design Engine API	244
DesignConfig class	245
DesignEngine class	245
SessionHandle class	246
ModuleHandle class	246
ReportDesignHandle class	247
LibraryHandle class	247
DesignElementHandle class	247
Individual element handle classes	248
Design engine class hierarchy	248
DesignElementHandle hierarchy	250
ReportElementHandle hierarchy	251
ReportItemHandle hierarchy	253
ElementDetailHandle hierarchy	254
StructureHandle hierarchy	255
Design engine interface hierarchy	256
About the BIRT Chart Engine API	257
Using the BIRT Chart Engine API	257
Chart engine class hierarchy	258
chart.aggregate class and interface hierarchy	259
chart.datafeed class and interface hierarchy	259
chart.device class and interface hierarchy	259
chart.event class and interface hierarchy	260
chart.exception class hierarchy	262
chart.factory class and interface hierarchy	262
chart.log class and interface hierarchy	263
chart.model interface hierarchy	263
chart.model.attribute class and interface hierarchy	264
chart.model.component interface hierarchy	268
chart.model.data interface hierarchy	269
chart.model.layout interface hierarchy	270
chart.model.type interface hierarchy	271
chart.render class and interface hierarchy	272
chart.script class and interface hierarchy	273
chart.util class hierarchy	274
Chapter 14 Programming using the BIRT Reporting APIs	275
Building a reporting application	276

About the environment for a reporting application	277
About plug-ins used by the report engine	277
About libraries used by the report engine	277
About required JDBC drivers	278
Modifying a report design with the API	279
Generating reports from an application	279
Setting up the report engine	279
Configuring the BIRT home	279
Configuring the report engine	280
Setting up a stand-alone or WAR file environment	281
Starting the platform	283
Creating the report engine	283
Using the logging environment to debug an application	285
Opening a source for report generation	286
Understanding an IReportRunnable object	286
Understanding an IReportDocument object	287
Accessing a report parameter programmatically	287
Preparing to generate the report	293
Setting the parameter values for running a report design	295
Adding to the report engine's class path	295
Providing an external object to a report design	296
Setting up the rendering options	297
Generating the formatted output programmatically	300
Accessing the formatted report	301
Cancelling a running report task	301
Programming with a report design	302
About BIRT model API capabilities	303
Opening a report design programmatically for editing	304
Configuring the design engine to access a design handle	304
Using an IReportRunnable object to access a design handle	305
Using a report item in a report design	305
Accessing a report item by name	306
Accessing a report item by iterating through a slot	306
Examining a report item programmatically	307
Accessing the properties of a report item	307
Modifying a report item in a report design programmatically	309
Accessing and setting complex properties	310
Understanding property structure objects	311
Adding a report item to a report design programmatically	315
Accessing a data source and data set with the API	315
About data source classes	316
About data set classes	316
Using a data set programmatically	317
Saving a report design programmatically	318
Creating a report design programmatically	319
Chapter 15 Programming using the BIRT Charting API	321
About the chart engine contents	321
About the environment for charting application	322

Configuring the chart engine run-time environment	322
Verifying the environment for charting applications	323
Using the charting API to modify an existing chart	324
Getting a Chart object from the report design	324
Modifying chart properties	325
Modifying plot properties	326
Modifying the legend properties	327
Modifying the series properties	327
Modifying axes properties	328
Adding a series to a chart	328
Adding a chart event handler to a charting application	329
Adding a Java chart event handler to a charting application	329
Adding a JavaScript chart event handler to a charting application	329
Using the charting APIs to create a new chart	330
Creating the chart instance object	331
Setting the properties of the chart instance object	331
Setting the chart color and bounds	331
Setting plot properties	331
Setting legend properties	331
Setting legend line properties	332
Setting axes properties	332
Creating a category series	332
Creating an orthogonal series	333
Defining the orthogonal series data values	334
Setting the orthogonal series properties	334
Setting the series definition properties	335
Adding a series definition to a chart	335
Adding series and queries to the series definitions	336
Using a chart item in a report design	336
Setting the chart type and subtype	336
Creating sample data	337
Getting a design engine element factory object	338
Getting an extended item handle object	338
Setting up the report item as a chart	338
Getting a data set from the report design	339
Binding the chart to the data set	339
Adding the new chart to the report design	339
Saving the report design after adding the chart	339
Putting it all together	340
Using the BIRT charting API in a Java Swing application	345
Understanding the chart programming examples	351
api.data examples	351
DataCharts example	351
GroupOnXSeries example	351
GroupOnYAxis example	352
api.data.autobinding example	352
api.format example	352
api.interactivity examples	352

api.pdf example	353
api.preference example	353
api.processor example	353
api.script examples	353
api.viewer examples	354
Chart3DViewer example	354
SWTChartViewerSelector example	355
builder example	355
report examples	355
MeterChartExample example	356
SalesReport example	356
StockReport example	356
report.design examples	356
report.design.script examples	356
view example	356

Part V Working with the Extension Framework 359

Chapter 16 Building the BIRT Project	361
About building the BIRT project	361
Installing a working version of BIRT	362
Configuring the Eclipse workspace to compile BIRT	362
Downloading and extracting the correct version of the BIRT source code	364
Importing, building, and testing the BIRT project	364
Building new JAR files to display BIRT output	367
Building the viewservlets.jar file	367
Building the chart-viewer.jar file	368
Chapter 17 Extending BIRT	369
Overview of the extension framework	369
Understanding the structure of a BIRT plug-in	370
Understanding an extension point schema definition file	370
Understanding a plug-in manifest file	372
Understanding a plug-in run-time class	375
Working with the Eclipse PDE	377
Understanding plug-in project properties	379
Understanding the Eclipse PDE Workbench	379
Creating the structure of a plug-in extension	381
Creating the plug-in extension content	385
Building a plug-in extension	390
Generating an Ant build script	392
Testing a plug-in extension	392
Deploying the extension plug-in	393
Installing feature updates and managing the Eclipse configuration	394
Creating an update site project	396
Downloading the code for the extension examples	398

Chapter 18 Developing a Report Item Extension	399
Understanding a report item extension	399
Developing the sample report item extension	401
Downloading BIRT source code from the CVS repository	402
Creating a rotated label report item plug-in project	403
Defining the dependencies for the rotated label report item extension	405
Specifying the run-time package for the rotated label report item extension	407
Declaring the report item extension points	408
Creating the plug-in extension content	412
Understanding the rotated label report item extension	418
Understanding RotatedLabelItemFactoryImpl	421
Understanding RotatedLabelUI	421
Understanding RotatedLabelPresentationImpl	421
Understanding GraphicsUtil	422
Understanding RotatedLabelCategoryProviderFactory	425
Understanding RotatedLabelGeneralPage	426
Deploying and testing the rotated label report item plug-in	428
Deploying a report item extension	429
Launching the rotated label report item plug-in	429
Chapter 19 Developing a Report Rendering Extension	433
Understanding a report rendering extension	433
Developing a CSV report rendering extension	434
Creating a CSV report rendering plug-in project	434
Defining the dependencies for the CSV report rendering extension	437
Declaring the emitters extension point	438
Understanding the sample CSV report rendering extension	440
Implementing the emitter interfaces	441
Implementing the content interfaces	442
Understanding the CSV report rendering extension package	443
Understanding CSVReportEmitter	444
Understanding CSVTags	449
Understanding CSVWriter	449
Understanding CSVRenderOption	449
Testing the CSV report rendering plug-in	450
Launching the CSV report rendering plug-in	451
About ExecuteReport class	455
About the report design XML code	457
Developing an XML report rendering extension	462
Creating an XML report rendering plug-in project	463
Defining the dependencies for the XML report rendering extension	464
Declaring the emitters extension point	465
Understanding the sample XML report rendering extension	465
Understanding the XML report rendering extension package	466
Understanding XMLReportEmitter	466
Understanding XMLTags	471
Understanding XMLFileWriter	471
Understanding XMLRenderOption	471

Understanding LoadExportSchema	472
Testing the XML report rendering plug-in	474
Chapter 20 Developing an ODA Extension	477
Understanding an ODA extension	478
Developing the CSV ODA driver extensions	479
About the CSV ODA plug-ins	480
Downloading BIRT source code from the CVS repository	480
Implementing the CSV ODA driver plug-in	481
Understanding the ODA data source extension points	486
Understanding dataSource extension point properties	486
Understanding ConnectionProfile properties	489
Understanding the dependencies for the CSV ODA driver extension	489
Understanding the sample CSV ODA driver extension	491
Implementing the DTP ODA interfaces	491
Understanding the CSV ODA extension package	493
Understanding Driver	494
Understanding Connection	494
Understanding Query	494
Understanding ResultSet	497
Understanding ResultSetMetaData	498
Understanding DataSetMetaData	499
Understanding Messages	499
Understanding CommonConstants	500
Developing the CSV ODA UI extension	500
Creating the CSV ODA UI plug-in project	501
Understanding the ODA data source UI extension points	506
Understanding the ConnectionProfile extension point	506
Understanding the propertyPages extension point	507
Understanding the dataSource extension point	508
Understanding the sample CSV ODA UI extension	509
Implementing the ODA data source and data set wizards	509
Understanding the org.eclipse.birt.report.data.oda.csv.ui.impl package	510
Understanding the org.eclipse.birt.report.data.oda.csv.ui.wizards package	510
Understanding Constants	511
Understanding CSVFilePropertyPage	511
Understanding CSVFileSelectionPageHelper	512
Understanding CSVFileSelectionWizardPage	514
Understanding FileSelectionWizardPage	515
Testing the CSV ODA UI plug-in	519
Developing a Hibernate ODA extension	524
Creating the Hibernate ODA driver plug-in project	526
Understanding the sample Hibernate ODA driver extension	534
Understanding HibernateDriver	535
Understanding Connection	536
Understanding DataSetMetaData	538
Understanding Statement	538
Understanding ResultSet	542

Understanding HibernateUtil	544
Building the Hibernate ODA driver plug-in	547
Developing the Hibernate ODA UI extension	548
Understanding the sample Hibernate ODA UI extension	554
Understanding HibernatePageHelper	555
Understanding HibernateDataSourceWizard	558
Understanding HibernatePropertyPage	558
Understanding HibernateHqlSelectionPage	558
Building the Hibernate ODA UI plug-in	564
Testing the Hibernate ODA UI plug-in	566
Chapter 21 Developing a Fragment	571
Understanding a fragment	571
Developing the sample fragment	572
Creating a fragment project	573
Understanding the sample fragment	576
Building, deploying, and testing a fragment	577
Glossary	583
Index	645

This page intentionally left blank

It is a common misconception that Eclipse projects are focused on simply providing great tools for developers. Actually, the expectations are far greater. Each Eclipse project is expected to provide both frameworks and extensible, exemplary tools. As anyone who has ever tried to write software with reuse and extensibility in mind knows, that is far more difficult than simply writing a tool.

“Exemplary” is one of those handy English words with two meanings. Both are intended in its use above. Eclipse projects are expected to provide tools that are exemplary in the sense that they provide an example of the use of the underlying frameworks. Eclipse tools are also intended to be exemplary in the sense that they are good and provide immediate utility to the developers who use them.

Since its inception, the BIRT project has worked hard to create both reusable frameworks and extensible tools. This book focuses primarily on how to extend BIRT and how to use BIRT in your own applications and products. As such, it illustrates BIRT’s increasing maturity and value as an embedded reporting solution.

As Executive Director of the Eclipse Foundation, I’m pleased with the tremendous progress the BIRT team has made since the project’s inception in September of 2004, and I’m equally pleased with the vibrant community that has already grown up around it. As you work with BIRT and the capabilities that are described in this book, I’d encourage you to communicate your successes back to the community, and perhaps consider contributing any interesting extensions you develop. The BIRT web site can be found here:

<http://www.eclipse.org/birt>

It includes pointers to the BIRT newsgroup, where you can communicate and share your results with other BIRT developers, and pointers to the Eclipse installation of Bugzilla, where you can contribute your extensions. If you like BIRT—and I am sure this book will help you learn to love it—please participate and contribute. After all, it is the strength of its community that is the true measure of any open source project’s success.

Mike Milinkovich
Executive Director, Eclipse Foundation

This page intentionally left blank

About this book

The second of a two-book series on business intelligence and reporting technology, *Integrating and Extending BIRT*, introduces programmers to BIRT architecture and the reporting framework. Its companion book, *BIRT: A Field Guide to Reporting*, shows report developers how to create reports using the graphical tools of BIRT Report Designer. Built on the open-source Eclipse platform, BIRT is a powerful reporting system that provides an end-to-end solution, from creating and deploying reports to integrating report capabilities in enterprise applications.

BIRT technology makes it possible for a programmer to build a customized report using scripting and BIRT APIs. This book informs report developers about how to write scripts that:

- Customize the report-generation process
- Incorporate complex business logic in reports

This book also informs application developers about how to:

- Deploy reports
- Integrate reporting capabilities into other applications
- Extend BIRT functionality

A programmer can extend the BIRT framework by creating a new plug-in using the Eclipse Plug-in Development Environment (PDE). This book provides extensive examples on how to build plug-ins to extend the features of the BIRT framework. The source code for these examples is available for download.

The topics discussed in this book include:

- Installing and deploying BIRT
- Deploying a BIRT report to an application server
- Understanding BIRT architecture

- Scripting in a BIRT report design
- Integrating BIRT functionality into applications
- Working with the BIRT extension framework

This revised BIRT 2.2.1 edition adds the following new content:

- Updated architectural diagrams
- Expanded scripting examples
- Tag library descriptions
- In-depth description of BIRT Web Viewer
- Configuring BIRT to use a JNDI connection
- XML report rendering plug-in example
- Fragment plug-in localization example
- Open Data Access (ODA) plug-in example implementing the new Data Tools Platform (DTP) design and run-time wizards

Who should read this book

This book is intended for people who have a programming background. These readers can be categorized as:

- Embedders and integrators

These individuals work with the software to integrate it into their current application infrastructure.

- Extenders

These individuals leverage APIs and other extension points to add capability or to establish new interoperability between currently disparate components or services.

To write scripts in report design, you need knowledge of JavaScript or Java. More advanced tasks, such as extending BIRT's functionality, require Java development experience and familiarity with the Eclipse platform.

Contents of this book

This book is divided into several parts. The following sections describe the contents of each of the parts.

Part I, Installing and Deploying BIRT

Part I introduces the currently available BIRT reporting packages, the prerequisites for installation, and the steps to install and update the packages. Part I includes the following chapters:

- *Chapter 1. Prerequisites for BIRT.* BIRT provides a number of separate packages as downloadable archive (.zip) files on the Eclipse web site. Some of the packages are stand-alone modules, others require an existing Eclipse environment, and still others provide additional functionality to report developers and application developers. This chapter describes the prerequisites for each of the available packages.
- *Chapter 2. Installing a BIRT Report Designer.* BIRT provides two report designers as separate packages, which are downloadable archive (.zip) files on the Eclipse web site. This chapter describes the steps required to install each of the available report designers.
- *Chapter 3. Installing Other BIRT Packages.* This chapter describes the steps required to install and update each of the available packages.
- *Chapter 4. Deploying a BIRT Report to an Application Server.* This chapter introduces the distribution of reports through an application server such as Apache Tomcat, IBM WebSphere, or BEA WebLogic. The instructions in the chapter provide detailed guidance about deploying a BIRT report to Apache Tomcat version 5.5. From those instructions, a developer can infer how to deploy to other versions.
- *Chapter 5. Using Eclipse BIRT Web Viewer.* This chapter describes how to use the sample Eclipse BIRT Web Viewer to generate and run a report. The Web Viewer is an AJAX-based, J2EE application that illustrates how to use the BIRT engine to generate and render report content. The Web Viewer uses the prototype JavaScript framework to implement AJAX-based communications within the Web Viewer and to the Web Viewer servlets. You can use the plug-in format of the Web Viewer in existing Eclipse-based architectures, such as RCP applications.

Part II, Understanding the BIRT Framework

Part II introduces the BIRT architecture and the Report Object Model (ROM) and provides background information that will help programmers design or modify reports programmatically, instead of using the graphical tools in BIRT Report Designer. Part II includes the following chapters:

- *Chapter 6. Understanding the BIRT Architecture.* This chapter provides an architectural overview of BIRT and its components, including the relationships among the BIRT components and BIRT's relationship to Eclipse and Eclipse frameworks. Architectural diagrams illustrate and clarify the relationships and workflow of the components. The chapter also provides brief overviews of all the major BIRT components.
- *Chapter 7. Understanding the Report Object Model.* This chapter provides an overview of the BIRT ROM. ROM is a specification for a set of XML

elements that define both the visual and non-visual elements that comprise a report design. The ROM specification includes the properties and methods of those elements, and the relationships among the elements.

Part III, Scripting in a Report Design

Part III describes how a report developer can customize and enhance a BIRT report by writing event handler scripts in either Java or JavaScript. Part III includes the following chapters:

- *Chapter 8. Using Scripting in a Report Design.* This chapter introduces the writing of a BIRT event handler script in either Java or JavaScript, including the advantages and disadvantages of using one language over the other. BIRT event handlers are associated with data sets, data sources, and report items. BIRT fires specific events at specific times in the processing of a report. This chapter identifies the events that BIRT fires and describes the event firing sequence.
- *Chapter 9. Using JavaScript to Write an Event Handler.* This chapter discusses the coding environment and coding considerations for writing a BIRT event handler in JavaScript. This chapter describes several BIRT JavaScript objects that a developer can use to get and set properties that affect the final report. The BIRT JavaScript coding environment offers a pop-up list of properties and functions available in an event handler. A JavaScript event handler can also use Java classes. This chapter includes a tutorial that describes the process of creating a JavaScript event handler.
- *Chapter 10. Using Java to Write an Event Handler.* This chapter discusses how to write a BIRT event handler in Java. BIRT provides Java adapter classes that assist the developer in the creation of Java event handlers. The report developer uses the property editor of the BIRT Report Designer to associate a Java event handler class with the appropriate report element. This chapter contains a tutorial that steps through the Java event handler development and deployment process. This chapter also describes the event handler methods and their parameters.
- *Chapter 11. Working with Chart Event Handlers.* This chapter describes the BIRT event handler model for the Chart Engine. The model is similar to the model for standard BIRT report elements and supports both the Java and JavaScript environments. This chapter provides details on both environments. The Chart Engine also supports this event model when used outside of BIRT.
- *Chapter 12. Accessing Data Programmatically.* This chapter describes how to access a data source using JavaScript code. A data source that you access using JavaScript is called a scripted data source. With a scripted data source, you can access objects other than an SQL, XML, or text file data source. A scripted data source can be an EJB, an XML stream, a Hibernate object, or any other Java object that retrieves data.

Part IV, Integrating BIRT Functionality into Applications

Part IV describes the public APIs that are available to Java developers, except the extension APIs. Part IV includes the following chapters:

- *Chapter 13. Understanding the BIRT APIs.* This chapter introduces BIRT's public API, which are the classes and interfaces in three package hierarchies:
 - The report engine API, in the org.eclipse.birt.report.engine.api hierarchy, supports developers of custom report generators.
 - The design engine API, in the org.eclipse.birt.report.engine.api hierarchy, supports the development of custom report designs.
 - The chart engine API, in the org.eclipse.birt.chart hierarchy, is used to develop a custom chart generator.
- *Chapter 14. Programming using the BIRT Reporting APIs.* This chapter describes the fundamental requirements of a reporting application and lists the BIRT API classes and interfaces that are used to create a reporting application. This chapter describes the tasks that are required of a reporting application and provides an overview of how to build a reporting application. The org.eclipse.birt.report.engine.api package supports the process of generating a report from a report design. The org.eclipse.bert.report.model.api package supports creating new report designs and modifying existing report designs.
- *Chapter 15. Programming using the BIRT Charting API.* This chapter describes the requirements of a charting application, either in a stand-alone environment or as part of a reporting application. The org.eclipse.birt.chart hierarchy of packages provides the charting functionality in BIRT. By describing the fundamental tasks required of charting applications, this chapter introduces the API classes and interfaces that are used to create a chart. This chapter also describes the chart programming examples in the chart examples plug-in.

Part V, Working with the Extension Framework

Part V shows Java programmers how to add new functionality to the BIRT framework. By building on the Eclipse platform, BIRT provides an extension mechanism that is familiar to developers of Eclipse plug-ins. This part also provides information about how to build the BIRT project for developers who need access to the complete BIRT open source code base. Part V includes the following chapters:

- *Chapter 16. Building the BIRT Project.* This chapter explains how to download BIRT 2.2.1 source code and build the BIRT project for development. This chapter describes how to configure an Eclipse workspace, download BIRT and Data Tools Platform (DTP) source code from the Eclipse Concurrent Versions System (CVS) repository, and build the BIRT report and web viewers.

- *Chapter 17. Extending BIRT.* This chapter provides an overview of the BIRT extension framework and describes how to use the Eclipse Plug-in Development Environment (PDE) and the BIRT extension points to create, build, and deploy a BIRT extension.
- *Chapter 18. Developing a Report Item Extension.* This chapter describes how to develop a report item extension. The rotated text extension example is a plug-in that renders the text of a report item as an image. The extension rotates the image in the report design to display the text at a specified angle. This chapter describes how to build the rotated text report item plug-in and add the report item to the BIRT Report Designer using the defined extension points.
- *Chapter 19. Developing a Report Rendering Extension.* This chapter describes how to develop a report rendering extension using the Eclipse PDE with sample CSV and XML report rendering extensions as the examples. The chapter describes how to extend the emitter interfaces using the defined extension points to build and deploy a customized report rendering plug-in that runs in the BIRT Report Engine environment.
- *Chapter 20. Developing an ODA Extension.* This chapter describes how to develop several types of DTP ODA extensions. The CSV ODA driver example is a plug-in that reads data from a CSV file. The Hibernate ODA driver example uses Hibernate Query Language (HQL) to provide a SQL-transparent extension that makes the ODA extension portable to all relational databases. This chapter shows how to develop an ODA extension to the BIRT Report Designer 2.2.1 user interface that allows a report designer to select an extended ODA driver.
- *Chapter 21. Developing a Fragment.* This chapter describes how to build a fragment. The BIRT Report Engine environment supports plug-in fragments. A plug-in fragment is a separately loaded package that adds functionality to an existing plug-in, such as a specific language feature in a National Language Support (NLS) localization application. The example in this chapter creates a Java resource bundle that adds translations to the messages defined in the messages.properties files for the org.eclipse.birt.report.viewer plug-in.

The *Glossary* contains a glossary of terms that are useful to understanding all parts of the book.

Typographical conventions

Table P-1 describes the typographical conventions that are used in this book.

Table P-1 Typographical conventions

Item	Convention	Example
Code examples	Monospace font	<code>StringName = "M. Barajas";</code>

Table P-1 Typographical conventions (*continued*)

Item	Convention	Example
File names	Initial capital letter, except where file names are case-sensitive	SimpleReport.rptdesign
Key combination	A + sign between keys means to press both keys at the same time	Ctrl+Shift
Menu items	Capitalized, no bold	File
Submenu items	Separated from the main menu item with a small arrow	File ➔ New
User input	Monospace font	2008
User input in Java code	Monospace font italics	chkjava.exe <i>cab_name.cab</i>

Syntax conventions

Table P-2 describes the symbols that are used to present syntax.

Table P-2 Syntax conventions

Symbol	Description	Example
[]	Optional item	int count [= <value>];
	Array subscript	matrix[]
< >	Argument that you must supply	<expression to format>
	Delimiter in XML	<xsd:sequence>
{ }	Groups two or more mutually exclusive options or arguments when used with a pipe	{TEXT_ALIGN_LEFT TEXT_ALIGN_RIGHT}
	Defines array contents	{0, 1, 2, 3}
	Delimiter of code block	if (itemHandle == null) { // create a new handle }
	Separates mutually exclusive options or arguments in a group	[public protected private] <data type> <variable name>;
	Java bitwise OR operator	int newflags = flags 4

This page intentionally left blank

A c k n o w l e d g m e n t s

John Arthorne and Chris Laffra observed, "It takes a village to write a book on Eclipse." In the case of the BIRT books, it continues to take a virtual village in four countries to create these two books. Our contributors, reviewers, Addison-Wesley editorial, marketing, and production staff, printers, and proofreaders are collaborating by every electronic means currently available to produce the major revisions to these two books. In addition, we want to acknowledge the worldwide community of Java programmers who have completed over three million downloads of the multiple versions of the software. Their enthusiastic reception to the software creates an opportunity for us to write about it.

We want to thank Greg Doench, our acquisitions editor, who asked us to write a book about BIRT and has been supportive and enthusiastic about our success. Of course, we want to acknowledge the staff at Addison-Wesley who worked on the first edition and this revision. In particular, we would like to acknowledge John Fuller, Michelle Housley, Mary Kate Murray, Julie Nahil, Stephane Nakib, Sandra Schroeder, Beth Wickenhiser, and Lara Wysong. We also want to thank Mike Milinkovich at the Eclipse Foundation and Mark Coggins at Actuate Corporation for providing the forewords for the books.

We particularly want to acknowledge the many, many managers, designers, and programmers too numerous to name who have worked diligently to produce, milestone by milestone, the significant upgrades to BIRT, giving us a reason for these two books. You know who you are and know how much we value your efforts. The following engineers have been of particular assistance to the authors: Linda Chan, Wenbin He, Petter Ivmark, Rima Kanguri, Nina Li, Wenfeng Li, Yu Li, Jianqiang Luo, Zhiqiang Qian, Kai Shen, Aniruddha Shevade, Pierre Tessier, Krishna Venkatraman, Mingxia Wu, Gary Xue, Jun Zhai, and Lin Zhu. We want to recognize the important contribution of David Michonneau in the area of charting. Yasuo Doshiro worked closely with the authors to develop the material on Using Fragments, which provides suggestions for the practical application of BIRT technology to the challenges of translation and localization. Doshiro manages BIRT translation, assists with test plans and execution for IBM, and participates in the Eclipse Babel project. In addition, we want to acknowledge the support and significant contribution that was provided by Paul Rogers.

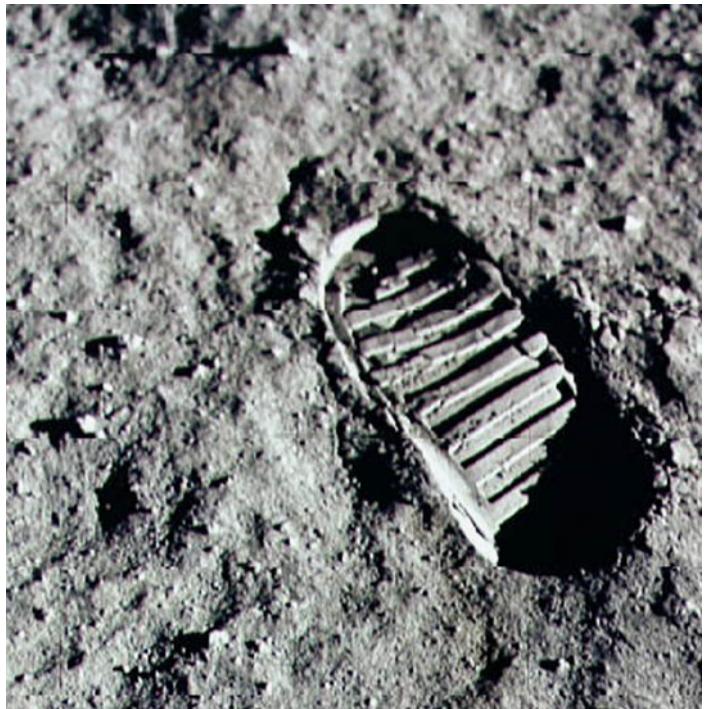
Dan Melcher's and Daniel O'Connell's insights into the techniques for building reusable components that can be applied to building internationalized reports. Working examples are to be found at <http://reusablereporting.blogspot.com/>

Creating this book would not have been possible without the constant support of the members of the Developer Communications team at Actuate Corporation. Many of them and their families sacrificed long personal hours to take on additional tasks so that members of the team of authors could create this material. In particular, we wish to express our appreciation to Terry Ryan who pulled together the terminology in the glossary that accompanies each of the books. In addition, Mary Adler, Frances Buran, Bruce Gardner, Kris Hahn, Mike Hovermale, Melia Kenny, Cheryl Koyano, Madalina Lungulescu, Liesbeth Matthieu, Audrey Meinertzhangen, James Monaghan, Jon Nadelberg, Lois Olson, Jeff Wachhorst, and Forest White all contributed to the success of the books.

Actuate's active student intern program under the Executive Sponsorship of Dan Gaudreau, Chief Financial Officer, made it possible for Hamid Foroozandeh, Arvind Kanna, Arash Khaziri, Maziar Jamalian, Gene Sher, C. J. Walter-Hague, and Samantha Weizel to support the project in Developer Communications while actively engaged in pursuing undergraduate and graduate degrees in accounting, business and information science, economics, physics, and technical writing at five different universities in California, New York, and Ontario, Canada.

I

Installing and Deploying BIRT



This page intentionally left blank

1

Prerequisites for BIRT

There are two designer applications that you can use to create BIRT reports:

- **BIRT Report Designer**

Requires multiple Eclipse platform components and a Java Development Kit (JDK). This designer version provides all the functionality of BIRT RCP Report Designer, plus support for writing Java code. BIRT Report Designer is useful for a report designer who wants to modify the underlying Java or JavaScript code that BIRT uses to create a report.

- **BIRT RCP Report Designer**

A stand-alone module for report developers who do not have programming experience. This designer is based on the Eclipse Rich Client Platform (RCP). BIRT RCP Report Designer is a stand-alone package, which requires only a Java JDK as an additional resource.

This chapter describes the requirements for installing these BIRT Report Designers and related components.

Downloading Eclipse BIRT components

You can download BIRT 2.2.1 from the following location:

[http://download.eclipse.org/birt/downloads/build.php?
build=R-R1-2_2_1-200710010630](http://download.eclipse.org/birt/downloads/build.php?build=R-R1-2_2_1-200710010630)

The download page contains a mix of packages. Some packages contain stand-alone modules while others require an existing Eclipse environment. Some packages provide extra functionality for BIRT report and application developers.

The BIRT 2.2.1 download site contains the following packages:

- Report Designer Full Install (All-in-One) for Windows

Contains all the components necessary to run BIRT except the required Java 1.5 JDK. This all-in-one installation is the easiest way to install BIRT. In addition to the complete BIRT Reporting Framework, this package includes the following Eclipse components:

 - Software Development Kit (SDK)
 - Graphical Editing Framework (GEF)
 - Modeling Framework (EMF)
 - Web Tools Platform (WTP)
 - Axis plug-in
- Report Designer Full Install (All-in-One) for Linux

Contains the same components as BIRT Report Designer Full Eclipse Install for Windows.
- Report Designer

Contains only the BIRT Report Designer plug-ins for installing in an existing Eclipse Integrated Development Environment (IDE).
- RCP Report Designer

Contains a simplified report designer that uses Eclipse Rich Client Platform (RCP) technology without the additional perspectives available in the standard Eclipse IDE.
- BIRT Data Tools Platform (DTP) Integration

Contains the minimal set of Eclipse Data Tools Platform (DTP) plug-ins that BIRT requires when installing the Report Designer framework package.
- BIRT SDK

Contains the source code for the BIRT plug-ins and examples.
- Report Engine

Contains the report engine that you can install in a J2EE application server to run BIRT reports in a viewer.
- Chart Engine

Contains the chart engine plug-ins for the Eclipse environment, run-time JAR files for Java applications, a WAR file for Web deployment, and SDK plug-ins, including source code, examples, documentation, and a web tools extension. The chart engine is a stand-alone library that supports adding a chart to a Java application that runs independent of a BIRT report.
- BIRT Web Tools Integration

- Contains the plug-ins required to use the BIRT Web Project Wizard in a Web Tools Project, including the source code.
- **BIRT Source Code**
Contains the BIRT source code for a specific build. All source code is in a plug-in format ready to import into a workspace to build BIRT. These plug-ins are the required libraries for a standard BIRT installation. Additional libraries may be necessary. For example, this package does not include the Data Tools Platform (DTP) source code.
- **BIRT Samples**
Contains sample reports and charts, plus application examples that use the Chart, Report Engine, and Design Engine APIs.
- **BIRT Demo Database**
Contains the package for defining and loading the demonstration database into Apache Derby and MySQL, including SQL and data files. The demonstration database package is a convenient way to install the Classic Models database schema and data in the Apache Derby and MySQL systems. The package does not include any BIRT software. The Report Designer and the RCP Report Designer packages include the demonstration database for Apache Derby.
The demonstration database supports the following Apache and MySQL versions:
 - Apache Derby version 5.1 or higher
 - MySQL Connector/J version 3.1 or MySQL client version 4.x

BIRT Report Designer software requirements

Because BIRT is a Java-based platform, installing a required component typically involves only unpacking an archive. Unpacking the all-in-one archive places the components in the required locations in the installation path.

Most BIRT components are packed in archives that have an eclipse directory at the top level. As a result, you follow the same unpacking procedure for most modules.

If you install BIRT components from separate packages, examine the archive structure carefully before you unpack an archive to confirm that you are unpacking to the correct directory. A common installation mistake for a new BIRT user is to unpack the archives in the wrong directory.

BIRT Report Designer requires the software shown in Table 1-1. Table 1-1 lists the software versions required for developing report designs using BIRT

Report Designer 2.2.1. You cannot use other versions of these listed components.

Table 1-1 Supported configuration for BIRT 2.2.1

Component	Required version
Eclipse Platform	3.3.1
Data Tools Platform (DTP)	1.5.1
Eclipse Modeling Framework (EMF)	2.3.1
Graphical Editing Framework (GEF)	3.3.1
Web Tools Platform (WTP)	2.0.1
Java Development Kit (JDK)	1.5

The following section provides additional information about these software components:

- **Eclipse Platform**

The Eclipse SDK is an archive file that you extract to your hard drive. The installation of Eclipse is complete once you extract this archive. The result is the creation of a directory named Eclipse.

You must specify where to place the eclipse directory on your hard drive. You can extract the Eclipse archive to any location that you prefer. A typical location for Eclipse is the root directory of the C drive. If you specify this directory, the result of installing Eclipse is the following directory:

c:/eclipse

- **DTP**

The Data Tools Platform (DTP) is a collection of Eclipse plug-ins that BIRT uses to access data sources and retrieve data from those data sources. If you do not need the full functionality of DTP, you can use the BIRT DTP Integration package instead of the full DTP platform.

- **EMF**

The Eclipse Modeling Framework (EMF) is a collection of Eclipse plug-ins that BIRT charts use. The required EMF download includes the Service Data Objects (SDO) component, which is a graph-structured data object that supports applying changes to a graph back to the data source.

- **GEF**

Graphical Editing Framework (GEF) is an Eclipse plug-in that the BIRT Report Designer user interface requires. This framework provides a rich, consistent, graphical editing environment for an application running on the Eclipse Platform.

- WTP
Eclipse Web Tools Platform (WTP) is a set of Eclipse plug-ins that support deploying the BIRT report viewer to an application server. The package includes source and graphical editors, tools, wizards, and APIs that support deploying, running, and testing.
- Java Development Kit (JDK)
Eclipse recommends using the Java J2SE JDK 1.5 release with BIRT 2.2.1. JDK 1.5 is also known as JDK 5.0.
The Java J2SE JDK 1.5 download is currently available at the following URL:
http://java.sun.com/javase/downloads/index_jdk5.jsp

About types of BIRT builds

The Eclipse BIRT download site makes several types of builds available for BIRT. The following list describes these builds:

- Release build
A production build that passes the complete test suite for all components and features. Use the release build to develop applications.
- Milestone build
A development build that provides access to newly completed features. The build is stable, but it is not production quality. Use this type of build to preview new features and develop future reporting applications that depend on those features.
- Stable build
A development build that is stable, but passes a reduced test suite. New features are in an intermediate stage of development. Use a stable build to preview new features.
- Nightly build
The Eclipse BIRT development team builds BIRT every night. As an open source project, these builds are available to anyone. These builds are unlikely to be useful to a report developer.
If a certain feature that you require does not work, you can provide feedback to the development team by filing a bug report. Later, you can download a new build to confirm that the fix solves the problem that you reported.

This page intentionally left blank

2

Installing a BIRT Report Designer

Installing BIRT Report Designer adds a report design perspective to the Eclipse Integrated Development Environment (IDE). You install BIRT Report Designer by downloading an archive file from the Eclipse web site and extracting it in your existing Eclipse environment. The following examples describe how to install BIRT Release 2.2.1.

Installing BIRT Report Designer Full Eclipse Install

If you are new to Eclipse and BIRT, you can download and install BIRT Report Designer Full Eclipse Install (All-in-One) package to start developing and designing BIRT reports immediately. This package includes the Eclipse Integrated Development Environment (IDE), BIRT Report Designer, and all other required Eclipse components. You must also download and install Java JDK 1.5.

Complete the following procedure to download this installation package on a Windows or Linux system.

How to install BIRT Report Designer Full Eclipse Install

- 1 Using your browser, navigate to the main BIRT web page at:

<http://www.eclipse.org/birt>

- 2 From BIRT Project, choose Download.
- 3 From BIRT Report Downloads—More Downloads, choose Recent Builds Page.

This page shows all the downloadable BIRT projects.

- 4 From BIRT Downloads—Latest Releases, choose 2_2_1.

BIRT Downloads for build 2.2.1 appears.

- 5 On BIRT Downloads for build 2.2.1, perform one of the following tasks:

- If you are using Windows, choose the following archive file in Report Designer Full Eclipse Install:

`birt-report-designer-all-in-one-2_2_1.zip`

- If you are using Linux, choose the following archive file in Report Designer Full Eclipse Install for Linux:

`birt-report-designer-all-in-one-linux-gtk-2_2_1.tar.gz`

Eclipse downloads appears. This page shows all the sites that provide this archive file.

- 6 Choose the download site that is closest to your location.

The BIRT Report Designer all-in-one archive file downloads to your system.

- 7 Extract the archive file to a hard drive location that you specify.

The extraction creates a directory named `eclipse` at the location that you specify.

To test the BIRT Report Designer installation, start Eclipse, then start BIRT Report Designer as described in the following procedure. BIRT Report Designer is a perspective within Eclipse.

How to test the BIRT Report Designer installation

- 1 Start Eclipse.

- 2 From the Eclipse window menu, choose Open Perspective→Report Design. If Report Design does not appear in the Open Perspective window, choose Other. A list of perspectives appears. Choose Report Design.

Eclipse displays the BIRT Report Designer perspective.

Installing BIRT RCP Report Designer

BIRT RCP Report Designer is a stand-alone report design application that enables report developers to produce reports in both web and PDF formats. This application uses the Eclipse Rich Client Platform (RCP) to provide a report design environment that is less complex than the full Eclipse platform. If you need the project-based environment that the full Eclipse platform provides, install BIRT Report Designer instead. BIRT RCP Report Designer runs only on Windows.

You install BIRT RCP Report Designer by downloading and extracting an archive file. The following examples use Release 2.2.1.

Complete the following procedure to download and install BIRT RCP Report Designer on a Windows system.

How to install BIRT RCP Report Designer

- 1** Using your browser, navigate to the main BIRT web page at:

`http://www.eclipse.org/birt`

- 2** From BIRT Project, choose Download.

- 3** From BIRT Report Downloads—More Downloads, choose Recent Builds Page.

This page shows all the downloadable BIRT projects.

- 4** From BIRT Downloads—Latest Releases, choose 2_2_1.

BIRT Downloads for build 2.2.1 appears.

- 5** In RCP Report Designer, choose the following archive file:

`birt-rcp-report-designer-2_2_1.zip`

Eclipse downloads appears. This page shows all the sites that provide this download file.

- 6** Choose the download site that is closest to your location.

The BIRT RCP Report Designer archive downloads to your system.

- 7** Extract the archive file to a hard drive location that you specify.

The extraction creates a directory named birt-rcp-report-designer-2_2_1 at the location that you specify.

To test the installation, start BIRT RCP Report Designer as described in the following procedure.

How to test the BIRT RCP Report Designer installation

- 1** Navigate to the birt-rcp-report-designer-2_2_1 directory.

- 2** To run BIRT RCP Report Designer, double-click BIRT.exe. BIRT RCP Report Designer appears.

Troubleshooting installation problems

Installing a BIRT report designer is a straightforward task. If you extract the archive file to the appropriate location and the required supporting files are also available in the expected location, your BIRT report designer will work. One of the first steps in troubleshooting an installation problem is confirming that all files are in the correct location.

Verify that the /eclipse/plugins directory contains JAR files whose names begin with org.eclipse.birt, org.eclipse.emf, and org.eclipse.gef. The following sections describe troubleshooting steps that resolve two common installation errors.

Avoiding cache conflicts after you install a BIRT report designer

Eclipse caches information about plug-ins for faster start-up. After you install or upgrade BIRT Report Designer or BIRT RCP Report Designer, using a cached copy of certain pages can lead to errors or missing functionality. The symptoms of this problem include the following conditions:

- The Report Design perspective does not appear in Eclipse.
- You receive a message that an error occurred when you open a report or use the Report Design perspective.
- JDBC drivers that you installed do not appear in the driver manager.

The solution is to remove the cached information. The recommended practice is to start either Eclipse or BIRT RCP Report Designer from the command line with the -clean option.

To start Eclipse, use the following command:

```
eclipse.exe -clean
```

To start BIRT RCP Report Designer, use the following command:

```
BIRT.exe -clean
```

Specifying a Java Virtual Machine when starting BIRT report designer

You can specify which Java Virtual Machine (JVM) to use when you start a BIRT report designer. This specification is important, particularly for users on Linux, when path and permission problems prevent the report designer from locating an appropriate JVM to use. A quick way to overcome such problems is by specifying explicitly which JVM to use when you start the BIRT report designer.

On Windows and Linux systems, you can either start a BIRT report designer from the command line or create a command file or shell script that calls the appropriate executable file with the JVM path. The example in this section uses BIRT Report Designer on a Windows system.

How to specify which JVM to use when you start a BIRT report designer

On the command line, type a command similar to:

```
eclipse.exe -vm $JAVA_HOME/jdk1.5/bin/java.exe
```

Installing a language pack

All BIRT user interface components and messages are internationalized through the use of properties files. BIRT uses English as the default language, but supports other languages by installing a language pack that contains the required properties files. There are four language packs for BIRT packages.

Each language pack contains support for a specific set of languages. The names of the language packs are identical for each product, although the archive file names differ. The following list describes these language packs and the languages that they support:

- NLpack1
 - Supports German, Spanish, French, Italian, Japanese, Korean, Brazilian Portuguese, traditional Chinese, and simplified Chinese.
- NLpack2
 - Supports Czech, Hungarian, Polish, and Russian.
- NLpack2a
 - Supports Danish, Dutch, Finnish, Greek, Norwegian, Portuguese, Swedish, and Turkish.
- NLpackBidi
 - Supports Arabic and Hebrew. Hebrew is available only for the Eclipse, GEF, and EMF run times.

The following instructions explain how to download and install a language pack for BIRT 2.2.1 on Windows.

How to download and install a language pack

To download and install a language pack, perform the following steps:

- 1 Using your browser, navigate to the main BIRT web page at:
<http://www.eclipse.org/birt>
- 2 From BIRT Project, choose Download.
- 3 From BIRT Report Downloads—More Downloads, choose Recent Builds Page.
This page shows all the downloadable BIRT projects.
- 4 From BIRT Downloads—Latest Releases, choose 2_2_1.
BIRT Downloads for build 2.2.1 appears.
- 5 In the Build Documentation section, choose Language Packs.
- 6 Download the language pack for the product and language that you need.

- 7** Extract the language pack archive file into the directory above the Eclipse directory.

For example, if C:/eclipse is your Eclipse directory, extract the language pack into C:/.

- 8** Start Eclipse and choose Window->Preferences->Report Design->Preview.
- 9** Select the language of choice from the drop-down list in Choose your locale.
- 10** Restart Eclipse.

If Windows is not running under the locale you need for BIRT, start Eclipse using the -nl <locale> command line option, where <locale> is a standard Java locale code, such as es_ES for Spanish as spoken in Spain. Sun Microsystems provides a list of locale codes at the following URL:

<http://java.sun.com/j2se/1.5.0/docs/guide/intl/locale.doc.html>

Eclipse remembers the locale you specify on the command line. On subsequent launches of Eclipse, the locale is set to the most recent locale setting. To revert to a previous locale, launch Eclipse using the -nl command line option for the locale to which you want to revert.

Updating a BIRT Report Designer installation

Because BIRT Report Designer is a Java-based application, updating an installation typically requires replacing the relevant files. Eclipse supports the update process for BIRT Report Designer by providing the Update Manager. BIRT RCP Report Designer is a stand-alone product, so you must replace the existing version with a newer version.

This section describes the steps required to update the following BIRT packages:

- Report Designer
- RCP Report Designer

You can use the Eclipse Update Manager to find and install newer major releases of BIRT Report Designer.

How to update a BIRT Report Designer installation using the Update Manager

- 1** In Eclipse, choose Help->Software Updates->Find and Install. Feature Updates appears.
- 2** Select Search for updates of currently installed features, and choose Finish. The Update Manager displays a list of update sites. Choose one to continue. Search Results appears and displays any updates that are available.
- 3** Select a feature to update, then choose Next.

- 4** On Feature License, select I accept the terms in the license agreement. To continue installing the update, choose Next.
- 5** On Installation, choose Finish to download and install the selected updates.

To install a milestone release or other pre-release version, use the manual update instructions.

How to update BIRT Report Designer manually

- 1** Back up the workspace directory if it is in the eclipse directory structure.
- 2** To remove the BIRT files, use one of the following techniques:
 - To prepare for a new all-in-one installation, remove the entire eclipse directory.
 - To prepare for only a BIRT Report Designer installation, remove only the BIRT components.
 - 1** Navigate to the eclipse\features directory.
 - 2** Delete all JAR files and subdirectories with birt in their names.
 - 3** Navigate to the eclipse\plugins directory.
 - 4** Delete all JAR files and subdirectories with birt in their names.
- 3** Download and install BIRT Report Designer as described earlier in this book.
- 4** Restore the workspace directory, if necessary.
- 5** Restart BIRT Report Designer with the -clean option:

```
eclipse.exe -clean
```

Updating BIRT RCP Report Designer installation

Unlike BIRT Report Designer, BIRT RCP Report Designer is a stand-alone application. To update this application, you delete the entire application and reinstall a newer version. If you created your report designs and resources in the birt-rcp-report-designer-<version> directory structure, you must back up your workspace directory and any resources that you want to keep before you delete BIRT RCP Report Designer. After you install a newer version of the application, you can copy your files back to the application directory structure.

As a best practice, do not keep your workspace in the birt-rcp-report-designer-<version> directory structure. Keeping your workspace in a different location enables you to update your installation more easily in the future.

How to update BIRT RCP Report Designer

- 1** Back up the workspace directory and any other directories that contain report designs, libraries, and other resources, if they are in the birt-rcp-report-designer-<version> directory structure.
- 2** Delete the birt-rcp-report-designer-<version> directory.
- 3** Download and install BIRT RCP Report Designer as described earlier in this book.
- 4** Restore the directories that you backed up in step 1, if necessary.
- 5** Restart BIRT RCP Report Designer with the -clean option:

```
BIRT.exe -clean
```

3

Installing Other BIRT Packages

Beyond the BIRT Report Designer packages, BIRT provides a number of other separate packages as downloadable archive files on the Eclipse web site. Some of these packages are stand-alone modules, others require an existing Eclipse or BIRT environment, and still others provide additional functionality to report developers and application developers. This chapter describes the steps required to install the BIRT packages shown in the following list:

- Chart Engine
- Data Tools Platform (DTP) Integration
- Demo Database
- Report Engine
- Samples
- Source Code
- Web Tools Integration

Installing Chart Engine

Chart Engine supports adding charting capabilities to a Java application. An application can use Chart Engine without using the BIRT reporting functionality or Report Engine. Chart Engine integrates into an existing Eclipse platform on Microsoft Windows, UNIX, or Linux. You can also install Chart Engine on an existing J2EE application server. To use Chart Engine, you use its public API, org.eclipse.birt.chart.

Both BIRT Report Designer and BIRT RCP Report Designer include all the components of Chart Engine. If you are using a BIRT report designer, you do not need to install BIRT Chart Engine separately.

How to install BIRT Chart Engine

On the BIRT web site, perform the following operations:

- 1 Navigate to BIRT Downloads for build 2.2.1.

For more information about how to navigate to the BIRT web site and BIRT Downloads for build 2.2.1, see Chapter 2, “Installing a BIRT Report Designer.”

- 2 In the Chart Engine section, choose the Chart Engine archive file:

`birt-charts-2_2_1.zip`

- 3 On Eclipse downloads, choose your closest download site.

- 4 Extract the archive file to a location of your choice.

- 5 Start Eclipse from the command line with the `-clean` option to remove cached information.

The archive extraction process creates the following subdirectories in the extraction directory:

- **ChartRuntime**

This directory contains the plug-ins and libraries that an Eclipse platform requires to run, render, and edit charts.

- **ChartSDK**

This directory contains the plug-ins and libraries from the ChartRuntime directory plus the SDK that you need to create your own charting applications. It also includes examples, source code, and a Web Tools Platform (WTP) extension to support charts in web applications.

- **DeploymentRuntime**

This directory contains the libraries that you need to run your charting application in a non-Eclipse environment such as on an application server.

The Chart Engine download file also includes extensive Frequently Asked Questions (FAQ) and examples illustrating how to use Chart Engine. After extracting the archive, you can find the FAQ at the following location:

`<CHART_ENGINE>/DeploymentRuntime/ChartEngine/docs/
Charts_FAQ.doc`

The examples are in a JAR file located at:

`<CHART_ENGINE>/ChartSDK/eclipse/plugins/
org.eclipse.birt.chart.examples_<version>.jar`

Installing BIRT Data Tools Platform Integration

This package includes the minimal set of Data Tools Platform (DTP) plug-ins that BIRT Report Designer requires. If you install the BIRT Report Designer package in an existing Eclipse installation, you can install this BIRT DTP Integration package instead of the full DTP platform.

How to install BIRT DTP Integration

On the BIRT web site, perform the following operations:

- 1 Navigate to BIRT Downloads for build 2.2.1.

For more information about how to navigate to the BIRT web site and BIRT Downloads for build 2.2.1, see Chapter 2, “Installing a BIRT Report Designer.”

- 2 In the BIRT DTP Integration section, choose the BIRT DTP Integration archive file:

`birt-dtp-integration-2_2_1.zip`

- 3 On Eclipse downloads, choose your closest download site.
- 4 Extract the archive file to the directory that contains your Eclipse directory.

Extracting creates the DTP features and plug-ins in the `eclipse\features` and `eclipse\plugins` directories.

- 5 Start Eclipse from the command line with the `-clean` option.

To test the BIRT DTP Integration package, open the Report Design perspective in Eclipse, as described in the following procedure.

How to test the BIRT DTP Integration installation

- 1 Start Eclipse.
- 2 From the Eclipse main menu, choose Open Perspective→Report Design. If Report Design does not appear in the Open Perspective window, choose Other. From the list of perspectives, choose Report Design.

Eclipse displays the BIRT Report Designer perspective.

Installing BIRT Demo Database

The BIRT Demo Database package provides the Classic Models database that this book uses for example procedures. The database is provided in the following formats:

- Apache Derby
- MySQL

BIRT Report Designer and BIRT RCP Report Designer include this database in Apache Derby format, as the Classic Models Inc. sample database data source. Install BIRT Demo Database if you want to use the native drivers to access this data source.

How to install BIRT Demo Database

On the BIRT web site, perform the following operations:

- 1** Navigate to BIRT Downloads for build 2.2.1.

For more information about how to navigate to the BIRT web site and BIRT Downloads for build 2.2.1, see Chapter 2, “Installing a BIRT Report Designer.”

- 2** In the Demo Database section, choose the Demo Database archive file:

birt-database-2_2_1.zip

- 3** On Eclipse downloads, choose your closest download site.

- 4** Extract the archive file to a location of your choice.

Extracting creates a directory, ClassicModels, that contains the BIRT Demo Database in Apache Derby and MySQL formats.

To test the BIRT Demo Database, first connect to the database with the native database client tool or a Java application.

How to access BIRT Demo Database using a database client tool

Perform one of the following sets of tasks, based on your preferred database:

- Apache Derby database

Connect to the database in the derby subdirectory of ClassicModels.

- MySQL

- 1** Navigate to the mysql subdirectory of ClassicModels.

- 2** Create a database to use or edit `create_classicmodels.sql` to uncomment the lines that create and select the classicmodels database.

- 3** Use the mysql command line interface to run `create_classicmodels.sql`.

- 4** Review `load_classicmodels.sql` to determine if you can use the script on your platform without editing. Use the mysql command line interface to run `load_classicmodels.sql`.

Next, connect to the database from BIRT Report Designer or BIRT RCP Report Designer.

How to access BIRT Demo Database from a BIRT report designer

Connect to the database using BIRT Report Designer or BIRT RCP Report Designer.

- 1 To access the Classic Models database in Apache Derby or MySQL format, first add the driver JAR files to your BIRT report designer installation.
 - 2 In any report design, create a data source on the database. In the same report design, create a data set on the data source.
-

Installing Report Engine

Report Engine supports adding reporting capabilities to a Java application. BIRT Report Engine integrates into an existing Eclipse platform on Microsoft Windows, UNIX, or Linux. You can also install report engine components on an existing J2EE application server. To support quick deployment of reporting functionality to an application server, Report Engine includes a web archive (.war) file.

How to install BIRT Report Engine

On the BIRT web site, perform the following operations:

- 1 Navigate to BIRT Downloads for build 2.2.1.

For more information about how to navigate to the BIRT web site and BIRT Downloads for build 2.2.1, see Chapter 2, “Installing a BIRT Report Designer.”

- 2 In the Report Engine section, choose the Report Engine archive file:

`birt-runtime-2_2_1.zip`

- 3 On Eclipse downloads, choose your closest download site.

- 4 Extract the archive file to a suitable directory.

- 5 Create a system variable, BIRT_HOME.

Set the value of BIRT_HOME to the BIRT Report Engine installation directory. For example, if you extracted the BIRT Report Engine to C:\, the value of BIRT_HOME is:

`C:\birt-runtime-2_2_1`

To test the installation, run the Report Engine report generation command line example. This example uses a batch (.bat) file on a Windows system and a shell script (.sh) file on a UNIX or Linux system. This file takes the parameters shown in Table 3-1.

Table 3-1 Parameters for genReport script

Parameter	Valid for mode	Values
Execution mode -m		Valid values are run, render, and runrender. The default is runrender.

(continues)

Table 3-1 Parameters for genReport script (*continued*)

Parameter	Valid for mode	Values
Target encoding -e	render, runrender	A valid encoding. The default is utf-8.
Output format -f	render, runrender	Valid values are HTML and PDF. The default value is HTML.
Report parameters file -F	run, runrender	Path to the parameter file. This file contains lines with the format: <parameter name>=<value>
Locale -l locale	run, runrender	A valid locale string. The default locale is en.
Output file name -o	render, runrender	The full path of the output file. The default value is the name of the report design with an extension based on the output format, .html for an HTML file and .pdf for a PDF file.
Report parameter -p "parameter name=value"	run, runrender	If you provide parameter values with the -p parameter, these values override the values in the report parameters file specified by -F.
HTML format -t	run, runrender	Valid values are HTML and ReportletNoCSS. HTML is the default. This format wraps the HTML output in an <HTML> tag. ReportletNoCSS does not wrap the HTML output in an <HTML> tag.
Report design file	All modes	The full path of the report design file. This parameter must be the last parameter on the command line.

How to test the BIRT Report Engine installation

- 1 From the command line, navigate to the directory where you installed BIRT Report Engine.
- 2 Navigate to the ReportEngine subdirectory.
- 3 To run the genReport script, run the appropriate file for your operating system:
 - On a Windows platform, run genReport.bat.
 - On a UNIX or Linux platform, run genReport.sh.

Enclose the value for a command line parameter in quotes. For example, the following Windows platform command uses the value, Hello, for the

parameter, sample, to generate an HTML file from the report design, test.rptdesign:

```
genReport -p "sample=Hello"  
"C:\birt-runtime-2_2_1\WebViewerExample\test.rptdesign"
```

genReport generates the required output file.

- 4 Open the output file. In this example, the file is C:\birt-runtime-2_2_1\WebViewerExample\test.html.

For more information about setting up the BIRT Report Engine, see Chapter 4, “Deploying a BIRT Report to an Application Server.”

Installing BIRT Samples

BIRT Samples provides examples of a BIRT report item extension and of charting applications. The report item extension integrates into BIRT Report Designer and BIRT Report Engine.

How to install BIRT Samples

On the BIRT web site, perform the following operations:

- 1 Navigate to BIRT Downloads for build 2.2.1.

For more information about how to navigate to the BIRT web site and BIRT Downloads for build 2.2.1, see Chapter 2, “Installing a BIRT Report Designer.”

- 2 In the Samples section, choose the Samples archive file:

[birt-samples-plugins-2_2_1.zip](#)

- 3 On Eclipse downloads, choose your closest download site.

- 4 Extract the archive file to the directory that contains your Eclipse directory.
-

Installing BIRT Source Code

This package includes the source code for all BIRT plug-ins. You can examine this code to see how BIRT generates reports from designs. You can also import this source code into a workspace to build a custom BIRT installation.

How to install BIRT Source Code

- 1 Navigate to BIRT Downloads for build 2.2.1.

For more information about how to navigate to the BIRT web site and BIRT Downloads for build 2.2.1, see Chapter 2, “Installing a BIRT Report Designer.”

- 2** In the BIRT Source Code section, choose the BIRT Source Code archive file:

`birt-source-2_2_1.zip`

- 3** On Eclipse downloads, choose your closest download site.

- 4** Extract the archive file to the directory that contains your Eclipse directory.

Extracting creates the build files and BIRT features and plugins directories in the workspace directory.

To test the BIRT Source Code package, import the source code projects into your workspace.

How to test the BIRT Source Code installation

- 1** Start Eclipse.

- 2** Set the Java preferences for BIRT.

1 From the Eclipse window menu, choose Window→Preferences.

2 Select Java→Compiler. Make the following selections:

- Set Compiler Compliance Level to 5.0.
- Deselect Use default compliance settings.
- Set Generated .class files compatibility to 5.0.
- Set Source compatibility to 5.0.

3 Choose OK.

- 3** From the Eclipse window menu, choose File→Import.

- 4** On Import—Select, select General→Existing Projects into Workspace. Choose Next.

- 5** On Import—Import Projects, select Select root directory, then type or browse to your workspace directory.

The BIRT features and plug-ins appear in Projects.

- 6** Choose Finish.

Eclipse builds the BIRT projects.

If the projects do not build correctly, check that you installed the prerequisites for BIRT Report Designer, as described in Chapter 1, “Prerequisites for BIRT.” If you have not installed the BIRT Report Designer Full Eclipse Install, download this package and extract any JAR files that the build requires. Add any libraries that Eclipse does not find to the build paths of specific projects to resolve other build errors.

Installing BIRT Web Tools Integration

This package includes the minimal set of BIRT plug-ins that the Eclipse Web Tools Platform (WTP) requires to build a BIRT web project using the BIRT Web Project Wizard. This package also includes the source code for these plug-ins.

How to install BIRT Web Tools Integration

On the BIRT web site, perform the following operations:

- 1 Navigate to BIRT Downloads for build 2.2.1.

For more information about how to navigate to the BIRT web site and BIRT Downloads for build 2.2.1, see Chapter 2, “Installing a BIRT Report Designer.”

- 2 In the BIRT Web Tools Integration section, choose the BIRT Web Tools Integration archive file:

`birt-wtp-integration-sdk-2_2_1.zip`

- 3 On Eclipse downloads, choose your closest download site.

- 4 Extract the archive file to the directory that contains your Eclipse directory.

Extracting creates the BIRT features and plug-ins in the `eclipse\features` and `eclipse\plugins` directories.

To test the BIRT Web Tools Integration package, create a BIRT web project in Eclipse.

How to test the BIRT Web Tools Integration installation

- 1 Start Eclipse.

- 2 From the Eclipse window menu, choose File→New Project.

- 3 On New Project—Select a Project, select Business Intelligence and Reporting Tools→Web Project. Choose Next.

- 4 On New Project—Dynamic Web Project and subsequent pages in the wizard, make the choices that you need for your BIRT web project, then choose Finish.

If you do not have the Java EE perspective open, Eclipse displays the following message:

This kind of project is associated with the Java EE perspective. Do you want to open this perspective now?

Choose Yes.

This page intentionally left blank

4

Deploying a BIRT Report to an Application Server

One way to view a BIRT report on the web is to deploy the BIRT report viewer to an application server, such as Apache Tomcat, IBM WebSphere, JBoss, or BEA WebLogic. You deploy the BIRT report viewer by copying files from the BIRT Report Engine, which you must install separately from the BIRT Report Designer. The BIRT Report Engine includes the BIRT report viewer as a web archive (.war) file and as a set of files and folders.

This chapter provides information about deploying the BIRT report viewer using either of these sources.

About application servers

The instructions in this chapter specifically address deploying a BIRT report to Apache Tomcat version 5.5.7. While the information in this chapter is specific to this version of Tomcat, a BIRT report can also be deployed to other versions of Tomcat and to other application servers.

About deploying to Tomcat

There are only minor differences between the requirements for deploying to Tomcat version 5.5.7 and deploying to earlier versions of Apache Tomcat. Apache Tomcat 5.5.7 runs Java 5 by default, which is also the recommended version to use for BIRT 2.2.1. If you use an earlier version of Java, you need to install a compatibility package and configure Apache Tomcat to use the Java 1.4 run-time environment. For information about configuring Apache Tomcat to use Java 1.4 run-time, see the Apache Tomcat help pages. You can download Apache Tomcat from jakarta.apache.org/tomcat.

About deploying to other application servers

Most application servers require a web archive (.war) file that contains everything that the application requires, including a web.xml file describing the application and various deployment preferences. A WAR file appropriate for Tomcat is included as part of the BIRT Report Engine. Typically, the WAR file does not require modification. In some cases, developers who have experience with other application servers can modify the web.xml file to reflect the requirements of their environments. For more information about setting the web.xml parameters, see the section on mapping the report viewer folders, later in this chapter.

If you are deploying to JBoss, you might need to copy axis.jar and axis-ant.jar from WEB-INF/lib to the following directory:

```
jboss/server/default/lib
```

This step is not necessary for all versions of JBoss, but if you are experiencing difficulty with a JBoss deployment, copying these files can resolve the problem.

Placing the BIRT report viewer on an application server

You must place the BIRT report viewer in a location where Apache Tomcat can access it. Typically, this location is the \$TOMCAT_INSTALL/webapps directory. When the BIRT report viewer is in this folder, Apache Tomcat automatically recognizes and starts the BIRT report viewer the next time you restart Apache Tomcat.

Installing the BIRT report viewer as a web application

The BIRT report viewer files provide core functionality to run, render, and view BIRT reports. If you use additional JDBC drivers that are not part of the standard BIRT packages, you must install these drivers as well as the BIRT report viewer itself. If you install the BIRT report viewer as a WAR file, you must include the JDBC drivers in the WAR file.

The following instructions assume that you have installed the BIRT Report Engine from the BIRT web site, that your web application directory is \$TOMCAT_INSTALL/webapps, and that your BIRT run-time installation directory is \$BIRT_RUNTIME.

How to install the BIRT report viewer from the BIRT Report Engine WAR file

The steps to install the BIRT report viewer from the WAR file differ depending on whether you need to include additional JDBC drivers for your reports. If you have no additional drivers, you just install the WAR file from

the BIRT Report Engine installation. If you use additional JDBC drivers, you must pack them into the WAR file before you deploy it.

- To install the BIRT report viewer from the BIRT Report Engine WAR file, copy the BIRT report viewer WAR file, birt.war to the Tomcat applications folder, \$TOMCAT_INSTALL/webapps, as illustrated by the following DOS command:

```
copy $BIRT_RUNTIME/birt.war $TOMCAT_INSTALL/webapps
```

Then, restart Apache Tomcat.

- To install the BIRT report viewer with additional JDBC drivers, perform the following steps:

- 1 Create a temporary directory and navigate to that directory.
- 2 Unpack the BIRT Report Engine WAR file into the temporary directory, using a command similar to the following one:

```
jar -xf $BIRT_RUNTIME/birt.war
```

The BIRT report viewer application unpacks into your temporary directory.

- 3 Copy the JAR files for your JDBC drivers to the following folder in your temporary directory:

```
WEB-INF/platform/plugins/  
org.eclipse.birt.report.data.oda.jdbc_<version>/drivers
```

- 4 Repack the BIRT Report Engine WAR file from the temporary directory into a new birt.war file, using a command similar to the following one:

```
jar -cf birt.war *
```

This command creates birt.war in your temporary directory.

- 5 Copy the new birt.war file to the Tomcat applications folder, \$TOMCAT_INSTALL/webapps, as illustrated in the following DOS command:

```
copy birt.war $TOMCAT_INSTALL/webapps
```

- 6 Restart Apache Tomcat.

How to install the BIRT report viewer from the BIRT Report Engine viewer folder

To install the BIRT report viewer as an application in a file system folder, you use the WebViewerExample folder in the BIRT Report Engine installation.

- 1 Navigate to \$TOMCAT_INSTALL/webapps.
- 2 Create a subdirectory named birt.
- 3 Copy the web viewer example directory and all its subdirectories to this new folder, as illustrated by the following DOS command:

```
xcopy /E "$BIRT_RUNTIME/WebViewerExample"  
$TOMCAT_INSTALL/webapps/birt
```

- 4 If your BIRT reports need additional JDBC drivers, add the JAR files for your JDBC drivers to the following directory:

```
$TOMCAT_INSTALL/birt/WEB-INF/platform/plugins/  
org.eclipse.birt.report.data.oda.jdbc_<version>/drivers
```

- 5 Restart Apache Tomcat.

Testing the BIRT report viewer installation

To test the installation of the BIRT report viewer described in earlier sections, type the following URL in a web browser address field:

```
<server_name>:<port>
```

where `<server_name>` is the name of the application server and `<port>` is the port that the application server uses.

Tomcat opens the JavaServer Page (JSP), `index.jsp`. This file exists in both the WAR file and in the BIRT report viewer root directory. A link on this page runs the simple BIRT report design file, `test.rptdesign`. If the BIRT report viewer is installed correctly, Tomcat uses `index.jsp` to process the report design and generate and render the report that it describes. The first time you run the report, Tomcat compiles the JSP files that comprise the viewer, so there is a delay before the report appears in the web browser.

Using a different context root for the BIRT report viewer

By default, the context root of the URL for a web application is the path to the application directory or the WAR file. The default WAR file for the BIRT report viewer is `birt.war`, so the default URL to access a BIRT report from Apache Tomcat is similar to the following one:

```
http://localhost:8080/birt/run?__report=myReport.rptdesign
```

To change the BIRT context root, change the name of the `/birt` directory or the WAR file in `$TOMCAT_INSTALL/webapps`. Next, restart Apache Tomcat. In the URL to access your BIRT report, specify the name that you chose. For example, if you choose `reports`, the URL to access a BIRT report becomes:

```
http://localhost:8080/reports/run?__report=myReport.rptdesign
```

The URL examples in this section access the report design with a relative path. The `BIRT_VIEWER_WORKING_FOLDER` parameter sets the path to access a report design with a relative path.

Placing the BIRT report viewer in a different location

If you need to place the BIRT report viewer in a location other than `$TOMCAT_INSTALL/webapps`, you must add a context mapping entry to the `server.xml` file in `$TOMCAT_INSTALL/conf`.

To add a context mapping entry, add the following line to server.xml just above the </host> tag near the end of the file:

```
<Context path="/birt_context" docBase="BIRT_Path"/>
```

where *birt_context* is the context root you want for the BIRT report viewer application and *BIRT_Path* is the absolute path to the directory containing the BIRT report viewer.

Save your changes to server.xml and restart Apache Tomcat to make your changes active.

Understanding the BIRT report viewer context parameters

To determine the locations for report designs, images in reports, and log files, the BIRT report viewer uses context parameters defined in the web.xml file. When you provide a path as the value for one of these parameters, it can be relative or absolute. A relative path is relative to the root folder of the BIRT report viewer application. A path to a writable location for a BIRT report viewer that is deployed as a WAR file must be an absolute path.

Other context parameters determine other aspects of the behavior of the BIRT report viewer, such as the default locale and the level of detail in the viewer's log files. Chapter 5, "Using Eclipse BIRT Web Viewer" describes all these parameters.

How to set the location for report designs

By default, the relative path for report designs is relative to the BIRT report viewer's root folder. You must place all report designs in this folder or use the full path to the report design in the URL. This requirement is not convenient for deployment in a WAR file. To set a different location for report designs, change BIRT_VIEWER_WORKING_FOLDER in the BIRT report viewer application's web.xml file.

- 1 Navigate to \$TOMCAT_INSTALL/webapps.
- 2 Open web.xml in a code editor.

To open web.xml in an editor, perform one of the following steps, based on your deployment configuration:

- If you use a WAR file to deploy the BIRT report viewer, extract WEB-INF/web.xml from birt.war into a temporary location.
- If you use a folder to deploy the BIRT report viewer, navigate to <context root>/WEB-INF.

- 3 Locate the following element:

```
<context-param>
    <param-name>BIRT_VIEWER_WORKING_FOLDER</param-name>
    <param-value></param-value>
</context-param>
```

- 4** Change the param-value element, so that it includes the absolute path to the folder for the report designs:

```
<context-param>
  <param-name>BIRT_VIEWER_WORKING_FOLDER</param-name>
  <param-value>Report_Folder</param-value>
</context-param>
```

where Report_Folder is the absolute path to the folder for the report designs.

- 5** Save web.xml and close the editor.
- 6** If you use a WAR file to deploy the BIRT report viewer, replace WEB-INF/web.xml in birt.war with the name of the file you just modified.
- 7** Copy your report designs into the folder you specified in the param-value element for BIRT_VIEWER_WORKING_FOLDER.
- 8** Restart Apache Tomcat.

Verifying that Apache Tomcat is running BIRT report viewer

If you have problems accessing the BIRT report viewer, you can use the Tomcat manager to verify that the BIRT report viewer is running on Apache Tomcat. To run the Tomcat manager, you need a manager's account. If you have not already set up a Tomcat manager account, you can do so by adding the following line to \$TOMCAT_INSTALL/conf/tomcat-users.xml:

```
<user name="admin" password="tomcat" roles="manager" />
```

When you have a manager's account, first open the Tomcat main page, which for a typical Apache Tomcat installation is <http://localhost:8080>, as shown in Figure 4-1.

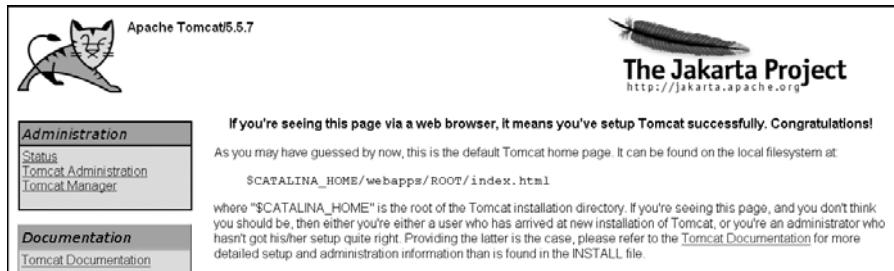


Figure 4-1 Apache Tomcat home page

On the Tomcat main page, in the Administration panel, choose Tomcat Manager. In the manager login window, type the user name and password of the manager account defined in the tomcat-users.xml file. When the BIRT report viewer application is running, the Running status for Eclipse BIRT Report Viewer is true, as shown in Figure 4-2.

Tomcat Web Application Manager					The Tomcat Server
Message:	OK				
Manager					
List Applications		HTML Manager Help		Manager Help	Server Status
Applications					
Path	Display Name	Running	Sessions	Commands	
/	Welcome to Tomcat	true	0	Start	Stop Reload Undeploy
/admin	Tomcat Administration Application	true	0	Start	Stop Reload Undeploy
/balancer	Tomcat Simple Load Balancer Example App	true	0	Start	Stop Reload Undeploy
/birt-viewer	Eclipse BIRT Report Viewer	true	0	Start	Stop Reload Undeploy

Figure 4-2 Running status for the BIRT report viewer

Placing fonts on the application server

BIRT Report Engine requires certain TrueType fonts in order to display a PDF report. BIRT searches for fonts in the common font directories for Windows and Linux. The list of directories that BIRT searches includes on a Windows system:

- /windows/fonts for drives A through G
 - /WINNT/fonts for drives A through G
- and on a Linux system:
- /usr/share/fonts/default/TrueType
 - /usr/share/fonts/truetype

If your PDF reports appear to be missing content, you can place the necessary fonts in any of the directories in the preceding list. You can also specify your own font search path in the environment variable `BIRT_FONT_PATH`.

Viewing a report using a browser

After you deploy the BIRT report viewer to your J2EE container, you can access your BIRT reports using a web browser to access the two available BIRT report viewer servlets. To view a BIRT report using a browser, you navigate to a URL with one of the following two formats:

`http://localhost:8080/birt/run?parameter_list`
`http://localhost:8080/birt/frameset?parameter_list`

where `parameter_list` is a list of URL parameters. For more information about the URL parameters, see the section about URL parameters in Chapter 5, “Using Eclipse BIRT Web Viewer.”

The run and frameset servlets display reports in two different ways. When you use the run servlet, it displays the report as a stand-alone web page or a PDF file. If the report requires parameters, you must specify them in the URL.

When you use the frameset servlet, the browser displays a page with a toolbar with four buttons to do the following tasks:

- Print the report.
- Display a table of contents.
- Display a parameters dialog.
- Display a dialog for exporting data.

Using connection pooling on Tomcat

BIRT provides support for connection pooling. If you have a connection pool configured on your Tomcat application server, you can set up your BIRT reports to use a connection from the connection pool when connecting to a JDBC database.

The BIRT JDBC data source uses a property named JNDI URL to access the connection pool service on the web application server to retrieve a connection from the pool.

Setting up a report to use connection pooling

You configure your reports to use connection pooling using BIRT Report Designer.

The BIRT JDBC data source wizard requires also configuring a direct access connection along with the JNDI URL, you provide. The reason for that is that some JNDI service providers do not support client-side access. During design time, these JDBC drivers need to use the direct access JDBC connection. The JDBC data set query builder continues to use direct JDBC connection to obtain its metadata. In BIRT Report Designer, only the design functions directly related to a data source design, such as Test Connection and Preview Results of a data set, first attempt to use a JNDI name path. If the JNDI connection is not successful for any reason, the data source falls back to use the JDBC driver direct access URL.

Similarly, at report run-time, such as during report preview, the JDBC run-time driver attempts to look up its JNDI data source name service to get a pooled JDBC connection. If such lookup is not successful for any reason, it falls back to use the JDBC driver URL directly to create a JDBC connection.

Using a jndi.properties file

Each individual JNDI application on the web application server has its own environment settings. These settings are stored in the JVM system properties.

The JNDI reads the following standard JNDI properties from the system properties:

```
java.naming.factory.initial  
java.naming.factory.object  
java.naming.factory.state  
java.naming.factory.control  
java.naming.factory.url.pkgs  
java.naming.provider.url  
java.naming.dns.url
```

To simplify the task of setting up the JNDI initial context environment for an individual JNDI application, the JNDI feature supports the use of a jndi.properties resource file. Install this file in the drivers subfolder of the oda.jdbc plugin:

```
WEB-INF\platform\plugins\  
org.eclipse.birt.report.data.oda.jdbc_<version>\drivers
```

This file contains a list of key-value pairs presented in the properties file format, key=value. The key is the name of the property, for example, java.naming.factory.object, and the value is a string, for example, jnp://localhost:1099.

Here is an example of a JNDI resource file used with JBoss application server:

```
java.naming.factory.initial=  
    org.jnp.interfaces.NamingContextFactory  
java.naming.provider.url=jnp://localhost:1099  
java.naming.factory.url.pkgs=  
    org.jboss.naming:org.jnp.interfaces
```

The JDBC run-time driver looks for the jndi.properties file in the web application's folder tree. When the driver does not find the file or has a problem reading from it, the initial context uses the default behavior, as defined by javax.naming.Context, to locate any JNDI resource files. Note that you must configure the proper classpath for the classes referenced by the environment properties.

Configuring a JNDI connection object on Tomcat

The JNDI URL property for the JDBC data source supports retrieving a JDBC connection from a pool when BIRT reports are deployed to a web application server.

You can find more information about how to configure connection pooling on Tomcat in the Tomcat official documentation at:

```
http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html
```

The following example assumes you already have deployed the BIRT report viewer to Tomcat 5.5 application server in the folder, \$TOMCAT_INSTALL/webapps/birt, as described earlier in this chapter.

How to configure a JNDI connection object on Tomcat

- 1 Install your JDBC Driver.

Make an appropriate JDBC driver available to both Tomcat internal classes and your web application, for example, by installing the driver's JAR files into the following directory:

\$CATALINA_HOME/common/lib

- 2 Declare your resource requirements in the BIRT report viewer's WEB-INF/web.xml file. For example, add the following entry to set up a JNDI service for the Classic Models MySQL format database with the name, MySqlDB:

```
<resource-ref>
    <description>Resource reference to a factory for
        java.sql.Connection</description>
    <res-ref-name>jdbc/MySqlDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

- 3 Configure Tomcat's resource factory as a Resource element in the BIRT report viewer's META-INF/context.xml file, similar to the following lines:

```
<Context>
    <Resource name="jdbc/MySqlDB" auth="Container"
        type="javax.sql.DataSource" maxActive="5" maxIdle="-1"
        maxWait="10000" username="root" password=""
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/classicmodels"
        description="MySQL DB"/>
</Context>
```

- 4 Make the JNDI URL in your report design match this resource factory, similar to the following line:

java:comp/env/jdbc/MySqlDB

Open the report design using BIRT Report Designer. Edit the data source. On Edit Data Source, in JNDI URL, type the URL, as shown in Figure 4-3.

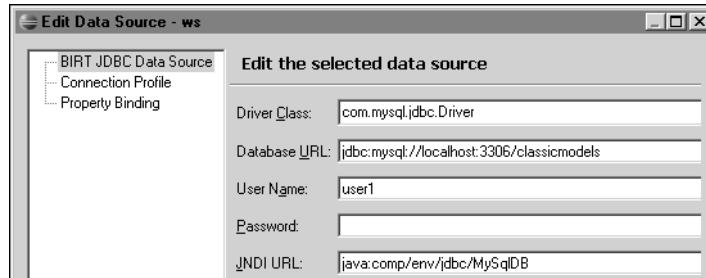


Figure 4-3 Setting the JNDI URL for a JDBC data source

- 5** Copy your report design to the BIRT report viewer root folder.
- 6** Restart the Tomcat service.
- 7** Run your report using a URL similar to the following one:

```
http://localhost:8080/birt/  
run?__report=myJNDIReport.rptdesign
```

The report uses a connection from the connection pool to connect to the Classic Models database on a MySQL server.

For more information on deploying using the Eclipse BIRT Web Viewer, see Chapter 5, “Using Eclipse BIRT Web Viewer.”

This page intentionally left blank

5

Using Eclipse BIRT Web Viewer

Eclipse BIRT 2.2.1 provides an improved sample Web Viewer. The Web Viewer is an AJAX-based, J2EE application that illustrates how to use the BIRT engine to generate and render report content.

The Web Viewer consists of JSP fragments, Java code-behind pages, servlets, JavaScript classes, and cascading style sheets. The Web Viewer uses the prototype JavaScript framework to implement AJAX-based communications within the Web Viewer and to the Web Viewer servlets. The Web Viewer also has a set of events monitored by the Web Viewer. You can use the plug-in format of the Web Viewer in existing Eclipse-based architectures, such as RCP applications.

This chapter describes how to use the sample Eclipse BIRT Web Viewer to generate and run a report.

Understanding Eclipse BIRT Web Viewer

The Web Viewer supports interactive features such as a table of contents, exporting report content to various formats, report pagination, and client and server-side printing. You can enable these features using the Web Viewer navigation and tool bars. Figure 5-1 shows an example of the Web Viewer displaying a sample report running in a browser.

The Web Viewer toolbar contains icons for displaying the table of contents, re-running the report, exporting data, exporting the report to another format, and client and server-side printing. Figure 5-2 shows the Web Viewer toolbar.

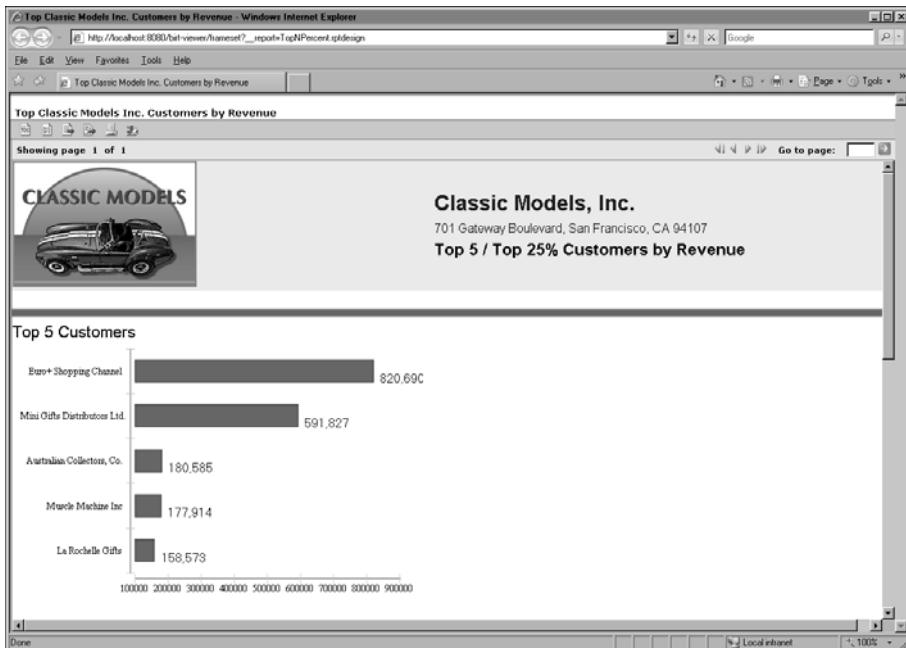


Figure 5-1 Displaying a sample report



Figure 5-2 Web Viewer toolbar

The Web Viewer supports automatic generation of a table of contents based on the report design. You can customize the table of contents (TOC) by modifying the group sections of the report.

To access the table of contents in the sample Web Viewer, select Toggle table of contents. Figure 5-3 shows the Web Viewer displaying a report that has the table of contents visible.

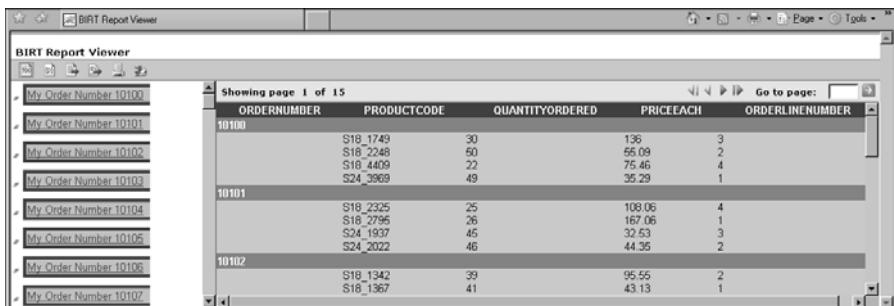


Figure 5-3 Displaying a report with the table of contents visible

To re-run a report, select Run report in the toolbar. This feature is useful when a user wants to modify the set of parameters used to generate the report. Figure 5-4 shows the Web Viewer re-running a report with new parameters.

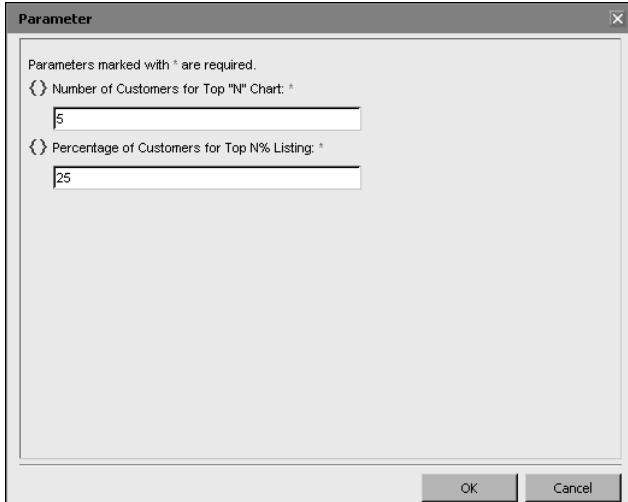


Figure 5-4 Re-running a report with new parameters



To export report data, select Export data. Figure 5-5 shows the Web Viewer exporting data from a report.

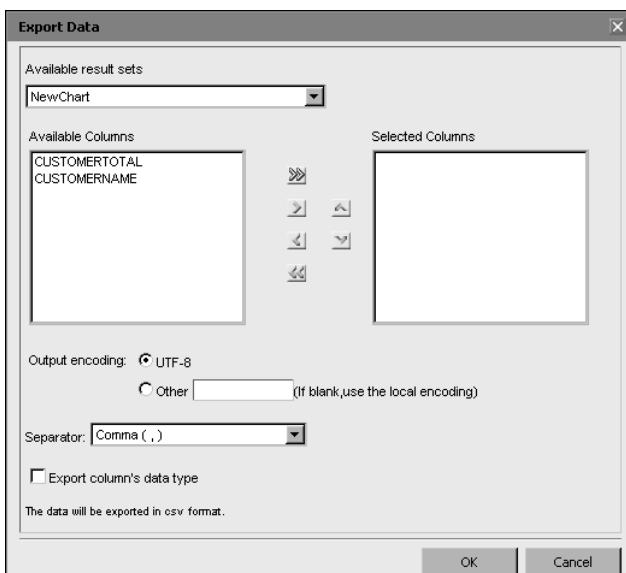


Figure 5-5 Exporting data from a report

In the Web Viewer, Export Data provides the following features:

- Available result sets

Contains the list of all data-bound controls in the report. The result sets are not BIRT data sets. These result sets contain the bound columns from the data container retrieved from a data set.

- Available columns
Displays exportable report columns.
- Selected columns
Displays the report columns selected for export.
- Output encoding
Specifies UTF-8 or other encoding.
- Separator
Specifies the data delimiter, such as comma, semicolon, colon, vertical line, or tab.
- Export column's data type
If selected, includes data type information for each column in the output.



To export a report, select Export report in the toolbar. Figure 5-6 shows the Web Viewer exporting a report.

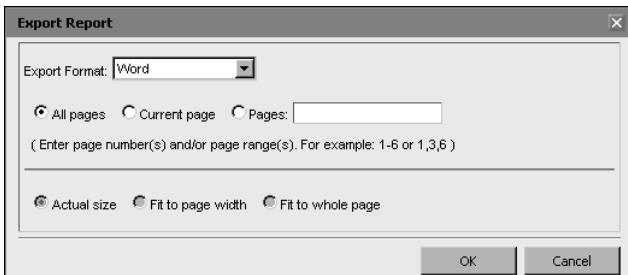


Figure 5-6 Exporting a report

In the Web Viewer, Export Report provides the following features:

- Export Format
Contains the list of available output formats, such as Word, PowerPoint, PDF, Postscript, or Excel.
- Pages
Specifies all pages, the current page, or a page range.
- Size
Specifies actual size, fit to page width, or fit to whole page.



To print a report on the client side, select Print report. Figure 5-7 shows the Web Viewer specifying how to print a report on the client side.

In the Web Viewer, Print Report provides the following features:

- Print Format

- Specifies HTML or PDF. PDF supports specifying the page size as actual size, fit to page width, or fit to whole page.
- Pages
 - Specifies all pages, the current page, or a page range.

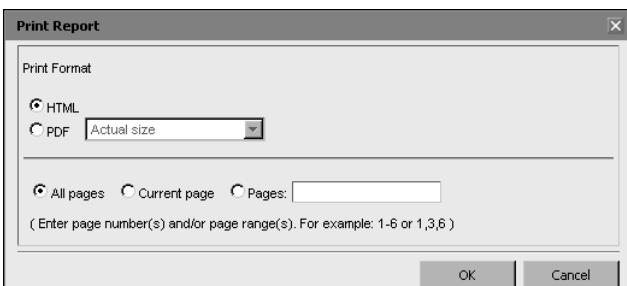


Figure 5-7 Printing a report on the client side

To print a report on the server side, select Print report on the server. Figure 5-8 shows the Web Viewer printing a report on the server side.

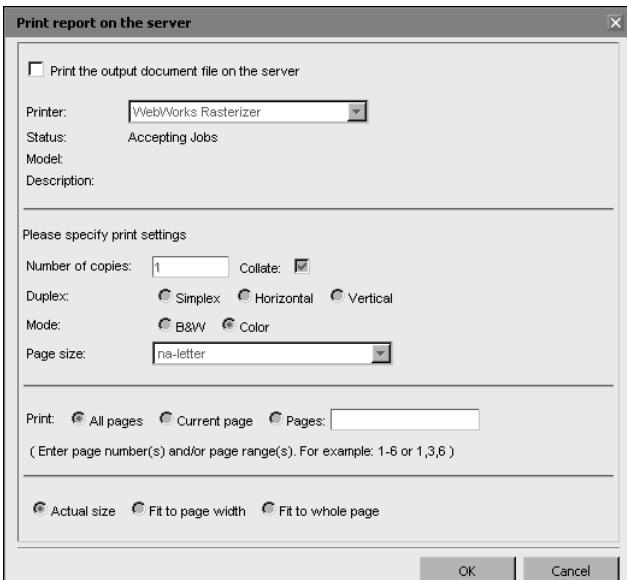


Figure 5-8 Printing a report on the server side

In the Web Viewer, Print report on the server provides the following features:

- Print the output document on the server
 - Enables server-side printing
- Printer
 - Selects the printer to use in printing

- Please specify print settings
Indicates the number of copies, whether to collate, and other preferences such as simplex or duplex, page orientation, color mode, and page size
- Pages
Specifies all pages, the current page, or a page range
- Size
Specifies actual size, fit to page width, or fit to whole page

To disable server-side printing, add the following settings to web.xml, as shown in Listing 5-1.

Listing 5-1 Disabling server-side printing in web.xml

```
<context-param>
    <param-name>BIRT_VIEWER_PRINT_SERVERSIDE</param-name>
    <param-value>OFF</param-value>
</context-param>
```

The Web Viewer navigation bar supports displaying a multipage report. The page break properties of the report elements and groups determine pagination. A report developer can control pagination with before and after page breaks or by using the page break interval, which specifies how many rows for a container element to process before inserting a page break into the generated HTML.

When using the Web Viewer with the servlet frameset mapping, the default behavior loads the first page and enables the navigation bar controls. The navigation bar controls allow the user to navigate to the first, previous, next, and last page or enter a page number to load a page directly. Figure 5-9 shows the Navigation toolbar.

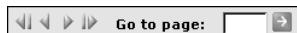


Figure 5-9 Navigation toolbar

Understanding Web Viewer architecture

The Web Viewer is a sample AJAX-based, J2EE application that encapsulates the BIRT engine. The birt-runtime-2_2_1.zip file contains the sample Web Viewer. The Web Viewer is in the WebViewerExample directory of the ZIP file. Download the birt-runtime-2_2_1.zip file from the following location:

<http://download.eclipse.org/birt/downloads/>

You can use the plug-in format of the Web Viewer in existing Eclipse-based architectures, such as RCP applications. When launching in this environment, the Web Viewer depends on the org.eclipse.tomcat plug-in to run the application. This chapter discusses this option later.

The Web Viewer consists of JSP fragments, Java code-behind pages, servlets, JavaScript classes, and cascading style sheets. The functionality of each of these components depends on the Servlet mapping currently in use.

Web Viewer servlets

The Web Viewer consists of two main servlets, the `ViewerServlet` and the `BirtEngineServlet`. The `ViewerServlet` handles the frameset and run servlet mappings set in the `web.xml`. The frameset mapping renders the report in the full AJAX viewer, complete with toolbar, navigation bar, and table of contents features.

This mapping also generates an intermediate report document from the report design file to support the AJAX-based features. The run mapping runs and renders the report, but does not create the report document.

This mapping does not supply HTML pagination, table of contents, or toolbar features, but it does use the AJAX framework to collect parameters and retrieve the report output in HTML format. This mapping also supports the AJAX features required to cancel report execution using the progress bar.

The `BirtEngineServlet` handles the preview, output, and download mappings specified in `web.xml`. The preview mapping runs and renders the report, but does not generate the report document.

This mapping can use an existing report document in which case only the render operation occurs. This process sends the output from the run and render operation directly to the browser.

The download mapping is used internally to export the report data to CSV format and requires a report document to exist. The output mapping is used by the Web Viewer when a user selects the print icon in the toolbar and runs the report if no report document exists. The engine then renders the report to the browser and calls the browser print dialog.

Both servlets extend the Axis servlets that handle SOAP communications. Table 5-1 shows the Web Viewer Servlet mappings.

Table 5-1 Web Viewer servlet mappings

Mapping	Description
download	Used internally by the Web Viewer or frameset when extracting results in CSV format. Do not use externally. <code>BirtSimpleExportDataDialog.js</code> creates a hidden form and submits to servlet for processing.
frameset	Use to launch the AJAX-based Web Viewer. Contains the toolbar, navbar, and table of contents features. Run and render tasks are separated.
parameter	Used internally by the designer to create a <code>rptconfig</code> file when selecting the preview tab for a report that contains parameters. Do not use externally.

(continues)

Table 5-1 Web Viewer servlet mappings (*continued*)

Mapping	Description
preview	Use to run and render a report or to render an existing rptdocument directly to an output format. Does not use the AJAX framework, but can launch a Parameter page.
run	Use to launch the Web Viewer without the toolbar, navbar, or table of contents. Uses the runandrender task to create the output, but does not support pagination or create a rptdocument. Uses the AJAX framework to support canceling a report.

Figure 5-10 shows the structure of the Web Viewer application.

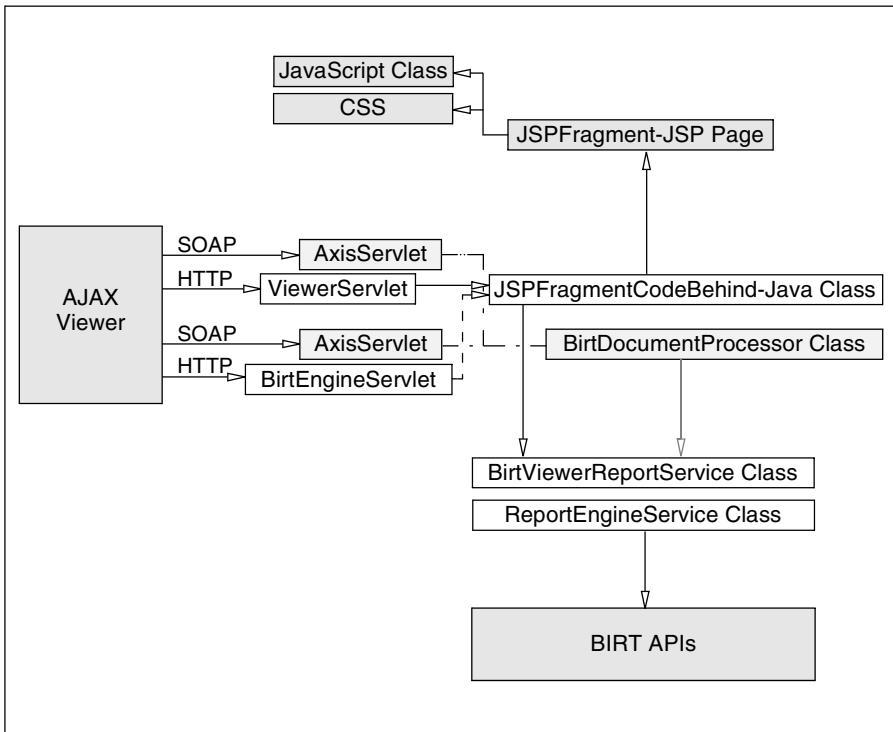


Figure 5-10 Structure of the Web Viewer application

Web Viewer URL parameters

The Web Viewer supports a list of URL parameters that you can use to modify the way a report generates and displays. Typically, these parameters are passed by name value pairs.

Append this parameter to the URL as &__format=pdf. Table 5-2 describes the URL parameters.

Table 5-2 URL parameters

Attribute	Description
__agentstyle	If true, the HTML emitter outputs styles directly to the report output and depends on the browser to implement the style calculation. If false, the BIRT style engine calculates the styles.
__bookmark	Specifies a specific bookmark within the report to load. The Web Viewer loads the appropriate page.
__clean	This value instructs the Web Viewer to clean up temporary files when a session is complete. Specifying false will instruct the Web Viewer to not clear the temporary data. The default is true.
__designer	Specifies that the Web Viewer executes in the report designer. Do not use this parameter externally.
__document	Sets the name for rptdocument. The document is created when the report engine separates run and render tasks, and is used to support features such as the table of contents and pagination. This setting is an absolute path or relative to the working folder. If no document parameter is used, a unique document is created in the document folder.
__dpi	Used to set the chart resolution.
__exportEncoding	This setting is used internally to set the encoding of exported data, when the Export Data toolbar button is used.
__fittopage	Specifies whether PDF generation should fit content to a page. Valid values are true and false.
__format	Specifies the desired output format, such as PDF, HTML, DOC, PPT, or XLS.
__id	A unique identifier for the Web Viewer.
__islocale	Specifies whether the parameter is localized.
__imageID	This setting is used internally by the Web Viewer to retrieve a specific image within a report.
__isnull	Specifies that a report parameter has a null value. For example, __isnull=Myparameter.

(continues)

Table 5-2 URL parameters (*continued*)

Attribute	Description
__isreportlet	<p>Set to true to render only a portion of the report. This parameter works in conjunction with the __bookmark parameter. The parameter affects only non-HTML output when using the frameset mapping. To render only a portion of HTML, use the preview mapping and supply a __document parameter for a previously generated report.</p> <p>For example, if a pre-generated report contains three charts and each chart uses a bookmark expression, such as chart1 and chart2, use the following URL to render only chart2:</p> <pre>http://server:port/viewer/preview?__report=reportname.rptdesign&__document=pregeneratedreport.rptdocument&__isreportlet=true&__bookmark=chart2</pre>
__locale	<p>Specifies the locale for the specific operation, overriding the default locale. The following list is in the order of precedence:</p> <ul style="list-style-type: none">■ __locale parameter■ Locale from client browser■ Locale web.xml setting■ Locale for the application server
__masterpage	Indicates that the report master page should be used or not. Valid values are true and false.
__maxrows	This parameter sets the maximum number of rows to retrieved using the report engine. The designer specifies this parameter when previewing the report.
__navigationbar	Determines if the navigation bar is shown in the Web Viewer frameset. Defaults to true. Valid values are true and false.
__overwrite	This setting, if set to true, forces an overwrite of the existing report document. This setting overrides the initial setting in the web.xml. By default, the report document is overwritten any time the report design is changed.
__page	Specifies a specific page to render. When used with the frameset mapping and HTML output, this parameter renders the entire report and loads the selected page. To render only the page, use the preview mapping with a __document parameter and no __report parameter.

Table 5-2 URL parameters (*continued*)

Attribute	Description
<code>_pagebreakonly</code>	This setting is used to generate a PDF. If true, only explicit page breaks are placed in the emitted PDF. Page Breaks do not occur because of page size.
<code>_pagerange</code>	Specifies page range to render. See the notes on <code>_page</code> parameter. Using <code>_pagerange</code> and the frameset mapping with HTML output has no effect.
<code>_parameterpage</code>	Determines if the Parameter page displays. By default, the frameset, run, and preview mappings automatically determine if the Parameter page is required. This setting overrides this behavior. Valid values are true and false.
<code>_pattern</code>	This parameter is used with the tag libraries only to set the servlet pattern. <ul style="list-style-type: none">■ <code>_target</code>■ <code>_nocache</code>■ <code>_asattachment</code>■ <code>_action</code> Used internally by the Web Viewer to call server side printing■ <code>_dpi</code>
<code>_report</code>	Sets the name of the report design to process. This setting can be an absolute path or relative to the working folder.
<code>_resourceFolder</code>	Specifies the resource folder to use. This setting will override the default setting in the web.xml. The resource folder is used to locate libraries, images, and resource files.
<code>_rtl</code>	Specifies whether to display the report in right-to-left format. The default is false.
<code>_sep</code>	This setting is used internally to set the data delimiter of exported data, when the Export Data toolbar button is used.
<code>_showTitle</code>	Determines if the report title is shown in the Web Viewer frameset. Defaults to true. Valid values are true and false.
<code>_svg</code>	Specifies whether SVG is supported.
<code>_title</code>	Sets the report title.
<code>_toolbar</code>	Determines if the Report toolbar is shown in the Web Viewer frameset. Defaults to true. Valid values are true and false.

Web Viewer fragments

The Web Viewer is constructed using JSP fragments that have a corresponding Java class instantiated by the Web Viewer servlets. The JSP fragments are in the webcontent/birt/pages directory.

The pages that are included depend on the servlet mapping called. There are four top-level fragments that include child fragments. Two are used with the ViewerServlet and two with the BirtEngineServlet.

The two associated with the ViewerServlet are the FramesetFragment and the RunFragment. These fragments correspond to the /viewer and run servlet mappings. The BirtEngineServlet has the RequestorFragment and the EngineFragment.

Figure 5-11 shows the ViewerServlet FramesetFragment, including the top-level and associated child fragments.

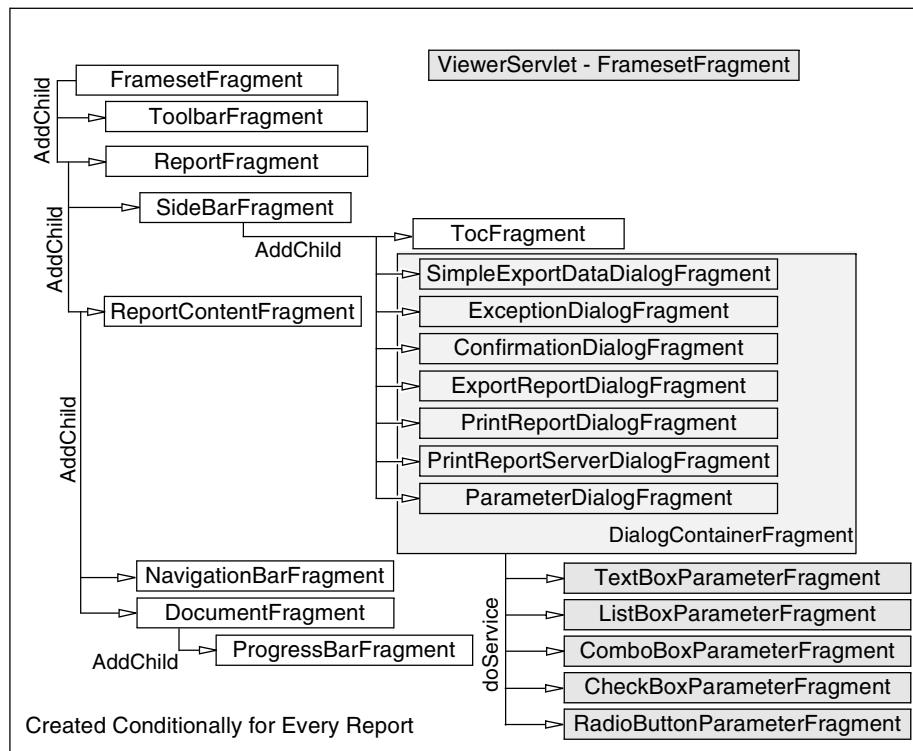


Figure 5-11 ViewerServlet FramesetFragment

Figure 5-12 shows the ViewerServlet RunFragment.

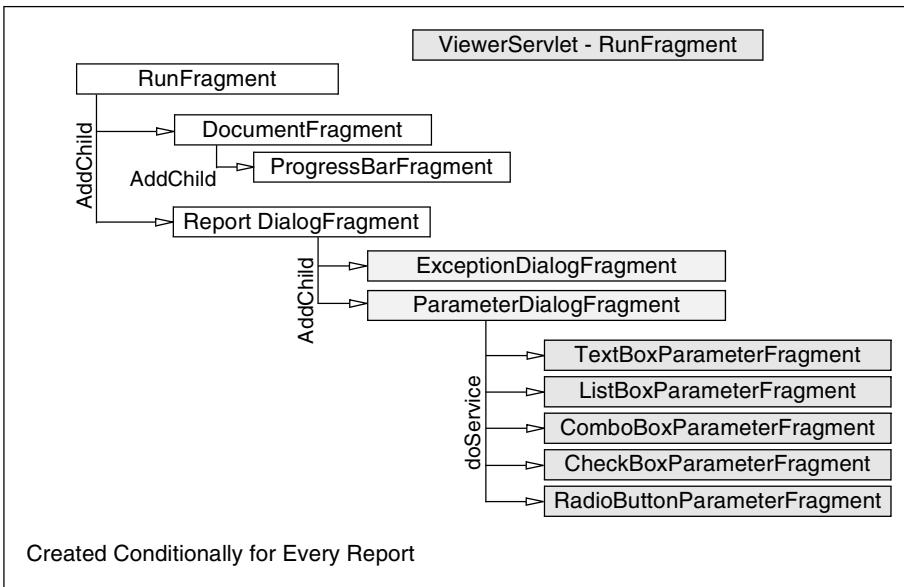


Figure 5-12 ViewerServlet Run Fragments

Figure 5-13 shows the BirtEngineServlet EngineFragment.

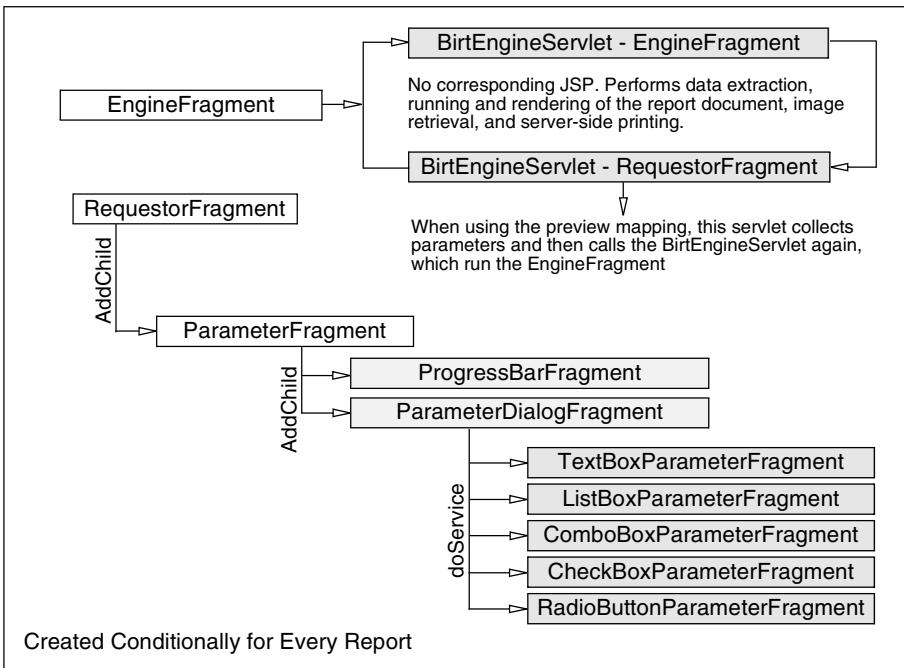


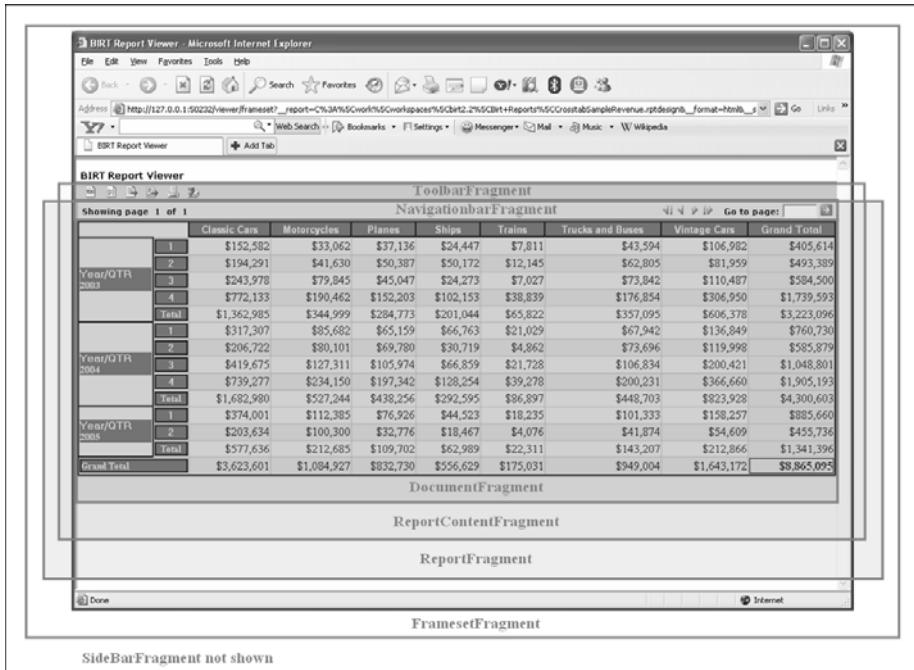
Figure 5-13 BirtEngineServlet EngineFragment

The RequestorFragment is called when the preview mapping is used with parameters and when the parameter mapping is called. The EngineFragment is used with the download mapping or with the preview mapping when parameters are not required. The EngineFragment essentially presents no GUI.

When the service method is called for the FramesetFragment, its service method includes FramesetFragment.jsp. FramesetFragment.jsp calls the AbstractBaseFragment callBack method. This method calls the AbstractBaseFragment service method for each child of FramesetFragment, which in turn includes their JSP pages, repeating the process for the entire set of fragments for the composite Web Viewer.

Each top-level fragment includes the appropriate sub-fragments to construct the Web Viewer. The preceding figures show the structure of the JSP fragments for each servlet.

Figure 5-14 shows a typical FramesetFragment rendered by the Web Viewer.



Web Viewer web.xml settings

The web.xml file contains many settings used to configure the Web Viewer. Table 5-3 lists these web.xml settings.

Table 5-3 Web Viewer web.xml settings

Setting	Description
BIRT_OVERWRITE_DOCUMENT	Specifies whether to overwrite the report document every time a report is executed. Valid values are true and false. Default is true.
BIRT_RESOURCE_PATH	This setting specifies the resource path used by report engine. The resource path is used to search for libraries, images, and properties files used by a report. If this setting is left blank, resources are searched for in the same directory as the report.
BIRT_VIEWER_CONFIG_FILE	Specifies the location of the viewer.properties file. This file contains various settings used by the Web Viewer.
BIRT_VIEWER_DOCUMENT_FOLDER	If a report document parameter, such as __document, is not used, this setting specifies the location to generate the report documents. If this setting is left blank, the default value, webapps/documents, is used. If the __document URL parameter is used and the value is relative, the report document is created in the working folder.
BIRT_VIEWER_IMAGE_DIR	Specifies the default location to store temporary images generated by the report engine. If this setting is left blank, the default location of webapps/report/images is used.
BIRT_VIEWER_LOCALE	Sets the default locale for the Web Viewer.
BIRT_VIEWER_LOG_DIR	Specifies the default location to store report engine log files. If this setting is left blank, the default location of webapps/logs is used.
BIRT_VIEWER_LOG_LEVEL	Sets the report engine log level. Valid values are <ul style="list-style-type: none">■ OFF■ SEVERE■ WARNING■ INFO■ CONFIG■ FINE■ FINER■ FINEST

(continues)

Table 5-3 Web Viewer web.xml settings (*continued*)

Setting	Description
BIRT_VIEWER_MAX_ROWS	Specifies the maximum number of rows to retrieve from a data set.
BIRT_VIEWER_PRINT_SERVERSIDE	Specifies whether server-side printing is supported. If set to OFF, the toolbar icon used for server-side printing is removed automatically. Valid values are ON and OFF.
BIRT_VIEWER_SCRIPTLIB_DIR	Specifies the default location to place JAR files used by the script engine. These files can be JAR files used by the script engine or JAR files containing event handlers written in Java. These JAR files are appended to the classpath. If this setting is left blank, the default value of webapps/scriptlib is used.
BIRT_VIEWER_WORKING_FOLDER	Specifies the default location for report designs. If the report design specified in a URL parameter is relative, this path is prepended to the report name.
HTML_ENABLE_AGENTSTYLE_ENGINE	Specifies how BIRT styles are handled with the HTML emitter. If set to TRUE, the BIRT engine outputs the styles directly to the report and depends on the browser to implement the style calculations. If set to FALSE, the emitter uses the BIRT style engine to calculate the styles and outputs the results directly to the report.
WORKING_FOLDER_ACCESS_ONLY	Specifies whether to prevent a user from entering a full path to a report. Relative paths below the working folder are accessible. If this value is set to true, reports are only searched for relative to the working folder.

Web Viewer directory structure

Figure 5-15 shows the WebViewerExample directory structure.

Most of the directories are configurable using variables set within WEB-INF/web.xml. The WEB-INF/platform, webcontent, and WEB-INF/lib directories are exceptions.

The BIRT plug-ins and associated OSGi configuration files are located in the WEB-INF/platform directory. This hard-coded directory is in the Web Viewer, but allows the application to be deployed in WAR format by using the PlatformServletContext class.

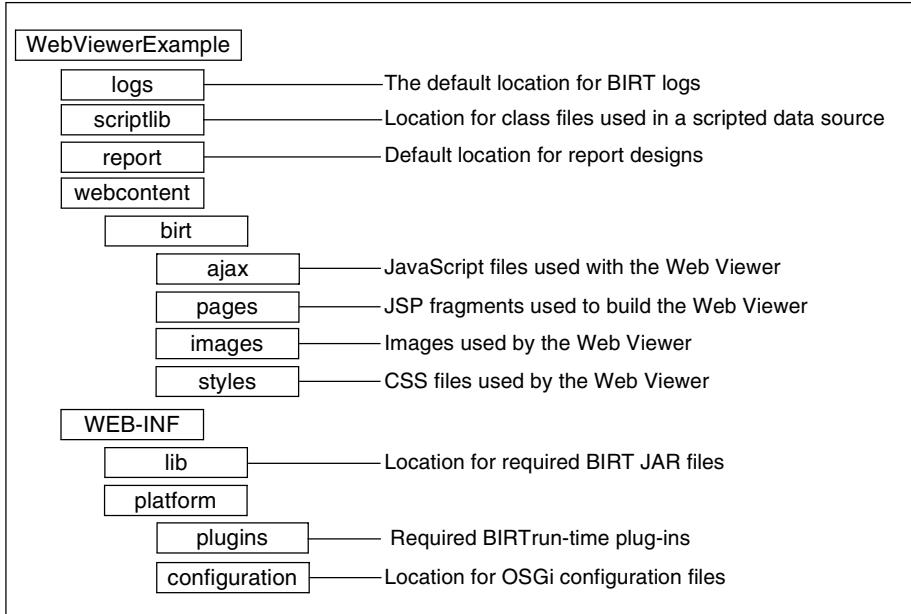


Figure 5-15 WebViewerExample Directory Structure

The **webcontent** directory contains the JavaScript files used for AJAX communications, the JSP fragments used to construct the Web Viewer instance, image files used by the Web Viewer, and the cascading style sheets used within the Web Viewer.

The **WEBINF/lib** directory contains the required JAR files for the Web Viewer to operate. Place additional JAR files used by deployed reports in the **WEB-INF/lib** or the **scriptlib** directories.

Web Viewer AJAX operation

The Web Viewer uses the prototype JavaScript framework to implement AJAX-based communications within the Web Viewer and to the Web Viewer servlets. The Web Viewer has a set of events that are monitored by various portions of the Web Viewer.

The Web Viewer makes use of the `BirtEventDispatcher` class to handle registering and firing these events. Do not confuse these events with native events that are handled with `Prototype.js Event.Observe` method.

When these events are fired, the BIRT event dispatcher calls the handler functions that monitor the selected event. Typically, the handler posts a SOAP message to the Web Viewer servlet to replace content, such as navigating to a new page, launching a dialog, or executing a standard HTTP post to the Web Viewer to export a report to another format. The key to understanding the AJAX operation is to know what events are registered and what code is monitoring the event, including when each event is fired.

The following list provides a map of these events:

- `__E_BLUR`
- `__E_CACHE_PARAMETER`
- `__E_CANCEL_TASK`
- `__E_CASCADING_PARAMETER`
- `__E_CHANGE_PARAMETER`
- `__E_EXPORT_REPORT`
- `__E_GETPAGE_ALL`
- `__E_GETPAGE_INIT`
- `__E_GETPAGE`
- `__E_PARAMETER`
- `__E_PDF`
- `__E_PRINT`
- `__E_PRINT_SERVER`
- `__E_QUERY_EXPORT`
- `__E_TOC`
- `__E_TOC_IMAGE_CLICK`
- `__E_WARN`

The majority of these events are registered for handling in the `BirtReportDocument` class initializer and are fired in UI JavaScript classes. BIRT generally uses the `__beh_function` syntax convention to denote that this event handler is written to handle BIRT events and uses the `__neh_function` convention when using the prototype `Event.Observe` method to register a native event.

The `BirtReportDocument.js` file is included in the `FramesetFragment`, `RequestorFragment`, and the `RunFragment`, corresponding to each of the composite Web Viewers, which are constructed based on the selected servlet mapping. In all cases, the `BirtReportDocument` class is initialized, setting up the event handlers.

The following list of directory and files is in the AJAX directory of the Web Viewer:

- Core
 - Contains JavaScript files used to handle BIRT events and communications with the Web Viewer servlets.
 - `BirtCommunicationManager.js`

- Handles most Ajax.Request calls to the Web Viewer servlet or SOAPAction
 - **BirtDndManager.js**
Handle drag-and-drop for Web Viewer dialogs
 - **BirtEvent.js**
Prototype BIRT Web Viewer Events
 - **BirtEventDispatcher.js**
Supplies functions for registering, firing, and broadcasting BIRT events
 - **BirtSoapRequest.js**
Supplies functions used to create SOAP messages to be sent by the BirtCommunicationManager
 - **BirtSoapResponse.js**
Processes the SOAP responses from the SOAP request
 - **Mask.js**
Conceals a button in toolbar when dialogs are presented
- **Lib**
Contains the prototype.js library. Prototype.js is the prototype JavaScript framework file.
- **Mh**
Contains files used to parse the SOAP responses, from the AJAX servlet calls.
 - **BirtBaseResponseHandler.js**
Base class used to add associations for SOAP responses. Currently there is only one.
 - **BirtGetUpdatedObjectsResponseHandler.js**
Main class used to consume the SOAP responses. This class has process methods for updating entire report sections, such as a new report page and for updating portions of the Web Viewer, for example, the page number in the navigation bar.
- **Ui**
Contains files used in conjunction with the JSP fragments to control the Web Viewer user interface.
 - **App**
 - **AbstractBaseToc.js**
This class is used as the base class for the BirtToc.js.

- ❑ **AbstractBaseToolBar.js**
This class is the base class for the BirtToolbar.js.
 - ❑ **AbstractUIComponent.js**
The base class for all classes in the directory. Sets the instance of the class.
 - ❑ **BirtNavigationBar.js**
This JavaScript class contains methods that work in conjunction with the NavigationbarFragment.jsp to create the links to do page navigation. The NavigationbarFragment constructs the initial structure using INPUT tags and the BirtNavigationBar sets up native event handlers to monitor clicking or entering the navigation bar controls, which fire BIRT events to retrieve new content.
 - ❑ **BirtProgressBar.js**
The BirtProgressBar class is used when displaying the progress bar and handles canceling of the currently running task.
 - ❑ **BirtToc.js**
This JavaScript class sets up the events that retrieve the table of contents for a report document. In addition, event handlers load the correct pages when a table of contents node is selected.
 - ❑ **BirtToolbar.js**
The BirtToolbar JavaScript class works with the ToolbarFragment.jsp in a similar nature as the BirtNavigationBar works with the NavigationbarFragment.jsp. One difference is that some of the BIRT events make HTTP requests to the Web Viewer servlets as opposed to SOAP requests. For example, the _E_EXPORT_REPORT event launches the export dialog, which then creates a form post that is submitted to the Web Viewer. Many of the events registered in the BirtReportDocument class are fired here.
- **Dialog**
 - ❑ **AbstractBaseDialog.js**
This JavaScript class is the base class for all the dialogs in this directory and contains the event handlers for basic dialog operation.
 - ❑ **AbstractExceptionDialog.js**
Base Class for the BirtExceptionDialog used with the ExceptionDialogFragment.jsp page to add exceptions to the Web Viewer. The general exception is written to the element with faultstring as the id in ExceptionDialogFragment.jsp and the stack trace is written to the element with faultdetail as the

`id`. The writing occurs while the SOAP response message is being parsed by the `BirtSoapResponse` JavaScript class.

- ❑ `AbstractParameterDialog.js`

The base class for the `BirtParameterDialog` JavaScript class.

- ❑ `BirtConfirmationDialog.js`

This class is used to display a confirmation dialog used within the `BirtPrintReportDialog`.

- ❑ `BirtExceptiondialog.js`

This class extends the `AbstractExceptionDialog` to set up event handlers that show or hide the stack trace when the error detail is selected. Do not confuse these stack trace messages with report errors, which the HTML emitter writes and displays at the bottom of the report.

- ❑ `BirtExportReportDialog.js`

This class works with the `ExportReportDialogFragment.jsp` page to build the Export Report dialog. This class creates a form post to the Web Viewer servlet to download the report in one of the supported formats. The `BirtReportDocument.js` file contains the class that registers an event handler to call this class. The event is fired in the `toolbar.js` file when a user clicks on the appropriate image as defined in the `ToolbarFragment.jsp`.

- ❑ `BirtParameterDialog.js`

This class is responsible for displaying the Parameter page, collecting the parameter values, making calls for dynamic and cascading parameters, and submitting the URL to re-run the report.

- ❑ `BirtPrintReportDialog.js`

This class works with the `PrintReportDialogFragment.jsp` to create the print report dialog used to print the report on the server. This class works similar to the `BirtExportReportDialog`.

- ❑ `BirtPrintReportServerDialog.js`

This class works with the `PrintReportServerDialogFragment.jsp` file to handle server-side printing.

- ❑ `BirtSimpleExportDataDialog.js`

This class works with the `SimpleExportDataDialogFragment.jsp` to construct the Export Data dialog. This class creates a form post to the download mapping to download the report data and works similar to the `BirtExportReportDialog`.

- ❑ `BirtTabbedDialogBase.js`

This class is currently not used.

- report
 - AbstractBaseReportDocument.js

Base class for the BirtReportDocument class. Many of the event handlers for the BirtReportDocument class are implemented in this class.
 - AbstractReportComponent.js

Base class for the AbstractBaseReportDocument class. Provides framework functions for re-rendering content and setting up events.
 - BirtReportDocument.js

Handles registering most of the event handlers and is included in the FramesetFragment.jsp, RequestorFragment.jsp, and the RunFragment.jsp.
 - ReportComponentIdRegistry.js

Utility class used for id lookups.
- utility
 - BirtPosition.js

Utility class with functions for handling moving and sizing dialogs.
 - BirtUtility.js

Class containing many reused utility functions, such as functions for enabling and disabling buttons, trimming data, and message formatting.
 - BrowserUtility.js

Utility class with functions used when determining if output should be set up for Internet Explorer.
 - Constants.js

Class containing many of the constants used with the BIRT Web Viewer.
 - Debug.js

Utility class used for debugging the example Web Viewer. Can be launched by including the Debug.js and then calling the shoDebug function.
 - Printer.js

Utility class containing functions for printer manipulation.

Using the Web Viewer Deployment wizard

Using portions of the Web Tools Platform (WTP), BIRT 2.2.1 now supplies a web project. This project type automatically deploys the example Web Viewer to the selected application server. In addition, the project type deploys the new BIRT tag library.

To use this project type, select a new project, and under the Business Intelligence and Reporting Tools category, select Web Project. Figure 5-16 shows the New Project wizard.

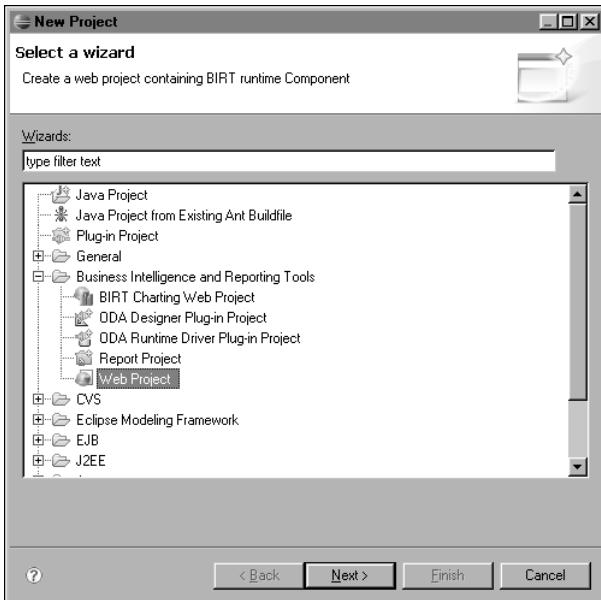


Figure 5-16 Selecting a new web project

After selecting a new web project, the wizard dialogs assist in creating a new project or adding the Web Viewer to an existing enterprise application as shown in Figure 5-17.

After specifying the Target Runtime, the Web Project wizard displays a series of configuration dialogs as shown in Figure 5-18. The web.xml settings section discusses these web configuration options. For more information about using the BIRT tag library, see "Web Viewer tag library," later in this chapter.

To test that the Web Viewer is deployed correctly, right-click the index.jsp file located under the content directory and choose Debug As->Debug on Server, as shown in Figure 5-19.

On the standard Web Viewer test page, select the View Example link to run the test report.

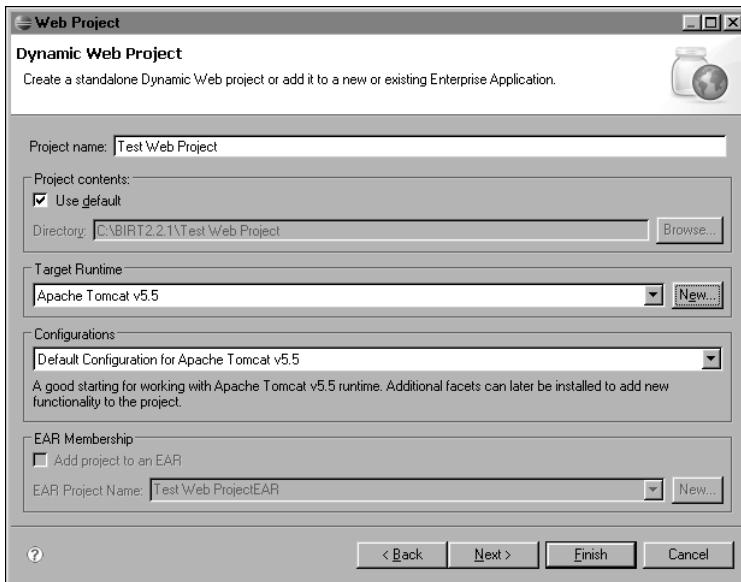


Figure 5-17 Creating a new web project

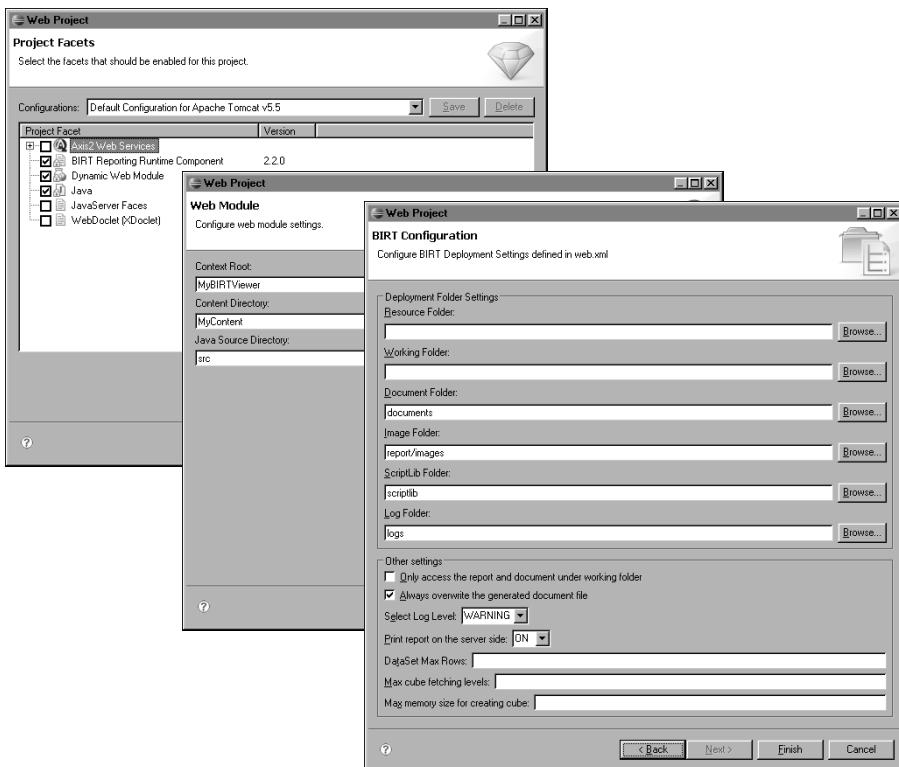


Figure 5-18 Configuring a new web project

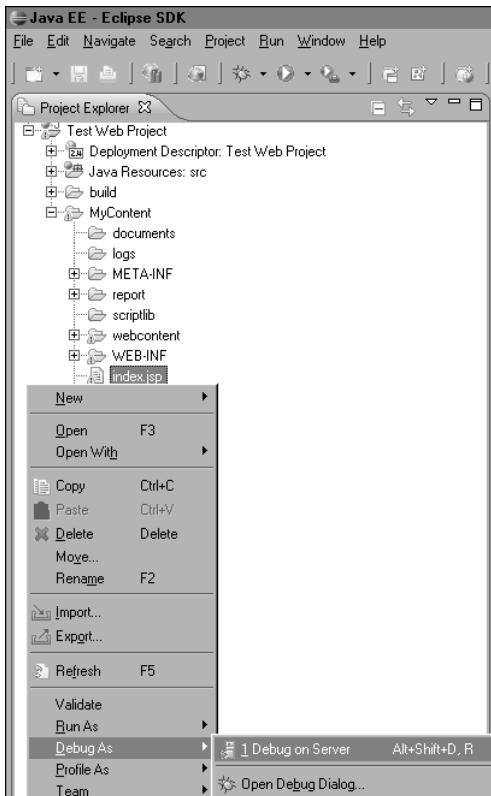


Figure 5-19 Debugging a web project on the server

Web Viewer tag library

The Web Viewer contains a tag library that can be used to customize the behavior of the Web Viewer. This tag library can be deployed by either deploying the Web Viewer manually or using the BIRT Web Deployment project wizard.

If you want to have BIRT deployed in one context and include the tag library in a separate context, copy the birt.tld file to the WEB-INF\tlds directory and copy coreapi.jar, modelapi.jar, viewerservlets.jar, and com.ibm.icu_version.jar from the Web Viewer libs directory to the new context/web-inf/lib directory. Add the following code reference to web.xml:

```
<jsp-config>
  <taglib>
    <taglib-uri>
      /birt.tld
    </taglib-uri>
    <taglib-location>
      /WEB-INF/tlds/birt.tld
    </taglib-location>
```

```
</taglib>
</jsp-config>
```

Using this deployment references reports in relation to the Web Viewer not the new context.

The Web Viewer tag library contains the following five tags:

- param
- paramDef
- parameterPage
- report
- viewer

Each tag has multiple attributes that control the behavior of the BIRT Web Viewer.

The viewer tag displays the standard Web Viewer. The viewer tag calls the frameset or the run mapping of the Web Viewer servlet-based on the selection in the pattern attribute. Using the frameset pattern creates a report document and using the run pattern does not create a report document.

The code in Listing 5-2 creates an iframe in the JSP page that calls the standard BIRT Web Viewer using the frameset mapping and formats the report as HTML.

Listing 5-2 Using the viewer tag to create an iframe in the JSP page

```
<%@ page language="java" contentType="text/html;
    charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/birt.tld" prefix="birt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=ISO-8859-1">
        <title>Insert title here</title>
    </head>
    <body>
        <birt:viewer id="birtViewer"
            reportDesign="TopNPercent.rptdesign"
            pattern="frameset"
            height="450"
            width="700"
            format="html">
        </birt:viewer>
    </body>
</html>
```

Table 5-4 describes the viewer tag attributes.

The last four attributes affect the presentation of the Web Viewer.

Table 5-4 Web Viewer tag attributes

Attribute	Description
baseURL	The baseURL determines the location of the Viewer application. If the tags are used in the same context as the Web Viewer, this attribute is not required. If the tag library is used in a separate context, but in the same application server, this setting may contain a value such as baseURL= "/WebViewerExample".
bookmark	Specifies which bookmark to load within the report. For example, adding a table of contents and specifying the table of contents entry will load the page containing that entry.
forceOverwrite Document	Specifies whether the created report document is overwritten. Only valid with the frameset mapping.
format	Specifies the output format, such as PDF, HTML, or XLS.
frameborder	Specifies whether or not to display a border around the iframe. Valid values are yes or no. If isHostPage is true, this value is ignored.
height	Sets the Height of the iframe in pixels. If isHostPage is true, this value is ignored.
id	An unique identifier for the Web Viewer.
isHostPage	If this value is set to true the viewer tag will occupy the entire page. The default value is false, which allows multiple reports to be contained in one JSP Page.
left	Sets the left of the iframe in pixels. If isHostPage is true, this value is ignored.
locale	Specifies the locale for the report.
pageNum	Displays a specific page within the report.
pageRange	Displays a specific page range within the report.
pattern	The viewer tag supports either run or frameset, which matches the standard viewer servlet mappings. frameset is used by default.
position	Sets the iframe position style attribute. Valid values are static, absolute, relative, and fixed. If isHostPage is true, these values are ignored.
reportDesign	Specifies the name of the report design file. This setting can be relative or set to a full path or a URL.

(continues)

Table 5-4 Web Viewer tag attributes (*continued*)

Attribute	Description
reportDocument	Sets the name of the report document file. This setting can be relative or set to a full path, or a URL. If using a URL, the setting must be a file URL, //.
reportletId	Specifies the instance id of the portion of the report to be displayed. Note this only works with the run pattern and a reportDocument setting.
resourceFolder	Specifies the resource folder, which contains libraries and images. Usually specified in web.xml, but this parameter will override that value.
rtl	Sets the right-to-left flag. Default is false.
scrolling	Sets the iframe scrolling style attribute. Valid values are auto, yes, and no. If isHostPage is true, these values are ignored.
showNavigationBar	When using the viewer tag with the frameset pattern, this setting determines if the navigation bar displays.
showParameterPage	When using the viewer tag, this setting determines if the Parameter page displays.
showTitle	When using the viewer tag with the frameset pattern, this setting determines if the report title displays.
showToolBar	When using the viewer tag with the frameset pattern, this setting determines if the toolbar displays.
style	Sets the style for the report container. If isHostPage is true, this value is ignored.
svg	Specifies where SVG for charts is supported.
title	Sets the title for the report container page.
top	Sets the top of the iframe in pixels. If isHostPage is true, this value is ignored.
width	Sets the width of the iframe in pixels. If isHostPage is true, this value is ignored.

Figure 5-20 illustrates using these settings.

The report tag renders the report without the AJAX framework and delivers the content in an iframe or a div element. Using this tag calls the Web Viewer preview mapping and does not create a report document. If you specify a report document attribute, the code renders the report from the specified document.

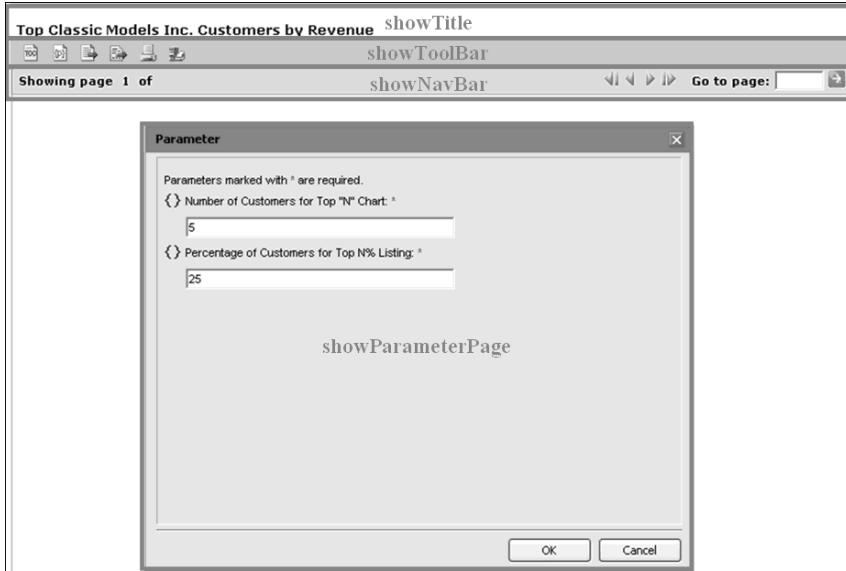


Figure 5-20 Using the viewer tag with the frameset pattern

In Listing 5-3, the code renders a page of myrptdoc.rptdocument in an iframe container.

Listing 5-3 Using the report tag to render a page

```
<%@ page language="java" contentType="text/html;
    charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/birt.tld" prefix="birt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=ISO-8859-1">
    </head>
    <body>
        <birt:report id="birtViewer"
            reportDesign="multipage.rptdesign"
            height="600"
            width="800"
            format="html"
            reportContainer="iframe"
            isHostPage="false"
            reportDocument="c:/temp/myrptdoc.rptdocument"
            pageNum="2"
            showParameterPage="true">
        </birt:report>
    </body>
</html>
```

Table 5-5 describes the report tag attributes.

Table 5-5 Attributes for the report tag

Attribute	Description
baseURL	The baseURL determines the location of the Web Viewer application. If the tags have the same context as the Web Viewer, this attribute is not required. If the tag library has a separate context, but is in the same application server, this setting can contain a value such as baseURL= "/WebViewerExample". The reportContainer attribute must be set to iframe if this attribute is used.
bookmark	Specifies which bookmark to load within the report. For example, adding a table of contents and specifying the table of contents entry loads the page containing that entry.
format	Specifies the output format, such as PDF, HTML, or XLS.
frameborder	Specifies whether or not to display a border around the iframe. Valid values are yes or no. If isHostPage is true, this value is ignored. reportContainer must be set to iframe.
height	Sets the Height of the iframe in pixels. If isHostPage is true, this value is ignored.
id	A unique identifier for the Web Viewer.
isHostPage	If this value is set to true, the viewer tag occupies the entire page. The default value is false, which allows multiple reports to be contained in one JSP page.
left	Sets the left of the iframe in pixels. If isHostPage is true, this value is ignored.
locale	Specifies the locale for the report.
pageNum	Displays a specific page within the report. The report document must already exist.
pageRange	Displays a specific page range within the report. For example, 2-5, 12. The report document must already exist.
position	Sets the div and iframe position style attribute. Valid values are static, absolute, relative, and fixed. If isHostPage is true, these values are ignored.
reportContainer	Specifies if the report is rendered in an iframe or a div element. This attribute affects other attributes.

Table 5-5 Attributes for the report tag (*continued*)

Attribute	Description
reportDesign	Specifies the name of the report design file. This setting can be relative or set to a full path or a URL.
reportDocument	Sets the name of the report document file. This setting can be relative or set to a full path or a URL. If using a URL, it must be a file URL, // . Note that a report document is not created, but, if the report document exists, it is rendered.
reportletId	Specifies the instance id of the portion of the report to be displayed. Note this only works with the run pattern and a report document setting.
resourceFolder	Specifies the resource folder, which contains libraries and images. Usually specified in web.xml, but this parameter overrides that value.
rtl	Sets the right to left flag. Default is false.
scrolling	Sets the div and iframe scrolling style attribute. Valid values are auto, yes, and no. If isHostPage is true, these values are ignored.
showParameterPage	When using the report tag, this setting determines if the Parameter page is displayed. If reportContainer is set to div, this setting is ignored and the Parameter page is not displayed.
style	Sets the style for the report container. If isHostPage is true, this value is ignored.
svg	Specifies where SVG for charts is supported.
top	Sets the top of the iframe in pixels. If isHostPage is true, this value is ignored.
width	Sets the Width of the iframe in pixels. If isHostPage is true, this value is ignored.

The param tag can be used to set a parameter value or display text when using the report or viewer tags. This tag is useful when showParameterPage is set to false, but you do not wish to use report parameter defaults. This tag is also needed if you use the report tag and specify a div element as the report container, which does not display the default Parameter page.

In Listing 5-4, the code uses a param tag to set a parameter value. In the example, the JSP runs the TopNPercent report, passing the "Top Count" parameter a value. The name tag must match the report parameter name.

Listing 5-4 Using a param tag to set a parameter value

```
<%@ page language="java" contentType="text/html;
    charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/birt.tld" prefix="birt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=ISO-8859-1">
    </head>
    <body>
        <birt:report id="birtViewer"
            reportDesign="TopNPercent.rptdesign"
            height="600"
            width="800"
            format="html"
            reportContainer="div"
            isHostPage="false">
            <birt:param name="Top Count" value="3"></birt:param>
        </birt:report>
    </body>
</html>
```

Table 5-6 describes the param tag attributes.

Table 5-6 Attributes for the param tag

Attribute	Description
displaytext	Sets the display text for the parameter.
id	A unique identifier for the Web Viewer.
isLocale	Specifies whether the report parameter value is a locale or format-related string. Valid values are true or false.
name	Specifies the report parameter name, which must match the design file.
pattern	Specifies the report parameter pattern format. If isLocale is false, this value is ignored.
value	Sets the value for the report parameter. If this value is left blank, the default value for the parameter is used.

The parameterPage tag enters parameters before running a report. The tag can display the default BIRT Parameter page or display a custom Parameter page. The isCustom attribute determines which page to use. If this attribute is false or does not exist the default Parameter page is used.

Most tags launch the Parameter page by default, so this tag is often used when creating your own Parameter page. The default action after the report parameters are collected is to use the frameset mapping, which can also be set to preview or run, and is defined in the pattern attribute.

In Listing 5-5, the JSP page shows the Parameter page and runs the frameset mapping.

Listing 5-5 Using a parameterPage tag to set parameter values

```
<%@ page language="java" contentType="text/html;
    charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/birt.tld" prefix="birt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=ISO-8859-1">
    </head>
    <body>
        <birt:parameterPage id="birtParmPage"
            reportDesign="TopNPercent.rptdesign"
            pattern="frameset"
            height="600"
            width="800"
            format="html"
            title="My Viewer Tag"
            showTitle="true"
            showToolBar="true"
            showNavigationBar="true">
        </birt:parameterPage>
    </body>
</html>
```

In Listing 5-6, using the JSP page displays the parameters inside the parameterPage tag, and the Submit button runs the Web Viewer frameset mapping.

The names in the input tags for the parameters must match the design file parameter names. All required parameters must exist or the default Parameter page is displayed. If the isCustom attribute is true, you must use the name attribute. For more options when creating a custom Parameter page, see the paramDef tag.

Listing 5-6 Using a parameterPage tag

```
<%@ page language="java" contentType="text/html;
    charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/birt.tld" prefix="birt" %>
```

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=ISO-8859-1">
  </head>
  <body>
    <birt:parameterPage id="birtParmPage"
      reportDesign="TopNPercent.rptdesign"
      name="my form"
      pattern="frameset"
      height="600"
      width="800"
      format="html"
      title="My Viewer Tag"
      isCustom="true"
      showTitle="true"
      showToolBar="true"
      showNavigationBar="true">
      TOP COUNT PARAMETER
      <input type="Text" name="Top Count">
      <br><br>
      TOP PERCENT PARAMETER
      <input type="Text" name="Top Percentage">
      <br><br>
      <input type="Submit" value="Run Report">
    </birt:parameterPage>
  </body>
</html>

```

Table 5-7 shows the parameterPage tag attributes.

Table 5-7 Attributes for the parameterPage tag

Attribute	Description
baseURL	The baseURL is used to determine the location of the Web Viewer application. If the tags are used in the same context as the Web Viewer, this attribute is not required. If the tag library is used in a separate context, but in the same application server, this setting may contain a value such as baseURL="/WebViewerExample".
bookmark	Specifies which bookmark to load within the report. For example, adding a table of contents and specifying the table of contents entry loads the page containing that entry.
height	Sets the Height of the iframe in pixels.
id	A unique identifier for the Web Viewer.

Table 5-7 Attributes for the parameterPage tag (*continued*)

Attribute	Description
isCustom	Indicates whether the default BIRT Parameter page or a custom page is used. Valid values are true and false.
forceOverwrite Document	Specifies whether the report document that is created is overwritten. Only valid with the frameset mapping.
format	Specifies the output format, such as PDF, HTML, or XLS.
frameborder	Specifies whether or not to display a border around the iframe. Valid values are yes or no. If isCustom is true, this value is ignored.
left	Sets the left of the iframe in pixels.
locale	Specifies the locale for the report.
name	Specifies the Report Parameter page name. This attribute is used to create a form and is required if using the isCustom attribute. This attribute must be unique.
pattern	The parameterPage tag supports run, frameset, and preview mappings. frameset is the default.
position	Sets the iframe position style attribute. Valid values are static, absolute, relative and fixed.
reportDesign	Specifies the name of the report design file. This setting can be relative, set to a full path, or a URL.
reportDocument	Sets the name of the report document file. This setting can be relative, set to a full path, or a URL. If using a URL, the setting must be a file URL, //.
reportletId	Specifies the instance id of the portion of the report to be displayed. Note this attribute only works with the run pattern and report document settings.
resourceFolder	Specifies the resource folder, which contains libraries and images. Typically specified in web.xml, but this parameter overrides that value.
rtl	Sets the right-to-left flag. The default is false.
scrolling	Sets the iframe scrolling style attribute. Valid values are auto, yes, and no.
showNavigationBar	When using the viewer tag with the frameset pattern, this setting specifies whether to display the navigation bar.

(continues)

Table 5-7 Attributes for the parameterPage tag (*continued*)

Attribute	Description
showTitle	When using the viewer tag with the frameset pattern, this setting determines whether to display the report title.
showToolBar	When using the viewer tag with the frameset pattern, this setting specifies whether to display the toolbar.
style	Sets the style for the report container.
svg	Specifies location of support for SVG charts.
target	Specifies the target window for the form submit. For example, _blank or parent.
title	Sets the title for the report container page.
top	Sets the top of the iframe in pixels.
width	Sets the Width of the iframe in pixels.

You can use the parameterPage to construct a parameter form, but this configuration has limited functionality when discovering parameter control types like check boxes, radio, and dynamic parameters. The paramDef tag addresses these issues and must be used within a parameterPage tag with the isCustom attribute set to true.

Using the paramDef tag allows the control type to be determined automatically based on the report design. The HTML returned for each parameter can then be integrated into your custom Parameter page as needed. This tag includes using dynamic and cascaded parameters, which call BIRT data sets to populate the control. You can manipulate the returned HTML using the cssClass and style tag attributes. Listing 5-7 shows the code for an example JSP page that illustrates how to construct a Parameter page for a report that contains various parameter types.

Listing 5-7 Constructing a Parameter page for a report

```
<%@ page language="java" contentType="text/html;
    charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/birt.tld" prefix="birt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=ISO-8859-1">
        <title>Insert title here</title>
        <style type="text/css">
            .class1 { color:#ff0000; }
        </style>
    </head>
```

```

<body>
    Parameter Page
    <br>
    <birt:parameterPage
        id="report1"
        name="page1"
        reportDesign="testtag.rptdesign"
        isCustom="true"
        pattern="preview">
        Text Parameter:
        <birt:paramDef id="1" name="text"
            cssClass="cclass1"/>
        <br><br>
        Radio Parameter:
        <birt:paramDef id="2" name="radio1" value="value1"
            displayText="myradiotext" isLocale="true" />
        <br><br>
        Check Parameter:
        <birt:paramDef id="3" name="check" value="true"
            title="this is a check field"/>
        <br><br>
        CommList Parameter:
        <birt:paramDef id="4" name="CommList"
            title="this is a select field"/>
        <br><br>
        Cascading Parameter1:
        <birt:paramDef id="5" value="cascadeone"
            displayText="myfirstselection"
            name="cas1" title="this is a select field"/>
        <br><br>
        Cascading Parameter2:
        <birt:paramDef id="6" name="cas2"/>
        <br><br>
        Cascading Parameter3:
        <birt:paramDef id="7" name="cas3"/>
        <br><br>
        Hide Parameter:
        <birt:paramDef id="8" name="hide" value="abcdef"/>
        <br><br>
        <input type="submit" name="submit"
            value="Submit form"/>
        <br><br>
    </birt:parameterPage>
</body>
</html>

```

Table 5-8 describes the paramDef tag attributes.

Table 5-8 Attributes for the paramDef tag

Attribute	Description
cssClass	Sets the CSS class attribute of the parameter control.
displaytext	Set the display text for the parameter.

(continues)

Table 5-8 Attributes for the paramDef tag (*continued*)

Attribute	Description
id	A unique identifier for the parameter control.
isCustom	Indicates whether the default BIRT Parameter page or a custom page is used. Valid values are true and false.
isLocale	Specifies whether the report parameter value is a locale/format-related string. Valid values are true or false.
name	Specifies the report parameter name. This must match the parameter name in the report.
pattern	Specifies the report parameter pattern format. If isLocale is false, this value is ignored.
style	Specifies the style for the parameter control.
title	Specifies the title attribute of the parameter control.
value	Sets the value for the report parameter, using the default value if blank.

Web Viewer RCP deployment

The sample Web Viewer project also contains a utility class that allows the Web Viewer to be used in plug-in format. When used this way, the Web Viewer deploys using the org.eclipse.tomcat plug-in.

Previewing reports within the BIRT designer uses this utility class. RCP developers can make use of this plug-in to embed report generation and viewing capabilities within their applications.

To deploy BIRT this way, first modify your plug-in dependencies to include the plug-in, org.eclipse.birt.report.viewer.

The Web Viewer depends on several BIRT and Eclipse plug-ins, such as the Tomcat plug-in. The required plug-ins depend on what features are used within the report design. To allow all features, verify that the entire set of BIRT run-time plug-ins are available to your application. These plug-ins are located in the run-time download, such as birt-runtime-2_2_1
\ReportEngine\plug-in.

Next, you need to add code to your application to run and render the report.

The WebViewer utility class supplies several variants of a method named display, which you can use to run and render report content, as shown in Listing 5-8.

The first variant simply calls the second variant with the allowPage parameter set to true. The second variant takes allowPage as a boolean parameter.

Listing 5-8 Variants of display method using allowPage

```
public static void display( String report, String format )
public static void display( String report, String format,
    boolean allowPage )
```

If the allowPage parameter is true, the servlet frameset mapping is used. If the parameter is false, the run servlet mapping is used. As described in the Web Viewer servlets section, the frameset mapping separates running and rendering of the report and supports operations such as table of contents, export to CSV, and paginated HTML. The run mapping combines running and rendering of the report.

In both cases, these operations assume that the output format is set to any format other than PDF. PDF format is an option that automatically uses the run mapping.

Another variant of the display method allows several URL parameters to be specified in a map, as shown in Listing 5-9.

Listing 5-9 Variant of display method specifying parameters in a map

```
public static void display( String report, Map params )
```

Listing 5-10 shows an example of setting the values for a list of available parameters.

Listing 5-10 Setting the values for a list of available parameters

```
HashMap myparms = new HashMap();
myparms.put( "SERVLET_NAME_KEY", "preview" );
myparms.put( "FORMAT_KEY", "html" );
myparms.put( "RESOURCE_FOLDER_KEY", "c:/myresources" );
myparms.put( "ALLOW_PAGE", true );
myparms.put( "MAX_ROWS_KEY", "500" );
```

Using display as described in these variant methods renders the report in a separate browser window, which may not be desirable. You can render the BIRT reports to a Standard Widget Toolkit (SWT) browser composite.

Listing 5-11 shows two variants of display that take a browser as a parameter. You can add the browser to a view part to render the report to this browser.

Listing 5-11 Variants of display that take a browser as a parameter

```
public static void display( String report, String format,
    Browser browser, String servletName )
public static void display( String report, Browser browser, Map
    params )
```

The servletName parameter accepts run, preview, or frameset, which allows access to the full capabilities of the Web Viewer. The last variant accepts the same parameters in a Map with the exception that the report renders to Browser composite.

The Web Viewer allows entering parameters. In BIRT 2.2.1, if your report definition has parameters that are required and no default value is specified, a parameter entry dialog displays before running the report.

Passing a web context object to the Web Viewer

BIRT uses an application context map to store values and objects for use in all phases of report generation and presentation. You can reference objects in the application context from script, the BIRT Expression Builder, the ODA layer, and so forth. You can modify the application context before calling the Web Viewer.

To add a specific object to the application context for use within the Web Viewer requires that you set particular request attributes before calling the Web Viewer. These attributes are `AppContextKey` and `AppContextValue`.

The `AppContextKey` is the name used to reference the object. The `AppContextValue` is the object added to application context. Listing 5-12 shows an example JSP page that adds these attributes to the request. The example uses a `String` object, but this object can be any object in your application.

Listing 5-12 Adding `AppContextKey` and `AppContextValue` to a request

```
<% java.lang.String teststr = "MyTest";
   request.setAttribute( "AppContextKey", teststr );
   java.lang.String string0bj =
      "This test my Application Context From the Web Viewer";
   request.setAttribute( "AppContextValue", string0bj );
%>
<jsp:forward page=
   "<%= "/run?__report=AppContext.rptdesign" %>"/>
```

The `MyTest` object is now available to the context. You can reference this object in the BIRT expression builder using the following expression:

```
MyTest.toString( );
```

If this expression is tied to a BIRT data element, the following text displays in the report:

```
"This test my Application Context From the Web Viewer"
```

You can also achieve this result using simple session values that BIRT can access. Using the application context is helpful when objects need to be passed to the ODA layer.

Building the Web Viewer

The sample Web Viewer often requires changes to the Web Viewer code, which is possible with BIRT. The following list walks through the steps required to rebuild the Web Viewer. Check the FAQ in the BIRT wiki for revised instructions.

- 1** Install BIRT prerequisites in Eclipse home. Alternatively, use the BIRT all-in-one download, which contains everything you need.
- 2** Download or import BIRT source code into your workspace. See the wiki for details.
- 3** Make the required changes to the Web Viewer.
- 4** Right click on the MANIFEST.MF file, located in the org.eclipse.birt.report.viewer plug-in, and select PDE Tools->Create Ant Build File.
- 5** The PDE generates the build.xml file under the root folder of the org.eclipse.birt.report.viewer plug-in.
- 6** Right click on build.xml and select Run As->Ant Build to start the compile process.
- 7** When the build process finishes, you can find viewservlets.jar under birt/WEB-INF/lib. This JAR file is used with the Web Viewer when it is deployed as a J2EE application. Replace this JAR file in your existing deployment. You can find Web Viewer.jar under the root folder. This JAR file is used when the Web Viewer is used in a plug-in format.

This page intentionally left blank

II

Understanding the BIRT Framework



This page intentionally left blank

6

Understanding the BIRT Architecture

BIRT consists of many components that relate to one another in various ways. This chapter provides an overview of the BIRT architecture, the BIRT components, the Eclipse components upon which BIRT relies, and the relationships that tie them all together.

Understanding the BIRT integration

BIRT is an Eclipse project, which means that it is tightly integrated with Eclipse frameworks and platforms. Like all Eclipse projects, BIRT is implemented as a set of Eclipse plug-ins. The BIRT plug-ins provide the functionality for all BIRT components, including BIRT applications, the engines that drive the applications, and supporting application programming interfaces (APIs). The BIRT plug-ins also provide the interface mechanism for communicating with several Eclipse frameworks and platforms.

The relationships between BIRT and the Eclipse components are best viewed as a stack, where each tier in the stack depends upon, uses, and integrates with the tier below it. Figure 6-1 illustrates this stack of dependent tiers.

Figure 6-2 illustrates the various BIRT components and how they relate to one another. In this diagram, a component in a solid box is a standard BIRT component. A component in a dashed box is a custom component that a Java developer can provide. Some custom components are extensions of BIRT and other custom components are applications that use the BIRT APIs. A component in a dotted box is a standard BIRT component that the containing component uses. For example, because BIRT Report Designer uses the design

engine, the design engine appears in a dotted box within the box for BIRT Report Designer.

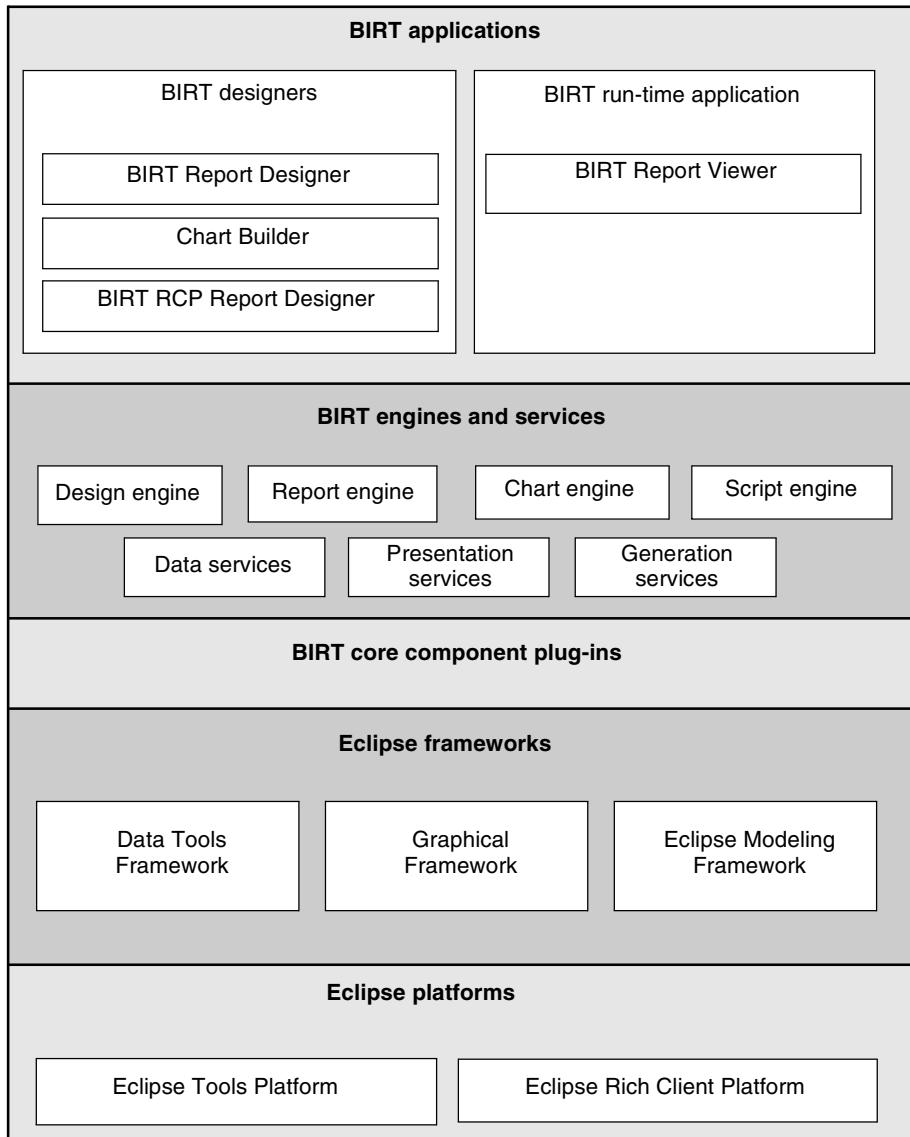


Figure 6-1 BIRT components as plug-ins to the Eclipse platform

BIRT Report Designer

Tools to create and edit report components and rules, including:

- Bookmarks
- Charts
- Data sources
- Data filters
- Data groups
- Data mapping rules
- Data sets
- Expressions
- Highlighting rules
- Hyperlinks
- Library components
- Master pages
- Properties
- Report layout
- Report parameters
- Report XML
- Scripts
- Styles

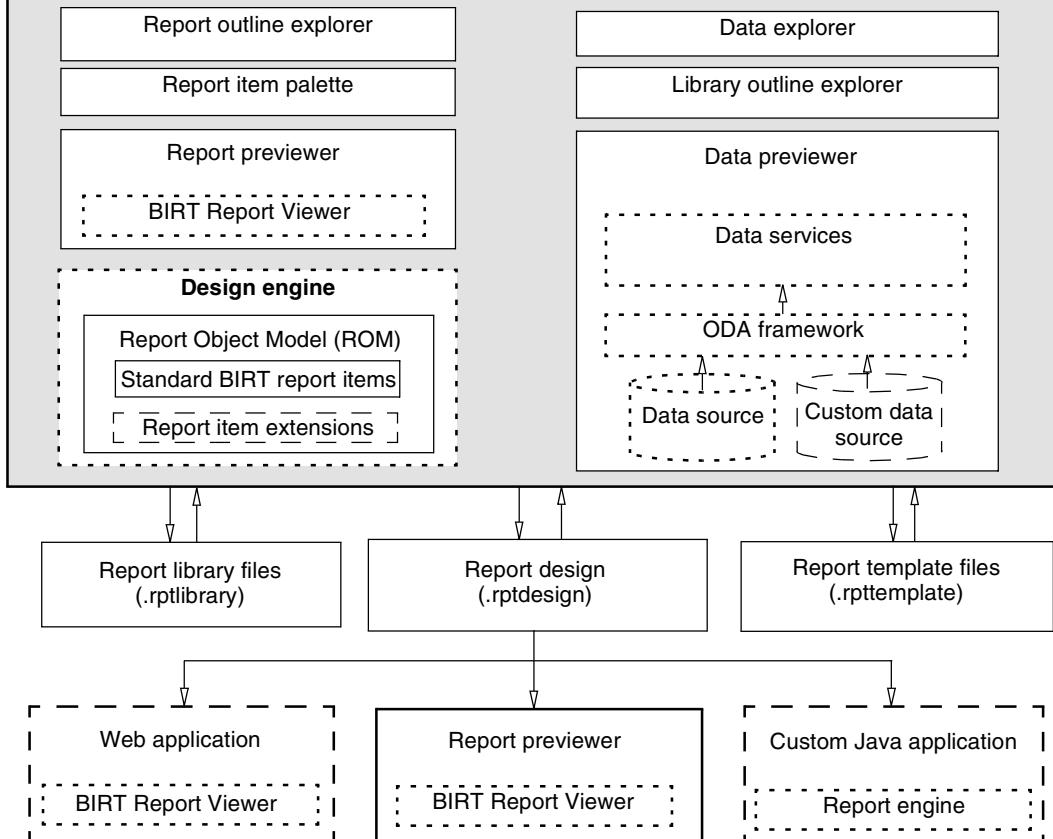


Figure 6-2 Relationships of standard BIRT components and custom components (*continues*)

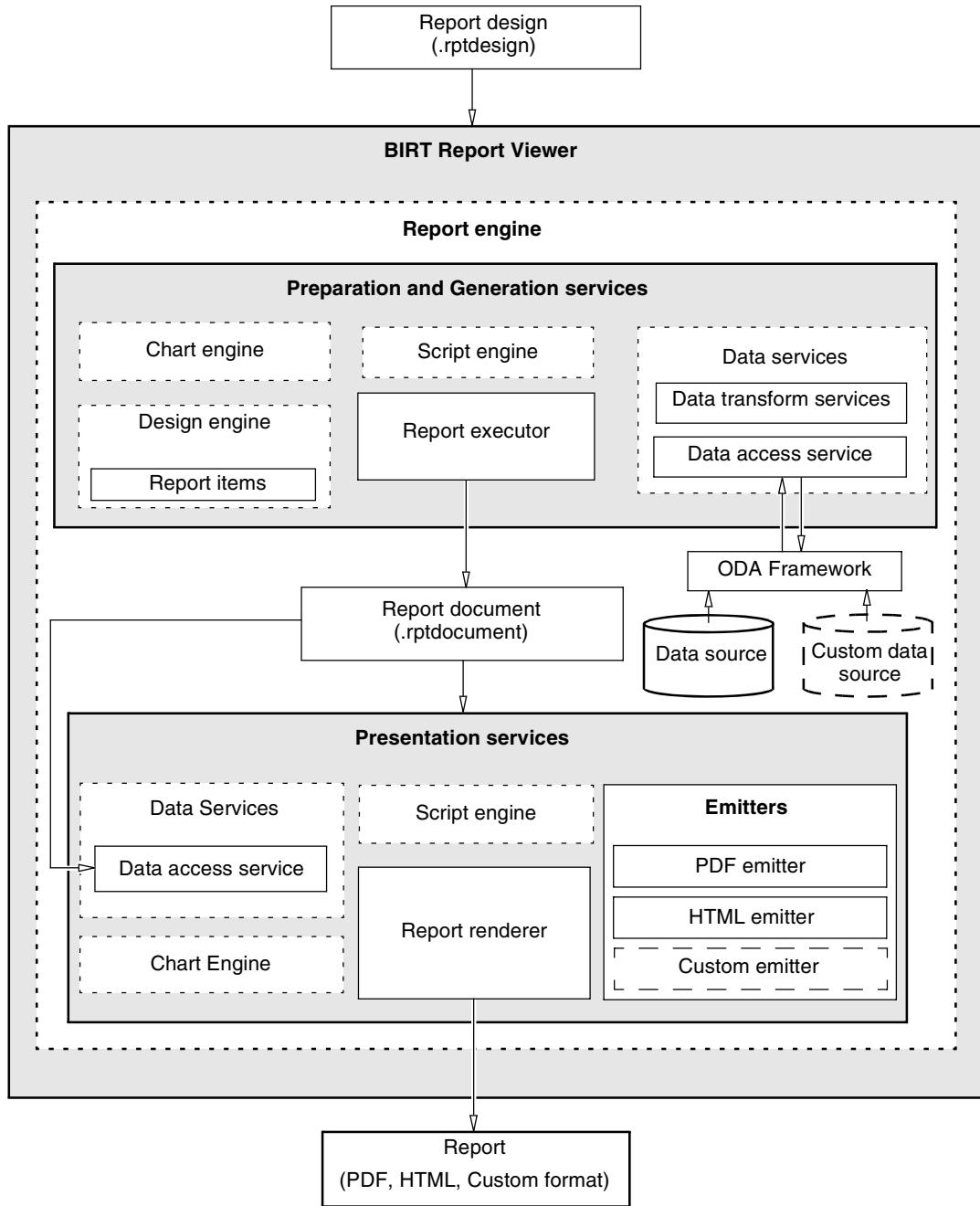


Figure 6-2 Relationships of standard BIRT components and custom components
(continued)

BIRT Report Designer provides drag-and-drop capabilities to quickly design reports. The report designer uses the report design engine to produce XML report design files. These report designs files are fed to the BIRT Report Engine, which at run-time fetches the appropriate data using queries defined at design-time. If the BIRT report is to be viewed immediately, then it is generated in memory and emitted in the desired output format, which can be non-paginated HTML, paginated HTML, PDF, WORD, POSTSCRIPT, PowerPoint Presentation, or Excel file. Otherwise, the report engine transforms and summarizes the data and caches the generated report for later viewing in an intermediate binary file, the report document file. This caching mechanism allows BIRT to scale to handle large quantities of data. The BIRT designer also has a tightly integrated charting component, which allows a variety of charts types to be included in reports. The charts are rendered at runtime using the charting engine. Figure 6-3 illustrates this architecture and process flow.

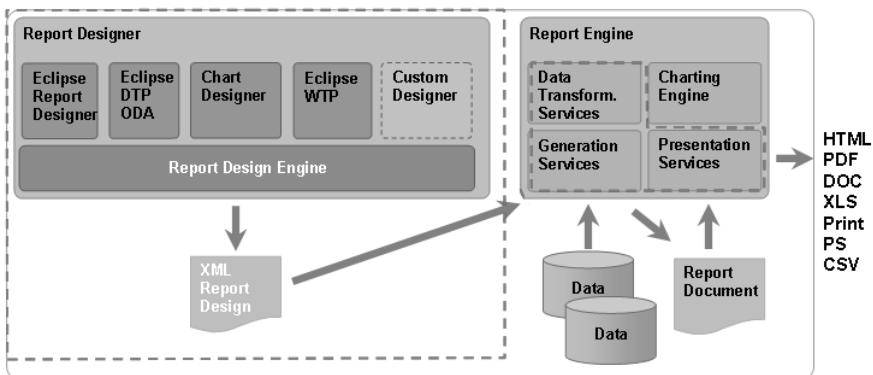


Figure 6-3 BIRT architecture and process flow

About the BIRT applications

There are three BIRT applications: BIRT Report Designer, BIRT RCP Report Designer, and BIRT Report Viewer. The two report designers are very similar. BIRT Report Designer runs as a set of Eclipse plug-ins and lets you build reports within the Eclipse workbench. BIRT RCP Report Designer has a simplified report design interface based on Eclipse RCP.

About BIRT Report Designer and BIRT RCP Report Designer

BIRT Report Designer is a graphical report design tool. BIRT Report Designer uses the report design engine to generate a report design file based on the ROM. ROM supports the standard set of BIRT report items and custom report items.

BIRT RCP Report Designer has a simplified report design interface based on Eclipse RCP. The primary functional differences between the BIRT RCP Report Designer and BIRT Report Designer are:

- BIRT RCP Report Designer has no integrated debugger.
- BIRT RCP Report Designer does not support Java event handlers.

Other than these differences, the functionality of the two report designers is identical and all further mentions of BIRT Report Designer in this chapter apply equally to BIRT RCP Report Designer.

BIRT Report Designer also supports the reuse of a report design by saving it as a template. You can also save individual report components in a component library, which is accessible to other report designs.

About the BIRT Viewer

BIRT provides the BIRT Viewer web application that includes all the necessary files to deploy to most J2EE application servers. This web application running reports and viewing paginated HTML, with a table of contents and bookmarks, and extracting data to a values file. URLs provide this level of integration. The host application forwards the reporting request to the BIRT Viewer and allows the user to interact directly with the report. The URL syntax within the viewer supports passing report parameters, setting report output formats, and specifying the locale. The directory location of BIRT reports can be configured in the BIRT Viewer. The BIRT engine is deployed once and as reports are completed, they are uploaded to the reports directory and made available to the end users, with the URL.

The BIRT Viewer is also available as an Eclipse plug-in. This plug-in is used within the report designer to preview and display a report while it is being developed. This plug-in can also be deployed to other Eclipse-based applications. Within BIRT Report Designer, it is deployed to and works with the Eclipse Tomcat application server plug-in. As report requests are made, the Tomcat application is started and the BIRT Viewer is launched. The major advantage of this approach is that no additional J2EE deployment is required and the reporting functionality is contained within the application.

About the BIRT engines and services

BIRT contains several engines, for example report design engine , report engine and chart engine. An engine is a set of Java APIs that provide basic functionality in a specific domain. These engines provide several different types of services. A service is a set of Java classes that provide functionality using the API provided from different engines. For example, the generation services use design engine API and report engine API to generate reports and produce report documents.

About the report design engine

The report design engine contains the APIs for validating and generating a report design file. The report design engine is used by BIRT Report Designer and by any custom Java application that generates a BIRT report design. The generation services also uses the report design engine when building the report document. The design engine contains APIs to validate the elements and structure of the design file against the ROM specification.

About the report engine

The BIRT Report Engine enables XML report designs created by the BIRT Report Designer to be embedded into a J2EE/Java application. To support this, the Report Engine provides two core services, generation and presentation.

The report engine supports extensions for custom report items, and for custom output formats. It also allows Java application developers to quickly integrate powerful report generation and viewing capabilities into their applications without having to build the infrastructure from lower level Java components.

The BIRT Report Engine application program interface (API) supports integrating the run-time part of BIRT into Java applications. The report engine provides the ability to specify parameters for a report, run a report to produce HTML, PDF, DOC, PS, or PPT output and fetch an image or chart.

About the generation services

The generation service within the Report Engine is responsible for connecting to the specified data sources, retrieving and processing the data such as sorting, grouping, and aggregations, creating the report layout and generating the report document. Report content can immediately be viewed using the presentation services, or saved for use later, permitting snapshot views of reports to be retained over time.

About the presentation services

The presentation services process the report document created by the generation services and produces the report in the format specified in the design. Like the generation services, the presentation services use the data transformation services. During the presentation stage, however, the data engine retrieves data from the report document rather than from a data source.

The presentation engine uses whichever report emitter it requires to generate a report in the requested format. BIRT has several standard emitters, HTML, PDF, DOC, PPT, PS and XLS. BIRT also supports custom emitters for formats.

Chart report items and custom report items extend the presentation engine to provide display capability for those items.

About the chart engine

The chart engine contains APIs for generating charts and associating them with data from a data source. Using the chart engine is not restricted to a BIRT application. Any Java application can use chart engine APIs to create and display a chart. The BIRT Report Viewer interprets any chart information in the report design and uses the chart engine to generate the chart.

About the data services

The data services contains the APIs to retrieve and transform data. When used by the generation engine, the data services retrieve data directly from the data source. When used by the presentation engine, the data services retrieve data from the report document.

About data services components

The data services consists of two primary components, the data access component and the data transform component. The data access component communicates with the ODA framework to retrieve data. The data transform component performs such operations as sorting, grouping, aggregating, and filtering the data returned from the data access component.

About the ODA framework

The ODA framework manages ODA and native drivers, loads drivers, opens connections, and manages data requests. The ODA framework is part of the Eclipse Data Tools Platform project. The ODA framework contains extension points through which you can add a custom ODA driver. The data engine extension provides the connection method and the driver for the data source. A custom ODA driver is necessary if you have a data source that BIRT does not support and a scripted data source is not desired. When you create a custom ODA driver you may need to extend not only the data engine but also BIRT Report Designer. A BIRT Report Designer extension is necessary if the data source requires a GUI component to specify the data set and data source properties.

About the types of BIRT report items

A report item is a visual component of a report, such as a label or a list or a chart. There are three categories of report items in BIRT, standard report items, custom report items, and the chart report item.

About standard report items

A report item is a visual component of a report. A report item can be as simple as a label or as complex as a 3D chart. Every report item has an icon on BIRT Report Designer Palette.

About custom report items

You can create new report items and you can extend an existing report item. An example of a simple extension to a report item is adding a property, such as color. An example of a new report item extension is the rotated text report item, which is a reference implementation of a report item extension.

Creating a new report item or extending an existing report item both involve extending BIRT through the Eclipse plug-in mechanism. Some custom items require an extension to a single component, while other custom items require extensions to multiple components. Depending on the report item, one or more of the following components may require an extension to support the new item:

- BIRT Report Designer
- The report design engine
- The report engine

About chart report items

A chart report item is a standard BIRT component, but it is implemented as a BIRT extension. The user interface for creating a chart report item is a chart builder that steps the report developer through the process of designing the chart and associating it with the appropriate database columns.

About the Report Object Model (ROM)

ROM is the model upon which BIRT is based. ROM is a specification for the structure, syntax, and semantics of the report design. The latest ROM specification appears in the following plug-in JAR file:

```
$INSTALL_DIR\ eclipse\plugins\  
org.eclipse.birt.report.model_version.jar
```

The formal expression of ROM is through an XML schema and a semantic definition file.

About the types of BIRT files

BIRT Report Designer uses four types of files:

- report design files
- report document files
- report library files
- report template files

The following sections provide a brief overview of each of these file types.

About report design files

A report design file is an XML file that contains the report design, the complete description of a BIRT report. The report design describes every aspect of a report, including its structure, format, data sources, data sets, and JavaScript event handler code. BIRT Report Designer creates the report design file and BIRT report engine processes it. The file extension of a report design file is .rptdesign.

About report document files

A report document file is a binary file that encapsulates the report design, incorporates the data, and contains additional information, such as data rows, pagination information, and table of contents information. The file extension of a report document file is .rptdocument.

About report library files

A report library file is an XML file that contains reusable and shareable BIRT report components. A report developer uses Library Explorer in BIRT Report Designer to manage access to a library.

A BIRT report library can contain any report element, for example:

- Data sets
- Data sources
- Embedded images
- Event handler code
- Styles
- Visual report items

The file extension of a report library file is .rptlibrary.

About report template files

A report template is an XML file that contains a reusable design. A report developer can use a template as a basis for developing a new report. A report developer uses a report template to maintain a consistent style across a set of report designs and for streamlining the report design process. A report template can specify many different elements of a report, including:

- One or more data sources
- One or more data sets
- Part or all of the layout of a report design, including grids, tables, lists, and other report items

- Grouping, filtering, and data binding definitions
- Styles
- Library components
- Master pages
- Cheat sheets

Report templates act as a starting point for report development. They speed up report development by capturing the layout of common types of reports. They also make it easy to create reports with a consistent look. Building BIRT templates is similar to building BIRT reports. The difference lies in converting report items into template report items which act as placeholders.

The file extension of a report template file is .rpttemplate.

About custom Java applications

Java developers can use the BIRT APIs to create a custom report designer or a custom report viewer.

About custom report designers

A custom report designer is a Java application that a Java developer creates to generate a well-formed report design file based on specific requirements. A custom report designer does not necessarily include a user interface. A typical example of a custom report designer is a Java application that dynamically determines the content, structure, or data source for a report, based on business logic. A custom report designer uses the same design engine API as BIRT Report Designer.

About custom Java report generators

A custom Java report generator performs the same function as the BIRT report generator, except that it is typically integrated into either a web application or a stand-alone Java application. Like the BIRT Report Viewer web application, a custom Java report generator uses the API of the report engine to read a report design file and generate a report. A custom Java report generator can use business logic to manage security issues, control content, and determine the output format.

About extensions to BIRT

Through its public APIs and the BIRT extension framework, BIRT enables a Java developer to expand the capabilities of BIRT. BIRT uses Eclipse extensions to allow developers to extend the functionality of the framework.

The extension points offered by BIRT allow the creation of new graph types, additional data sources, report controls, and emitters for rendering to additional outputs. This should appeal to users that have specialized data access and formatting needs. A list of possible custom extensions includes the following:

- A custom report item

A custom report item is a report item extension. This report item can be an extension, an existing BIRT report item, or a new report item.

- A custom ODA data source driver

A custom ODA data source driver is a custom ODA extension that connects to a data source type other than those that BIRT directly supports.

- A custom report emitter

A custom report emitter generates a report in a format other than HTML or PDF.

Understanding the Report Object Model

This chapter provides an overview of the BIRT Report Object Model (ROM) and the primary elements that comprise the model. ROM defines the rules for constructing a valid report design file in much the same way that HTML defines the rules for constructing a valid web page. ROM, therefore, is the model for the BIRT report design file in the same way that HTML is the model for the web page. For information about every component of ROM, see the online help entry at Help→Help Contents→BIRT Programmer Reference→Reference→Report Object Model (ROM) Definitions Reference.

About the ROM specification

The ROM specification defines a set of XML elements that describe the visual and non-visual components of a report. Visual components, known as report items, appear in a report, for example, data items, labels, and tables. ROM provides the framework for extended report items such as charts and cross tabs. Non-visual components support report items, but do not appear in a report, for example, data cubes, data sets, data sources, report parameters, and styles. The XML file that BIRT Report Designer generates to describe a report consists entirely of ROM elements. The ROM specification defines the elements, their properties, and an element's relationship to other elements. ROM elements describe:

- The data source and query with which to populate a report
- The placement, size, style, and structure of report items
- The report page layout

The report design file contains XML elements that describe the ROM elements that make up the report design. The BIRT design engine interprets the ROM elements using the ROM specification and the design.xsd file. This file is located at <http://www.eclipse.org/birt/2005/design> and also in the plug-in, org.eclipse.birt.report.model. BIRT Report Designer displays the elements that the design engine interprets. Visual report items appear in the layout window. Data-related items such as cubes, data sets, and report parameters appear in the data explorer. All elements in the report design appear in the Outline view.

ROM properties

ROM elements can have properties and every property has a type. Property types are similar to variable types in programming or data types in database terminology. Like variables and data types, ROM property types can be simple or complex. Simple types include string, number, dimension, color, and so forth. Complex types include structure and list. A complex type contains more than one component. For example, a text type contains both the text and a resource key used for internationalizing the text.

The components of a ROM property are:

- Property values

Most elements have simple properties that are defined by a name-value pair. There are several property types, described later in this section.

- User-defined property definitions

The userProperties array provides a way for users to define custom properties. Each item in the array is a UserProperty object.

- Executable expressions

The methods array is an associative array of method names. The method name is the key into the array. The return value is a string that contains the method text.

The property types defined in ROM include:

- property

This property type is the simplest and most common property type. A property definition of this type has the following syntax:

```
<property name="propName">value</property>
```

- property-list

This property type defines a set of properties, such as custom colors. A property definition of the property-list type has the following syntax:

```
<property-list name="propName">
  [ <structure> ... </structure> ] *
</property-list>
```

- **xml-property**

This property type defines custom XML. A property definition of the xml-property type has the following syntax:

```
<xml-property name="propName">value</xml-property>
```

- **expression**

The value for this property type is an expression. A property definition of the expression type has the following syntax:

```
<expression name="propName">value</expression>
```

- **structure**

This property is a collection of two or more properties. A property definition of the structure type has the following syntax:

```
<structure name="propName">
  <property name="member1">value1</property>
  <property name="member2">value2</property>
</structure>
```

ROM slots

A ROM slot is a collection of identically typed elements. For example, a report element has a slot of style elements that comprise all the styles available to the report.

ROM methods

A ROM element can have one or more methods, called event handlers. BIRT fires many different events during the course of executing a report. When BIRT fires an event, the appropriate event handler is executed to handle the event. By default, event handlers are empty methods that do nothing. By supplying code for an event handler, a report developer can customize and extend the functionality of BIRT. Supplying code for an event handler is called scripting. An event handler can be scripted in either JavaScript or Java.

Report items can have four events: onPrepare, onCreate, onPageBreak, and onRender. Each of these events fires during different phases of report creation. The onPrepare event fires in the preparation phase. The onCreate event fires during the generation phase. The onRender and onPageBreak events fire during the presentation phase.

ROM styles

The ROM style system is based on cascading style sheets (CSS), where a style set in a container cascades to its contents. The Report element contains all other elements, so the style property of the Report element defines the default style for the entire report. An element within the report can override the default style. A report developer can either choose a style from a defined

set of styles or create a new style. Typical style attributes include color, text size, alignment, background image, and so forth. For more information about the styles, see the ROM reference in the BIRT online help.

About the ROM schema

The ROM specification is encapsulated in a schema written in the language of XML Schema. XML Schema provides a standard way of defining the structure, content, and semantics of an XML file. XML Schema is similar to Document Type Definition (DTD). The ROM schema, therefore, contains the formal expression of the content, structure, and semantics of the ROM report design. The ROM schema is located at:

```
http://www.eclipse.org/birt/2005/design
```

A statement similar to the following one appears at the top of every report design file:

```
<report xmlns="http://www.eclipse.org/birt/2005/design"  
version="3.2.15" id="1">
```

This statement identifies the schema upon which BIRT bases the report design. If the design contains elements extraneous to or in violation of the rules set forth in the schema, it is not a valid design.

Opening a report design file with a schema-aware tool such as XMLSpy provides a means of verifying the report design against the schema. Using a schema-aware tool also can help a developer of a custom report designer to verify the output of the custom report designer.

The ROM schema defines syntax that allows extensions to BIRT without making changes to the actual schema. For example, an extended item uses the following tag:

```
<extended-item name="extension">
```

The ROM schema defines properties using the following syntax:

```
<property name="propertyName">value</property>
```

The ROM schema describes a syntax for representing properties. The ROM schema does not define any actual properties. ROM element properties are defined in another file, rom.def.

About the rom.def file

The rom.def file contains metadata defining the specific ROM elements, their properties, their slots, and their methods. You can find rom.def in:

```
$INSTALL_DIR\clipse\plugins  
\org.eclipse.birt.report.model_<version>.jar
```

The rom.def file is an internal file that the design engine uses to present a property sheet for a ROM element. The property sheet for an element contains the element's properties and their types, the element's methods, and valid choice selections for each of the element's properties.

The rom.def file specifies the following kinds of metadata:

- Choice

A choice definition specifies all the allowable values that an attribute can have. Most choice definitions relate to style attributes. The following example from rom.def defines all the allowable font families available to a fontFamily style specification.

```
<ChoiceType name="fontFamily">
    <Choice displayNameID="Choices.fontFamily.serif"
        name="serif" />
    <Choice displayNameID="Choices.fontFamily.sans-serif"
        name="sans-serif" />
    <Choice displayNameID="Choices.fontFamily.cursive"
        name="cursive" />
    <Choice displayNameID="Choices.fontFamily.fantasy"
        name="fantasy" />
    <Choice displayNameID="Choices.fontFamily.monospace"
        name="monospace" />
</ChoiceType>
```

- Class

A class definition defines a Java class that a report designer application can access using the BIRT model API. There are class descriptions for data types, such as String, Date, and Array. There are also class descriptions for the functional classes such as Total, Finance, and DateTimeSpan. Finally, there are class definitions for the report object definitions, such as Report, DataSet, DataSource, ReportDefn, and ColumnDefn. A class definition consists of definitions of the class attributes, methods, and localization identifiers. The following example from rom.def defines the Report class.

```
<Class displayNameID="Class.Report" name="Report"
    toolTipID="Class.Report.toolTip">
    <Member dataType="ReportDefn"
        displayNameID="Class.Report.design" name="design"
        toolTipID="Class.Report.design.toolTip" />
    <Member dataType="Object[]"
        displayNameID="Class.Report.params" name="params"
        toolTipID="Class.Report.params.toolTip" />
    <Member dataType="Object[]"
        displayNameID="Class.Report.config" name="config"
        toolTipID="Class.Report.config.toolTip" />
</Class>
```

The preceding class definition does not have methods. The following example illustrates a class method definition.

```
<Method displayNameID="Class.Total.sum" isStatic="true"
    name="sum" returnType="number"
    toolTipID="Class.Total.sum.toolTip">
    <Argument name="expr" tagID="Class.Total.sum.expr"
        type="number" />
    <Argument name="filter" tagID="Class.Total.sum.filter"
        type="String" />
    <Argument name="group" tagID="Class.Total.sum.group"
        type="String" />
</Method>
```

- Element

An element definition consists of the element's name, display name, methods, and properties, as well as the element from which it inherits. The rom.def file contains an element definition for every ROM element. The following example from the rom.def file illustrates an element definition.

```
<Element canExtend="true"
    displayNameID="Element.OdaDataSource"
    extends="DataSource"
    isAbstract="false" isNameRequired="true"
    javaClass="org.eclipse.birt.report.model.elements
        .OdaDataSource" name="OdaDataSource" since="1.0"
    xmlName="oda-data-source">
    <Property
        displayNameID="Element.OdaDataSource.extensionID"
        isIntrinsic="true" name="extensionID" since="1.0"
        type="string" />
    <Property detailType="ExtendedProperty"
        displayNameID=
            "Element.OdaDataSource.privateDriverProperties"
        isList="true" name="privateDriverProperties"
        since="1.0"
        type="structure" />
    <PropertyVisibility name="extensionID"
        visibility="hide" />
    <PropertyVisibility name="privateDriverProperties"
        visibility="hide" />
</Element>
```

The preceding element definition does not contain any methods. The following example illustrates an element method definition.

```
<Method context="factory"
    displayNameID="Element.ScriptDataSource.open"
    name="open" since="1.0"
    toolTipID="Element.ScriptDataSource.open.toolTip">
    <Argument name="reportContext"
        tagID="Element.ScriptDataSet.open.reportContext"
```

```

        type="org.eclipse.birt.report.engine.api.script
        .IRReportContext" />
<Argument name="object"
    tagID="Element.ScriptDataSet.open.object"
    type="Object" />
</Method>

```

- Structure

A structure is a complex data type that usually consists of two or more members. A few structures that are candidates for future expansion have only a single member. The following example from the rom.def file illustrates the definition of a structure.

```

<Structure displayNameID="Structure.DateTimeFormatValue"
    name="DateTimeFormatValue" since="1.0">
    <Member detailType="dateTimeFormat"
        displayNameID="Structure.DateTimeFormatValue.category"
        isIntrinsic="true" name="category" since="1.0"
        type="choice" />
    <Member
        displayNameID="Structure.DateTimeFormatValue.pattern"
        isIntrinsic="true" name="pattern" since="1.0"
        type="string" />
</Structure>

```

- Style

A style definition contains the least information of any type of metadata described in rom.def. A style definition defines the name of the style, its display name, and a reference value. The following example illustrates a style definition.

```

<Style displayNameID="Style.Report" name="report"
    reference="Overall default" />

```

- Validator

A validator definition specifies a Java class with which to do validation. Two of the validator classes are for validating values and all the rest are semantic validators. The following example from rom.def shows how to specify a validator.

```

<SemanticValidator
    class="org.eclipse.birt.report.model.api.validators
        .DataSetResultSetValidator"
    modules="design, library"
    name="DataSetResultSetValidator" />

```

About the primary ROM elements

The primary ROM elements consist of abstract elements from which other elements derive and concrete elements that provide the overall report

definition. The following elements are the primary components that form the basis for understanding ROM:

- **DesignElement**

DesignElement is an internal, abstract element used to implement basic features of ROM elements. DesignElement represents any component of a report design that has properties.

- **Listing**

Listing is the abstract base element for lists and tables. Both elements support a data set, filtering, sorting, and methods.

- **MasterPage**

MasterPage is an abstract base element that defines the basic properties of a page.

- **ReportDesign**

ReportDesign contains information about a report design, defining properties that describe the design as a whole. Report design properties do not inherit because a design cannot extend another design.

- **ReportElement**

ReportElement is an abstract report element that represents any item that can be named and customized. Most components in ROM derive from ReportElement, such as data sets, styles, master pages, and report items.

- **ReportItem**

The ReportItem element is the base element for the visual elements. A report item includes a style. The style provides visual characteristics for any element that prints in a report, such as a section or report item.

About the report item elements

There are many types of visual report components. Every type of visual report component has a corresponding ROM element that derives from the ReportItem element. Visual report components are called report items.

About the report items

There are top-level and lower-level report items. The top-level items are items that can contain other items. Examples of top-level items include:

- **Grid**

A grid contains a set of report items arranged into a grid with a fixed set of columns and a variable number of rows. Each cell in the grid can contain a single lower-level item or a container of items.

- List
A list contains a set of arbitrary content based upon data retrieved from a data set. A list is appropriate when some report items require a sophisticated layout and then repeat that layout for each row in a query.
- Table
A table contains a tabular layout of data retrieved from a data set.
Lower-level layout items have properties that describe their behavior and appearance in various ways. These items are not structural and do not contain other items. For example, the Image element is a lower-level element.

Understanding the report item element properties

The elements that derive from ReportItem are called the report item elements. Every report item has an entry in the palette, the visual BIRT Report Designer component that the report developer uses to build a report layout.

Each visual component has its own set of properties in addition to the properties it inherits from ReportItem. The types of inherited report item properties include:

- Method, which defines executable code.
- Property, which, includes such values as names and dimensions.
- StyleProperty, which defines style-related characteristics, such as color and font size.
- Slot, which contains Type elements that define its contents, as shown in the following element definition.

```
<Slot name="reportItems"
      displayNameID="Element.FreeForm.slot.reportItems"
      multipleCardinality="true">
    <Type name="Label" />
    <Type name="Data" />
    <Type name="Text" />
</Slot>
```

About the data elements

There are several elements in the ROM specification that apply to data rather than visual report items. These data elements describe data sources, data sets, and rows of data. The following elements are data elements:

- DataSource

The DataSource element represents a connection to an external data system, such as an RDBMS, text file, or XML file.

- **ScriptedDataSource**

The ScriptedDataSource element represents a connection to an external data system that is not an ODA data source. The developer must provide scripts for opening and closing a scripted data source. ScriptedDataSource inherits from DataSource.

- **DataSet**

The DataSet element represents a tabular result set retrieved from a data source. A DataSet element defines a query, filters, parameters, and result set columns.

- **JointDataSet**

The JointDataSet element represents a data set that results from a join of several data sets.

- **ScriptedDataSet**

The ScriptedDataSet element represents a data set that is associated with a scripted data source. The developer must provide scripts for opening, closing, and fetching a row from a scripted data source. ScriptedDataSet inherits from DataSet.

- **Row**

The Row element represents an integral set of column values that are a part of a result set.

III

Scripting in a Report Design



This page intentionally left blank

8

Using Scripting in a Report Design

BIRT provides a powerful scripting capability that allows a report developer to create custom code to control various aspects of report creation. This chapter provides an overview of scripting in BIRT. Subsequent chapters focus on implementing script event handlers in JavaScript and Java, writing event handlers for charts, and accessing data using scripted data sources.

Overview of BIRT scripting

When developing a BIRT report using the Eclipse Workbench, you can write custom event handlers in either JavaScript and Java. When developing a BIRT report using the BIRT RCP Report Designer, you can write only JavaScript event handlers.

Choosing between JavaScript and Java

Both JavaScript and Java have advantages and disadvantages when writing an event handler. For a developer who is familiar with only one of the two languages, the advantage of using the familiar language is obvious, but for others the decision depends on the report requirements.

The advantages of using JavaScript to write an event handler include

- Ease of adding a simple script for a particular event handler

Adding a JavaScript event handler to a report is less complicated than adding a Java event handler. When writing a JavaScript event handler, there is no need to create a Java environment in Eclipse or to learn the Eclipse Java development process. You are not required to specify a

package, implement an interface, or know the parameters of the event handler you write.

To add a JavaScript event handler, you type the code for the event handler on the Script tab after selecting the name of the event handler from a drop-down list.

- Simpler language constructs, looser typing, and less strict language rules
JavaScript is less demanding to code than Java due to these more relaxed requirements.

The advantages of using Java to write an event handler include

- Availability of the Eclipse Java development environment

The Eclipse Java development environment is very powerful and includes such features as autocompletion, context sensitive help, keyboard shortcuts, parameter hints, and more.

- Ease of finding and viewing event handlers

All the Java event handlers for a report exist in readily viewable Java files. By contrast, the JavaScript event handlers are embedded in the design and you can view only one handler at a time.

- Access to an integrated debugger

The integrated debugger only supports Java event handlers, not JavaScript event handlers.

Using both JavaScript and Java to write event handlers

You are not limited to writing all event handlers in one language. You can write some in JavaScript and others in Java. If you have both a JavaScript and a Java event handler for the same event, BIRT uses the JavaScript handler.

Events overview

When writing event handlers, understanding the event order is imperative. The order in which events fire depends on several factors. These factors include what BIRT processing phase is executing, the engine task executing the process, and what event type is processing.

Engine task processes

The scripting chapters make continuous reference to engine task processes. This section provides an overview of what these processes are and how they affect scripting.

The Report Engine that executes reports can be used in different ways depending on developer requirements. The report engine is task-oriented and provides three tasks related to the execution and rendering of reports; these are RunAndRenderTask, RunTask, and RenderTask.

The RunAndRenderTask uses one process to open the report design and produce a specific output, such as PDF. The RunTask opens a report design and executes the report producing a report document file with a .rptdocument extension. This report document is an intermediate binary file that can be used by a RenderTask to produce a report output type, such as HTML, PDF, XLS, Word, PPT, or PS. The RenderTask opens a report document (.rptdocument) and renders the appropriate output format. It is important to note that this task can be executed anytime after a RunTask. It can even occur on a separate system. Using separate run and render tasks therefore requires two processes to run and render a report to a particular output format.

When RunAndRenderTask processes a report, the event firing order is different than the when RunTask and Render Task are used as two separate processes. The differences in processing are discussed throughout these chapters.

The following sections describe how the report engine processes reports in the BIRT Web Viewer and Report Designer environments.

BIRT Web Viewer

The example BIRT Web Viewer application is a J2EE application that encapsulates the report engine to produce reports. This viewer contains three Servlet mappings used to generate reports. These are the frameset, run, and preview mappings.

When using the frameset servlet mapping in the example Web Viewer two processes, RunTask and RenderTask, generate and render to the output format, to create a report document. When selecting the Export report icon in the Web Viewer toolbar, a RenderTask executes on the current report document. When using the run or preview Servlet mappings, one process, RunAndRenderTask, executes without creating a report document.

All three Servlet mappings render an existing report document if the __document URL parameter is used. In this case, the report engine uses only a RenderTask.

For more information about the BIRT Web Viewer, see Chapter 5, “Using Eclipse BIRT Web Viewer.”

BIRT Report Designer

The BIRT Report Designer also uses the report engine to show a preview of a report. Selecting Preview in the Editor launches a RunAndRenderTask to produce the report. Selecting any other previewing option in the BIRT Report Designer toolbar launches a RunAndRenderTask to produce the output.

Selecting Run report in BIRT Web Viewer produces the report using a RunTask then RenderTask.

BIRT processing phases

The “Understanding the BIRT Architecture” chapter describes the BIRT services for generating and presenting report data. These services create a report during the following processing phases:

- Preparation
RunTask or RunAndRender Task prepare the report items for execution
- Generation
RunTask or RunAndRender Task create an instance of each report item, connect to the data source, execute the data sets, and process the data to produce the report
- Presentation
RenderTask or RunAndRender Task select the correct emitters to produce the output specified for the report

The RunTask handles the preparation and generation phases and the Render Task handles the presentation phase. RunAndRenderTask combines all the phases of report processing. The types of events and the order in which these events fire depends on the processing phase currently executing in an engine task.

BIRT event types

BIRT supports the following types of events:

- Parameter
- Report design
- Data source and data set
- Report item

Each event type has a series of events that fire during report processing.

Parameter events

BIRT currently supports only one parameter level event, which is the validate event. This event is only available in JavaScript and it is the first event triggered when a report containing parameters executes. This event expects the event handler to return a true or false value. Returning false throws a parameter not set exception. Returning true processes the report normally.

Writing an event handler for this event is useful when changes to the parameter are necessary after a user enters a parameter or extra parameter validation is required. The following example shows a validate script for a string parameter:

```
params["MyParameter"].value = "Something Else";
true;
```

Report design events

Report design events fire for all reports. Table 8-1 describes the events that can be overridden.

Table 8-1 Report design events

Event	Description
initialize()	Fires every time a task accesses the report design (.rptdesign) or the report document (.rptdocument). This event occurs once when using RunAndRenderTask. When the processing phases run separately, RunTask triggers once for the generation phase and RenderTask triggers once for every render operation. If you use the frameset mapping from the example Web Viewer, the generation phase fires the initialize event handler once. Displaying the first page of a report in the presentation phase and using the page navigation control to access a new page also trigger the this event handler.
beforeFactory()	Fires just prior to the generation phase after the elements in the report have been prepared in the preparation phase. This event handler only fires once and is useful when modifications to the report design are required before execution.
beforeRender()	Fires just prior to the presentation phase and is called for every render operation. In the RunAndRender task, this event occurs once just after the beforeFactory event. This event fires for every render operation when using frameset mapping because the task uses two processes.
afterFactory()	Fires at the conclusion of the generation phase and fires only once. If using a RunTask or RunAndRenderTask, this event is the last one fired. If using a RenderTask, this event does not execute.

(continues)

Table 8-1 Report design events (*continued*)

Event	Description
afterRender()	Fires at the conclusion of the presentation phase for a specific render operation. When creating a report, if you are using a single RunAndRenderTask process, this event is called only once. If using separate RunTask and RenderTask processes, this event fires for every render operation.

Data source and data set events

There are several kinds of data sources and data sets. A data source can be a flat file, a JDBC data source, a scripted data source, a web services data source, or an XML data source. All data sources have a common set of event handlers. A scripted data source has two additional event handlers that the others do not.

A scripted data set is a data set that accesses a scripted data source and a non-scripted data set is one that accesses a standard data source. A scripted data set contains all the event handlers of an unscripted data set plus three others.

The scripted data source and data set are used for writing custom code to retrieve data values. The extra events are used to retrieve this data. Data source and data set events fire prior to being used on a data bound item. If the data set is never used in the report, the data source and data set are never called.

It is important to realize that the data set events can be called multiple times to support multipass aggregation and data set sharing. It is not advisable to write event handlers that rely on the data set event firing order.

Non-scripted data source events

A non-scripted data source contains afterClose, afterOpen, beforeClose, and beforeOpen events. You use the non-scripted data source events to perform operations that are not directly related to managing the data source. There is no requirement to implement the non-scripted data source event handler methods.

Scripted data source events

A scripted data source contains the same four events that a non-scripted data source contains plus two others, open and close. You use the event handlers for the open and close events to perform the actions of opening and closing the data source.

Table 8-2 describes the data source events.

Table 8-2 Data source events

Event	Description
beforeOpen()	Fires prior to opening a connection to a data source and is most often used to modify public properties of the data source, including database URL, username, password, driver class and JNDI URL. This event fires only once before the connection to the data source opens event when the connection is used for multiple data sets.
afterOpen()	Fires after the connection to the data source opens.
beforeClose()	Fires at the conclusion of the generation phase, just prior to closing the data source connection.
afterClose()	Fires after the data source connection closes.
open()	Fires only for a scripted data source, providing a location for the developer to setup a connection to an external source.
close()	Fires for a scripted data source, providing a location for the developer to close a connection to an external source.

Non-scripted data set events

A non-scripted data set includes afterClose, afterOpen, beforeClose, beforeOpen, and onFetch events. As with the non-scripted data source events, you are not required to provide event handlers for a non-scripted data set.

Scripted data set events

A scripted data set contains the same events that a non-scripted data set contains plus three others, open, close, and fetch. You use the event handlers for the open and close events to perform the actions of opening and closing the data set. You use the fetch method to fetch a row from the data source. If using a scripted data set you must write an event handler for the fetch event.

Data source and data set events for data sets fire in the Generation phase prior to the onCreate event of the report item that uses them. Data Set events can fire several times. The event order is covered in more detail later in this chapter. For more information about scripted data sources, see Chapter 12, “Accessing Data Programmatically.” Table 8-3 describes the data set events.

ReportItem Events

ReportItem events are triggered for report items that are placed in the report. Most items support writing event handlers for the events listed in Table 8-4.

Table 8-3 Data set events

Event	Description
beforeOpen()	Fires prior to opening a data set and is used most often to modify public properties of the data set. For example, when using a JDBC data set, the query text can be altered using this event. This event fires for every report item bound to the data set. If two tables use the same data set, the data set is called twice, resulting in this event firing twice. BIRT 2.2.1 allows report items to be bound to other report items, in which case the data set is called only once, resulting in the event triggering only once. Future versions of BIRT will support more data set caching options, which will affect how often data source and data set events trigger.
afterOpen()	Fires after the data set opens.
onFetch()	Fires as the data set retrieves each row of data. This event triggers for all rows of data before the onCreate event for the particular report item that uses the data set. For more information, see “About data binding,” later in this chapter.
beforeClose()	Fires before closing the data set after creating the report item that uses the data set.
afterClose()	Fires after the data set closes.
open()	Fires for a scripted data set, providing a location for the developer to setup a data set.
fetch()	Fires for a scripted data set, providing a location for the developer to populate rows from the scripted data set.
close()	Fires for a scripted data set, providing a location for the developer to close the set.

Table 8-4 Report item events

Event	Description
onPrepare()	Fires at the beginning of the preparation phase before data binding or expression evaluation occurs. This event is useful for changing the design of the item prior to generating the item instance.
onCreate()	Fires at the time the generation phase creates the element. This event is useful when a particular instance of a report item needs alteration.

Table 8-4 Report item events (*continued*)

Event	Description
onRender()	Fires in the presentation phase. This event is useful for operations that depend on the type or format of the output document.
onPageBreak()	Fires for all report items currently on the page when the page break occurs. Not all report items support the onPageBreak event.

Event order sequence

Table 8-5 summarizes which engine task is responsible for a particular phase. The following sections describe the order of event firing for each phase in more detail.

Table 8-5 Engine task by phase

Report Engine Task	Preparation Phase	Generation Phase	Presentation Phase
RunTask	Yes	Yes	No
RenderTask	No	No	Yes
RunAndRender Task	Yes	Yes	Yes

Preparation phase operation

The preparation phase includes parameter validation as well as initialization and report element preparation for every element in the report. The preparation phase is identical for all reports. Table 8-6 lists the event types for RunTask and a RunAndRenderTask in the preparation phase in the order in which the events execute.

The RenderTask does not have a preparation phase. In a RunAndRenderTask, the preparation phase triggers a beforeRender event.

Table 8-6 Preparation phase events

Event type and event	RunTask	RunAnd RenderTask
Parameter validate()	Yes	Yes
ReportDesign	Yes	Yes
Initialize()		

(continues)

Table 8-6 Preparation phase events (*continued*)

Event type and event	RunTask	RunAnd RenderTask
ReportItem onPrepare() (iterative)	Yes	Yes
ReportDesign beforeFactory()	Yes	Yes
ReportDesign beforeRender()	No	Yes

The first event is the validate event, which triggers for each parameter. The ReportDesign event, initialize, follows. The initialize events triggers once when using only one engine task. Using a separate engine task triggers the initialize event at least twice, once for the Preparation phase, and a second time for the Render phase. Additional render tasks or phases also trigger the initialize method.

After the initialize event, every element in the report has its onPrepare event triggered. This process starts with the master page content and proceeds from left to right and top to bottom in the report body. All nested elements process before proceeding to the next element.

After the onPrepare event, the beforeFactory event triggers. This event signals that report creation is about to occur and provides a location for altering a running report.

The beforeRender event triggers only when using a RunAndRenderTask to run and render report content. When using two tasks, this event does not trigger until a render operation occurs.

Generation phase operation

The generation phase includes connecting to data sources, executing data sets and data cubes, data binding evaluation, and the creation of all the report items in the report. It is important to realize that the data source and data set events fire before the creation of data-bound items, but this processing may not occur before the creation of other report items. For example, if a table is bound to a data set and the report uses a master page with only a label in the footer, the master page content onCreate events fires before the data source and data set events for the data set bound to the table.

This phase begins by triggering the onCreate event for every report. This process starts with the master page content and proceeds from left to right and top to bottom in the report. All nested elements process before proceeding to the next element.

If a RunAndRenderTask process executes the generation phase, each element is created and immediately rendered, which fires the onRender event before

proceeding to the next element. If a RunTask process executes the generation phase, the onRender events do not fire.

BIRT processes the report body after processing content on the master page. The report body contains all the report items to be created and rendered. A report item that is not contained in another report item is called a top-level report item. BIRT processes the top-level items, going from left to right and proceeding a row at a time toward the bottom right. Every report has at least one top-level report item usually a grid, list, or table.

For each top-level item, BIRT processes all the second-level items before proceeding to the next top-level item. A second-level report item is a report item that is contained within a top-level item. For example, a table contained in a grid is a second-level report item.

There can be any number of levels of report items. To see the level of a particular report item, examine the structure of the report design in Outline, in the BIRT Report Designer, as shown in Figure 8-1.

BIRT processes all items at all levels in an iterative fashion, following the same process at each level as it does for the top-level items.

Table 8-7 lists the events triggered in the generation phase for each major report component in the order in which these events execute. Table 8-7 lists the data source, data set, and onPageBreak events as optional.

Table 8-7 Generation phase events

Report component	RunTask	RunAndRenderTask
MasterPage Content	Data source and data set events (optional) onCreate onPageBreak (optional)	Data source and data set events (optional) onCreate onRender onPageBreak (optional)
Body (iterative)	Data source and data set events (optional) onCreate onPageBreak (optional)	Data source and data set events (optional) onCreate onRender onPageBreak (optional)

Data source and data set events do not fire if a report item is not data bound or if the report item is bound to another report item that executed previously. The onPageBreak event only fires when an actual page break occurs.

The next three sections describe the data source, data set, data binding and page break events in more detail.

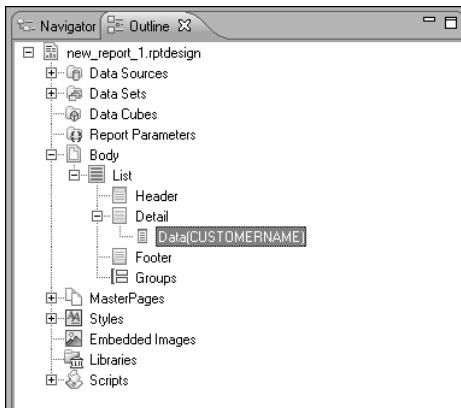


Figure 8-1 Outline showing the level of a report item

About data source and data set events

Events for data source and data set elements trigger just prior to creating the report item bound to the data set. This sequence occurs for every report item bound to a data set with the exception of the data source beforeOpen and afterOpen events.

These events do not trigger if a data source has already been used. When a report item is bound to another report item, none of these events trigger for the specific report item, allowing two data-bound items to share one data set without the need to re-execute the data set.

The data source beforeClose and afterClose trigger at the end of the generation phase just prior to when the afterFactory event triggers. In the generation phase, there is no difference in data source or data set processing in RunTask and the RunAndRenderTask.

In future versions of BIRT, data set caching will support altering when and if these events occur for a particular data-bound report item. These events will fire at least once before the data set is used in a data bound element. Table 8-8 lists the data source and data set types and events in the order in which these events execute.

Table 8-8 Data source and data set events

Event type and event	RunTask	RunAnd RenderTask
Data source beforeOpen()	Yes	Yes
Data source afterOpen()	Yes	Yes
Data set beforeOpen()	Yes	Yes

Table 8-8 Data source and data set events (*continued*)

Event type and event	RunTask	RunAnd RenderTask
DataSet afterOpen()	Yes	Yes
DataSet onFetch() for all rows of data	Yes	Yes
Process the data-bound Report Item	Yes	Yes
DataSet beforeClose()	Yes	Yes
DataSet afterClose()	Yes	Yes

About data binding

Data binding in BIRT makes a logical separation between BIRT data sets and data-bound elements, such as tables and list. You can see the current bindings for a table or list by selecting the Binding in the Property Editor with the table or list selected. Bound columns do not have to be limited to a bound data set, allowing a designer to create bound columns that use external objects to calculate a specific column value. The bound data set is still used to determine the number of rows that are processed for a given table or list.

With a tables or list, evaluation of the data bindings for the detail rows occurs before the onCreate event on the current row. This processing allows the onCreate event handler to retrieve the appropriate values for the current rows.

The bindings are evaluated for each detail row in a bound data set. Table footers containing data elements that use a binding are evaluated before the last onCreate event for the final detail row.

Trying to alter a bound column using the onCreate script is discouraged. For example, concatenating a column value for a row with all previous rows for the given column using the onCreate script with a JavaScript variable is possible. However, if this value is placed in the expression of a bound column, it will not produce the required results, resulting in the last value being excluded in the bound column. If this is a requirement, it is better to add a computed column, an aggregation report item or use the dynamic text <VALUE-OF> tag within a Text element to display your JavaScript value.

It is important to stress that a user should not assume any particular evaluation order of binding expressions relative to the various table item events, other than the obvious rule that a binding evaluates before the

`onCreate` event of an item that uses it. If any manipulation of the data needs to be done, it is best to do it in a data set script or the binding expression.

Data binding evaluation for a group also evaluates the bound column many times before creating the table or list item. It is necessary to perform multiple passes over the data to support grouping. It is important to understand that any script in a group-bound column expression is called every time the data binding evaluates.

All bindings of a table are always calculated, whether or not these bindings are used by the table. For performance reasons, check the binding list and remove any unused items from the report design.

With BIRT 2.2.1, report items can share data bindings. For example, two tables can share the same set of bindings. This feature prevents the second table from executing the query again. To use this feature in the BIRT Report Designer, select the second table, choose Binding in Property Editor, select Report Item, and choose the appropriate report item from the list. The list only shows named data-bound report items. Using this feature effects when data binding evaluation occurs. Instead of calling the binding evaluation just prior to the `onCreate` event for a given row, the entire data set evaluates prior to creating the table or list that uses the data set.

The above scenarios will affect any BIRT expression that use the Available Column Bindings category.

About the page break event

You can set page breaks in the BIRT Report Designer on many report elements. You can set a page break interval on a table, which instructs the table to process a certain number of rows and then automatically apply a page break. You can set a page break on a group section to apply a page break before and after or after the group processes. You can also apply a page break before many of the report elements. For example, applying a page break before a row in a table instructs the engine to apply a page break before every row of data that a table processes.

Many report elements support an `onPageBreak` event handler. Page break implementation depends on whether the presentation and generation occur within one engine task and what emitter is used. If using two tasks, one for generation and one for presentation, the page break events fire during the generation phase, while creating the report document. In this case, the page break events do not fire during the presentation phase.

If one engine task generates and renders the report, the emitter configuration determines if the page break event fires. For example, the Word and PDF emitters support pagination and fire `onPageBreak` events. The Excel emitter does not support pagination.

In either case, if page break events are supported, they only fire for the report elements contained on the page prior to the page break. For example, if a table footer has not been created when the page break event fires, the report

elements contained in the footer do not have their onPageBreak event handlers evaluated. The onPageBreak events fire just prior to the onCreate events for elements located on the master page for the next page.

About chart event order

Chart events are handled by the chart engine and execute within the presentation phase. These events are covered in a subsequent chapter.

About table and list event order

The BIRT engine fires the onCreate event for report items iteratively. When processing report items that iterate over data rows, such as a table or list, the event order changes to add additional events for each row of data. Data rows process using Row containers.

Row execution sequence

There are three kinds of rows:

- Header
- Detail
- Footer

Tables, lists, and groups have rows. BIRT processes all rows identically.

Figure 8-2 illustrates the execution sequence for a row. A list or table can contain multiple elements, such as detail rows or table header rows.

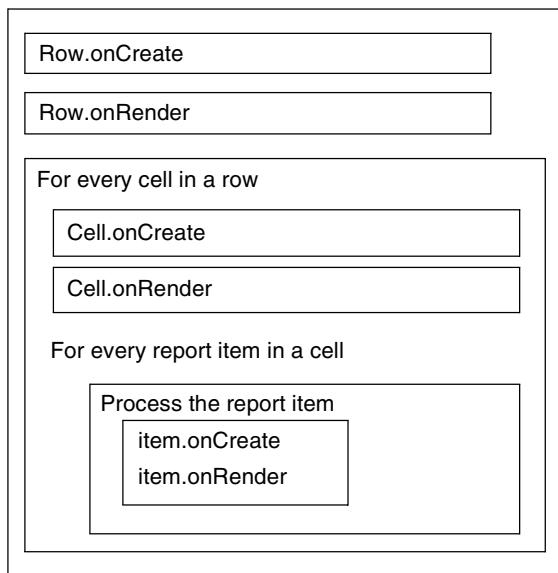


Figure 8-2 Row execution sequence

Table and list method execution sequence

A list is the same as a table, except that it only has a single cell in every row. BIRT processes tables and lists identically, except that for a list, BIRT does not iterate through multiple cells. BIRT processes tables in three steps, the setup, detail, and wrap-up processing steps, as shown in Figure 8-3.

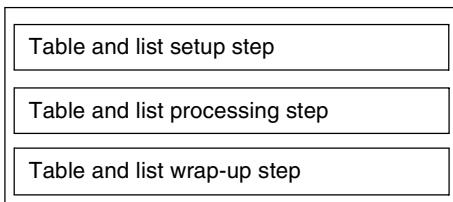


Figure 8-3 Table and list execution sequence

The following sections describe the table and list execution sequence steps.

Table and list setup step

The pre-table processing step is the same for all tables, both grouped and ungrouped.

Figure 8-4 illustrates the execution sequence for the pre-table processing step. This illustration shows the event order when using a RunAndRenderTask in the generation phase.

Disregard onRender events when using the RunTask. Disregard onCreate and data source and data set events when using a RenderTask. The data source and data set events are optional.

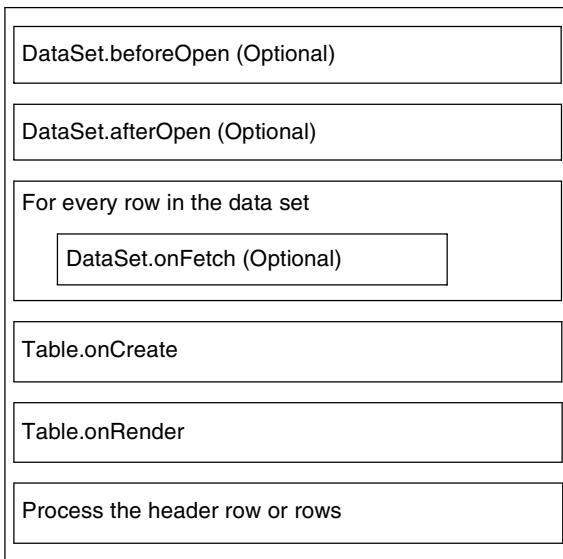


Figure 8-4 Table and list setup execution sequence

Table and list processing step

The sequence for the table and list processing step depends on whether the table or list is grouped. A table or list with no grouping has a different sequence than one with grouping.

Figure 8-5 illustrates the execution sequence for a table or list without grouping.

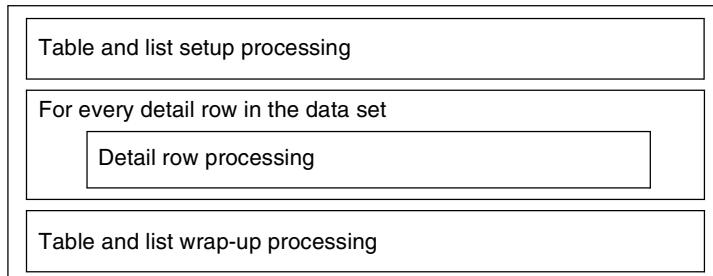


Figure 8-5 Ungrouped table or list detail execution sequence

For a table with grouping, BIRT creates one ListingGroup item per group.

A ListingGroup is very similar to a table because it has one or more header rows and one or more footer rows. BIRT processes grouped rows in the same way that it processes a table row.

In addition to the standard onPrepare, onCreate, and onRender events for these rows, the ListingGroup fires the onPageBreak and onPrepare events for the group. To access the script location for these event handlers, locate the group in the outline view and select the script tab. You can override the onPrepare event to modify grouping behavior, such as changing the sort order.

Figure 8-6 illustrates the method execution sequence for a table that has groups.

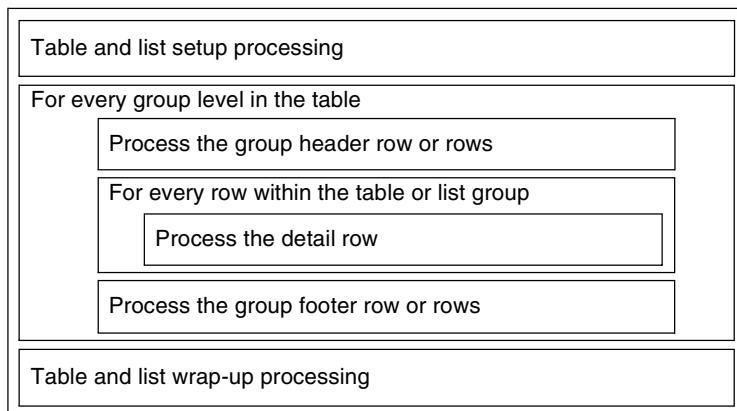


Figure 8-6 Grouped table execution sequence

If you need to verify the execution sequence of event handlers for a specific report, you can add logging code to your event handlers. For information about adding logging code, see “Determining method execution sequence” in Chapter 9, “Using JavaScript to Write an Event Handler.”

Table and list wrap-up step

The post-table processing step is the same for all tables, both grouped and ungrouped. Figure 8-7 illustrates the execution sequence for the post-table processing step.

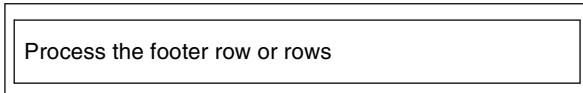


Figure 8-7 Table and list wrap-up execution sequence

Completion of the generation phase

On completion of the generation phase, all data sources beforeClose and afterClose events trigger followed by the afterFactory event. If using the RunAndRenderTask, the afterRender event fires before closing the data sources. Table 8-9 describes the events available at completion of the generation phase.

Table 8-9 Completing the generation phase

Event type and event	RunTask	RunAndRenderTask
afterRender	No	Yes
Data Source(s) beforeClose	Yes	Yes
Data Source(s) afterClose	Yes	Yes
afterFactory	Yes	No

Presentation phase operation

The presentation phase launches the appropriate emitter and produces report output based on the generated report. This phase triggers the onRender events for all items as they are created. If using the RenderTask to render an existing report document, the initialize event triggers first then each rendered report item onRender event triggers.

If the RenderTask renders pages, only the items that actually are rendered have onRender events triggered. For example, the frameset mapping in the example BIRT Web Viewer uses a RunTask to create a report document. The Web Viewer uses a RenderTask to render the first page.

This action fires the initialize event first and the onRender event for each item that appears on page. Selecting a new page and using the pagination controls

results in a new RenderTask that calls the initialize event again and triggers the onRender event for each item on the new page.

Event order summary

Table 8-10 summarizes the report design and report item events triggered in each processing phase. The events are listed in the order they are fired for a particular task. Page break and data source and data set events are not shown for brevity.

Table 8-10 Event order summary

Event type and event	RunTask	Run And Render Task	Render Task	Notes
Parameter validate	Yes	Yes	n/a	Only available in JavaScript
Report Initialize	Yes	Yes	Yes	Called multiple times using the RenderTask
MasterPage Content onPrepare	Yes	Yes	n/a	
Body Iterate onPrepare	Yes	Yes	n/a	
Report beforeFactory	Yes	Yes	n/a	
Report beforeRender	n/a	Yes	Yes	
MasterPage Content onCreate	Yes	Yes	n/a	
Body Iterate onCreate	Yes	Yes	n/a	
Body Iterate onRender	n/a	Yes	Yes	RunAndRender triggers this event immediately after onCreate for a report item
Report afterRender	n/a	Yes	Yes	

(continues)

Table 8-10 Event order summary (*continued*)

Event type and event	RunTask	Run And Render Task	Render Task	Notes
Report afterFactory	Yes	Yes	n/a	

9

Using JavaScript to Write an Event Handler

BIRT scripting is based on the Mozilla Rhino implementation of JavaScript, also called ECMAScript. Rhino implements ECMAScript version 1.5 as described in the ECMA standard ECMA-262 version 3. The complete specification for Rhino is located at:

[http://www.ecma-international.org/publications/standards/
Ecma-262.htm](http://www.ecma-international.org/publications/standards/Ecma-262.htm)

Using BIRT Report Designer to enter a JavaScript event handler

You can use BIRT Report Designer to enter a JavaScript event handler and associate it with a specific event for a particular element.

How to use BIRT Report Designer to enter a JavaScript event handler

- 1 In Outline, select the report element, data source, or data set for which you want to write an event handler.
- 2 Choose the Script tab.
- 3 Choose an event handler from the drop-down list of methods.
- 4 Enter the event handler code in the script editor.

Figure 9-1 demonstrates entering a line of code in the `onPrepare()` method of a Table element.

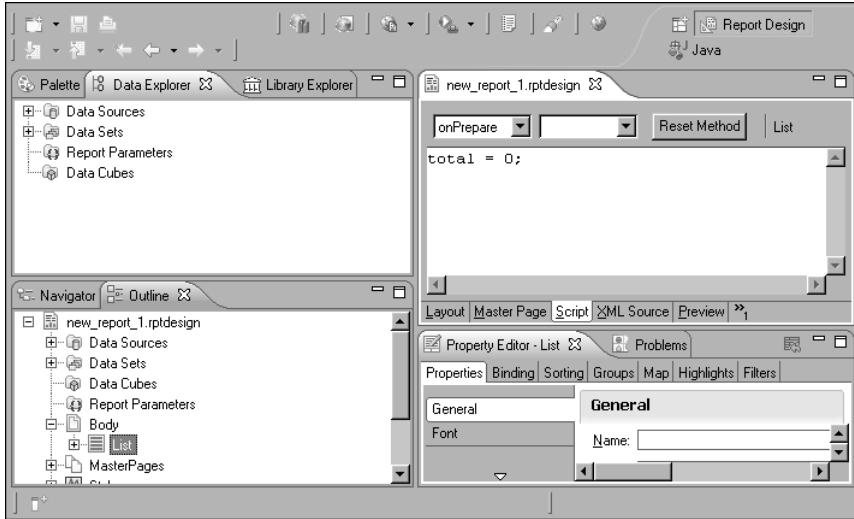


Figure 9-1 Code entry for the onPrepare() method

Creating and using a global variable

JavaScript has global variables and local variables. A local variable can only be accessed in the scope of the method in which it is created. You use the var identifier to create a local variable in JavaScript, as shown in the following line of code:

```
var localCounter = 0;
```

To create a global variable, you omit the var identifier, as shown in the following line of code:

```
globalCounter = 0;
```

When you create a global variable in JavaScript, that variable is visible to all other JavaScript code that executes in the same process. For example, you can use a global variable to count the detail rows in a table by first creating a global variable in the onCreate() method of the table, as shown in the following line of code:

```
rowCount = 0;
```

Since rowCount is global, the onCreate() method of the detail row can access and increment it, as shown in the following line of code:

```
rowCount++;
```

Global variables can also be stored using the reportContext.setGlobalVariable method. This allows the global variable to be passed to a Java event handler.

It is important to realize that if a report is being executed using two separate processes, such as RunTask and RenderTask, a global variable is only available to both processes if set using the

`reportContext.setPersistentGlobalVariable` method. This method writes the value of the variable to the report document before rendering. If you are setting a global variable in the initialize method, remember that this event triggers many times when using two processes.

Understanding execution phases and processes

There are three BIRT execution phases: preparation, generation, and presentation. There can be one or two execution processes. When a report runs in the BIRT Report Designer previewer, there is only one execution process, which executes a `RunAndRenderTask`.

There are two execution processes when the report runs in the interactive viewer or when the report deploys to an application server and uses the frameset servlet mapping. The first process, called the factory process, contains the preparation and generation phases. The second execution process, called the render process, contains only the presentation phase. The render process can occur at a much later time than the factory process and possibly on a different machine.

Because variables are only visible in the process in which they are created, it is important to know which event handlers run in which process. It is also important to be aware that code that works when running the report in the previewer may not work at run time if there is a render process dependency on a variable created in the factory process.

The event handlers that run in the factory process, in the order executed, include:

- 1 `Parameter.validate()`
- 2 `ReportDesign.initialize()`
- 3 `onPrepare()` for every report item
- 4 `ReportDesign.beforeFactory()`
- 5 `DataSource.beforeOpen()`
- 6 `DataSource.afterOpen()`
- 7 `DataSet.beforeOpen()`
- 8 `DataSet.afterOpen()`
- 9 `DataSet.onFetch()`
- 10 `onCreate()` for every report item
- 11 `DataSet.beforeClose()`
- 12 `DataSet.afterClose()`
- 13 `DataSource.beforeClose()`
- 14 `DataSource.afterClose()`

15 ReportDesign.afterFactory()

The event handlers that run in the render process, in the order executed, include:

- 1** ReportDesign.initialize()
- 2** ReportDesign.beforeRender()
- 3** onRender() for every report item
- 4** ReportDesign.beforeFactory()
- 5** ReportDesign.afterRender()

If using only one process, RunAndRenderTask, the render events occur right after the onCreate events for individual report items.

It is worth noting that ReportDesign.initialize() runs in both processes and that page break events occur at generation time when using two processes.

Using the reportContext object

Almost every event handler has access to an object called the reportContext object. The four exceptions are the open() and close() event handlers for ScriptedDataSource and ScriptedDataSet elements. Table 9-1 lists commonly used reportContext object methods.

Table 9-1 Methods of the reportContext class

Method	Task
deleteGlobalVariable()	Deletes a global variable created using setGlobalVariable()
deletePersistentGlobalVariable()	Deletes a persistent global variable created using setPersistentGlobalVariable()
getApplicationContext()	Returns the application context
getConfigVariableValue()	Returns the value of a config variable
getGlobalVariable()	Returns a global variable created using setGlobalVariable()
getHttpServletRequest()	Returns the HTTP servlet request object
getLocale()	Returns the current locale
getMessage()	Returns a localized message from the localization resource file
getOutputFormat()	Returns the format of the emitted report
getParameterValue()	Returns a parameter value

Table 9-1 Methods of the reportContext class (*continued*)

Method	Task
getPersistentGlobalVariable()	Returns a persistent global variable created using setPersistentGlobalVariable()
setGlobalVariable()	Creates a global variable accessible with getGlobalVariable()
setParameterValue()	Sets the value of a named parameter
setPersistentGlobalVariable()	Creates a persistent global variable accessible using getPersistentGlobalVariable()

Using `getOutputFormat`

To change styling for an element at render time depending on the output format, use the reportContext object to add the code to the onRender eventHandler, as shown in the following code example:

```
if (reportContext.getOutputFormat() == "pdf"){
    this.getStyle().backgroundColor = "red";
} else{
    this.getStyle().backgroundColor = "blue";
}
```

To create two different master pages in the design and swap pages based on the output format, set the beforeFactory event handler, as shown in the following code example:

```
rptDesignHandle =
    reportContext.getReportRunnable().designHandle
    .getDesignHandle();
tbl = rptDesignHandle.findElement("mytable");
var myoutputformat = reportContext.getOutputFormat();
if( myoutputformat == "html" ){
    tbl.setProperty("masterPage","MasterPageTwo");
} else{
    tbl.setProperty("masterPage","MasterPageOne");
}
```

This example uses the reportContext object to retrieve a handle to the report design. The design handle locates a table element named mytable and, depending on the output format, sets the masterPage property to the first or second master page. It is important to understand that master page creation occurs during report generation and can not change at run time. The example works only when recreating the entire report document or running and rendering the report using one task.

Using reportContext to retrieve the report design handle

You can use the reportContext object to retrieve the report design handle in the beforeFactory event handler. You can modify the currently running design before the design executes.

The following event handler examples are placed in the beforeFactory method, which executes only during the generation phase of report creation. The examples reference the Design Engine API, which is described in a later chapter.

The following example uses the design handle to add a filter to a table that is created based on report parameters:

```
importPackage(Packages.org.eclipse.birt.report.model.api.elemen  
ts);  
importPackage(Packages.org.eclipse.birt.report.model.api);  
  
tableHandle = reportContext.getReportRunnable().designHandle  
.getDesignHandle().findElement("mytable");  
if( params["FilterCol"].value.length > 0 ){  
    fc = StructureFactory.createFilterCond();  
    fc.setExpr("row['" + params["FilterCol"] + "']");
    fc.setOperator(params["FilterEq"]);
    fc.setValue1("\\" + unescape(params["FilterVal"]) + "\\");
    ph = tableHandle.getPropertyHandle(TableHandle.FILTER_PROP);
    ph.addItem(fc);
}
```

The next example is similar to the filter example. It adds a sort condition to a table, as shown in the following code:

```
importPackage(Packages.org.eclipse.birt.report.model.api);
importPackage(Packages.org.eclipse.birt.report.model.api.elemen  
ts);
tbl= reportContext.getReportRunnable().designHandle
    .getDesignHandle().findElement("mytable");
sc = StructureFactory.createSortKey();
sc.setKey("row[\"PRODUCTCODE\"]");
sc.setDirection("desc");
ph = tbl.getPropertyHandle(TableHandle.SORT_PROP);
ph.addItem(sc);
```

A report designer may decide to hide a portion of report content using the visibility property. The reason may be to perform an unrendered calculation or to customize the display of data based on report parameters.

When using the visibility property, it is important to realize that the report item is still processed, which may or may not be desirable. As an alternative to this approach, you can use the report design handle to drop elements from a running design, as shown in the following code:

```
reportContext.getReportRunnable().designHandle  
    .getDesignHandle().findElement("table1").drop();
```

You can alter additional properties of report items. The next example shows how to resize the width of a column within a table and change the bound data set. When changing the bound data set for a report item, it is important to realize that the new data set must contain the columns of previous data set.

```
table = reportContext.getReportRunnable().designHandle  
    .getDesignHandle().findElement("table1");  
ch = table.getColumns().get(0);  
ch.setProperty("width", "10%");  
table.setProperty("dataSet", "My Alternate Data Set");
```

You can change the styles property in the beforeFactory event. The following example uses one style for PDF generation and another style for all others.

```
tableHandle = reportContext.getReportRunnable().designHandle  
    .getDesignHandle().findElement("mytable");  
rowHandle = tableHandle.getDetail().get(0);  
if( reportContext.getOutputFormat() == "pdf" ){  
    rowHandle.setStyleName("style2");  
}else{  
    rowHandle.setStyleName("style1");  
}
```

Passing a variable between processes

Although a global JavaScript variable cannot pass between processes, there is a way to pass a variable from the factory process to the render process. The `setPersistentGlobalVariable()` method of the report context object creates a variable that you can access using the `getPersistentGlobalVariable()` method. The only restriction is that the variable must be a serializable Java object.

For example, you can generate two strings at run time and display one string when rendering in PDF and a different string in other format contexts. Use the `setPersistentGlobalVariable` method at generation time to store the variables in the report document. At render time, retrieve the variables in a render event handler using the `getPersistentGlobalVariable` method.

The following example calls the `setPersistentGlobalVariable` method in the `onCreate` event handler of a label element to initialize two strings then calls the `getPersistentGlobalVariable` method in the `onRender` event handler to specify the format context.

```
//Label onCreate  
reportContext.setPersistentGlobalVariable("mypdfstr", "My PDF  
String");  
reportContext.setPersistentGlobalVariable("myhtmlstr", "My HTML  
String");  
...  
//Label onRender
```

```

if( reportContext.getOutputFormat() == "pdf" ){
    this.text =
    reportContext.getPersistentGlobalVariable("mypdfstr");;
}else{
    this.text =
    reportContext.getPersistentGlobalVariable("myhtmlstr");;
}

```

Using getAppContext

BIRT uses an Application Context Map to store values and objects for use in all phases of report generation and presentation. To add an object to the application context, use the Report Engine API or the setAttribute() method of the request object.

The following JSP example adds an object to the application context. ApplicationContextKey is the name that references the object and ApplicationContextValue contains the value of the object.

```

<%
java.lang.String teststr = "MyTest";
request.setAttribute( "ApplicationContextKey", teststr );
java.lang.String stringObj = "This test my Application Context
From the Viewer";
request.setAttribute( "ApplicationContextValue", stringObj );
%>
<jsp:forward page= "<%= "/run?__report=ApplicationContext.rptdesign"
%>" />

```

Once the object is in the application context, you can reference it by name. The following example uses a try catch block to detect whether the object is not found and returns an appropriate error message.

```

try {
    MyTest.toString( );
} catch (e) {
    "My Object Was Not Found";
}

```

When using the Report Engine API, you can add an object using the EngineConfig class or the EngineTask class.

```

config = new EngineConfig( );
HashMap hm = config.getAppContext( );
hm.put( "MyTest", stringObj);
config.setAppContext(hm);

```

You can use the reportContext.getAppContext method to iterate through all the objects in the application context, as shown in the following example:

```

iter = reportContext.getAppContext( ).entrySet().iterator();
siz = reportContext.getAppContext( ).size();
kys = "";

```

```

vls = "";
while (iter.hasNext( )) {
    innerObject = iter.next( );
    kys = kys + innerObject.getKey( ) + "\n";
    val = innerObject.getValue( );
    if( val != null ){
        vls = vls + innerObject.getValue( ).toString( ) + "\n";
    }else{
        vls = vls + "NULL" + "\n";
    }
}

```

You can display the variables, kys and vls, in a data element or text element by entering kys and vls in an expression for a data element or <VALUE-OF>kys</VALUE-OF> and <VALUE-OF>vls</VALUE-OF> in a text element.

Getting information from an HTTP request object

The HTTP servlet request object contains various methods to retrieve information about the request to run the report. One useful method of the HTTP request object gets the query string that follows the path in the request URL. The query string contains all the parameters for the request. By parsing the query string, the code can extract the parameters in the request URL to conditionally determine the report output. For example, you can use this feature to pass in a user ID or to set or override a report parameter.

The following code gets the query string:

```

importPackage( Packages.java.util );
httpServletReq = reportContext.getHttpServletRequest( );
formatStr=httpServletReq.getQueryString( );

```

You can retrieve the Session object using reportContext, as shown in the following code:

```

var request = reportContext.getHttpServletRequest();
var session = request.getSession();
session.setAttribute("ReportAttribute", myAttribute);

```

Using the this object

Every JavaScript event handler is associated with a particular ROM element, such as a report item, a data source, a data set, or the report itself. Most report elements have properties that an event handler can access and, in some cases, change. Many report elements also have functions that you can call. A JavaScript event handler can access these properties and functions through a special object called the this object. For more information about report item properties, see Chapter 7, “Understanding the Report Object Model.”

Using this object methods

The this object represents the element for which the event handler is handling events. To use the this object, type the keyword, this, followed by a period in the script window for the event handler you are writing. At the time you type the period, a scrollable list of all the properties and functions for the element pops up, as shown in Figure 9-2.

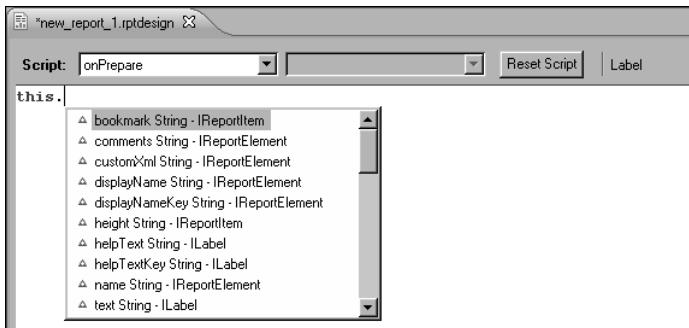


Figure 9-2 Using the this object to display a list of functions and properties

Scroll down the list using the arrow keys or the scroll bar controls and press Enter or double-click when the property or function you want is highlighted.

Using the this object to set the property of a report item

You can also use this pop-up list to select a method or property for other objects. The following procedure sets the background color of a label to yellow. The general process explained in this procedure is not specific to the label report item. You can modify all report item event handlers in the same way.

How to set a property of a report item using JavaScript

- 1 Select the label whose color you want to change by navigating in the Outline view to select the appropriate report item, as shown in Figure 9-3.

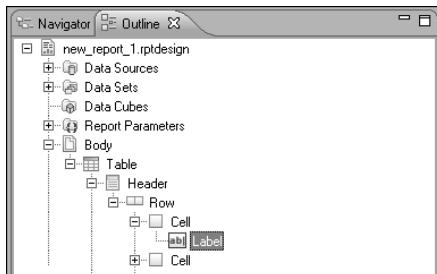


Figure 9-3 Selecting a report item to modify

- 2** Select onPrepare from the drop-down list in the Script window, as shown in Figure 9-4.

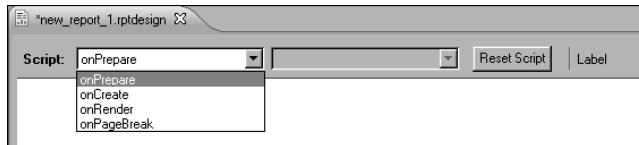


Figure 9-4 Selecting onPrepare()

- 3** Enter the keyword this, followed by a period in the onPrepare script window to open the scrollable list of properties and functions, as shown in Figure 9-5.

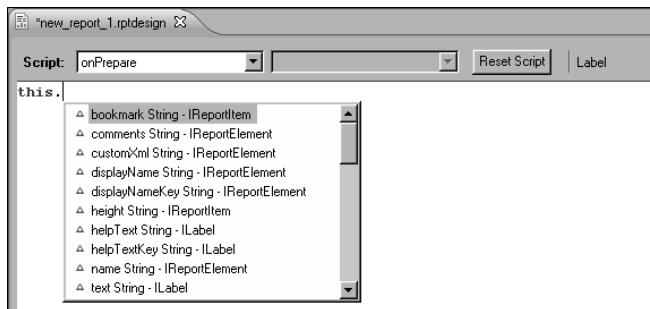


Figure 9-5 Using the this object

- 4** Select the getStyle() method from the list. The onPrepare script window appears as shown in Figure 9-6.



Figure 9-6 The onPrepare script window

- 5** Move the cursor to the end of the line in the onPrepare script window and type a period. The scrollable list of properties and functions of the Style element appears, as shown in Figure 9-7.

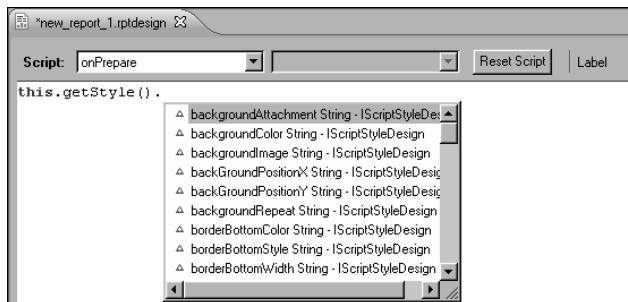


Figure 9-7 Properties and functions of the Style element

- 6 Select backgroundColor from the list of style properties and functions.
- 7 Complete the line of JavaScript in the onPrepare script window by appending ="yellow" as shown in Figure 9-8.



Figure 9-8 Changing the color of an element

- 8 Choose Preview to see the effect of the onPrepare event handler script. The label appears in the report with a yellow background, as shown in Figure 9-9.

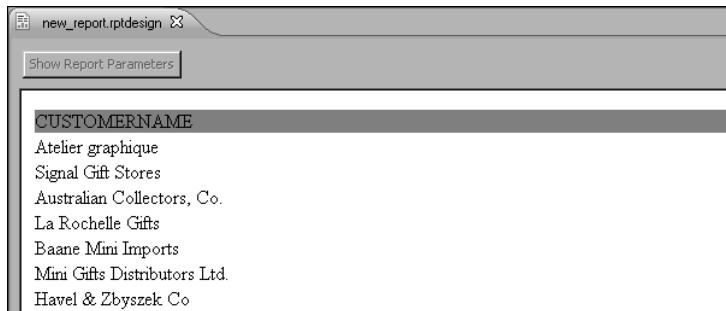


Figure 9-9 Preview of the color change

Using the row object

The row object provides access to the columns of the current row from within the DataSet.onFetch() method. You can retrieve the value of any column, using the column name in a statement similar to the following examples:

```
col1Value = row["custNum"];
col1Value = row.custNum;
```

You can only index the column position with the column name if the name is a valid JavaScript name with no spaces or special characters. Alternatively, you can use the column alias if the alias is a valid JavaScript name.

You can also get a column value by numerically indexing the column position, as shown in the following statement:

```
col1Value = row[1];
```

When you index the column position numerically, the number inside the brackets is the position of the column, beginning with 1. You can retrieve the row number with `row[0]`.

Although you use array syntax to access the row object in JavaScript, this object is not a JavaScript array. For this reason, you cannot use JavaScript array properties, such as `length`, with the row object.

Getting column information

The `DataSet` object has a method called `getColumnMetaData()`, which returns an `IColumnMetaData` object. The `IColumnMetaData` interface has methods that provide information about the columns in a data set, as shown in Table 9-2.

Table 9-2 Methods of the `IColumnMetaData` interface

Method	Returns
<code>getColumnAlias()</code>	Alias of the specified column
<code>getColumnCount()</code>	Number of columns in a row of the result set
<code>getColumnLabel()</code>	Column label
<code>getColumnName()</code>	Column name at the specified index
<code>getColumnNativeTypeName()</code>	One of the following data types: <ul style="list-style-type: none">■ <code>BOOLEAN</code>■ <code>DATETIME</code>■ <code>DECIMAL</code>■ <code>FLOAT</code>■ <code>INTEGER</code>■ <code>STRING</code> The data type is null if the column is a computed field or if the type is not known
<code>getColumnType()</code>	Data type of the column at the specified index
<code>getColumnTypeName()</code>	Data type name of the column at the specified index
<code>isComputedColumn()</code>	True or false depending on whether the column is a computed field

You get the IColumnMetaData object from the dataSet object, as shown in the following statement:

```
columnMetaData = this.getColumnMetaData( );
```

You can use the count of columns to iterate through all the columns in the data set, as shown in the following example:

```
colCount = columnMetaData.getColumnCount( );
for ( i = 0; i < colCount; i++ )
{
    pw.println( "Column val for col position " + i + " = " +
                row[i] );
    pw.println( "Column name for col position " + i + " = " +
                columnDefinitions[i].name );
}
```

Getting and altering the query string

You get the text of the query in any DataSet event handler as shown in the following example:

```
query = this.queryText;
```

You can modify a query in the DataSet beforeOpen() event handler by setting the value of the queryText string. To change the query, set the queryText string to a valid SQL query, as shown in the following example:

```
queryText = "select * from CLASSICMODELS.CUSTOMERS
            WHERE CLASSICMODELS.CUSTOMERS.CUSTOMERNUMBER
            BETWEEN + params[lownumber].value + AND
            + params[highnumber].value;
```

One advantage of dynamically altering the query is that you can use business logic to determine the proper query. This approach can be more flexible than using parameters.

Changing the connection properties of a data source

You can change the run-time connection properties of a data source by accessing the extensionProperties array of the DataSource object. The ODA extension defines the list of connection properties that can be set at run time.

Table 9-3 describes the JDBC data source properties that affect the connection at run time.

Table 9-3 JDBC data source run-time connection properties

Property	Description
odaUser	Login user name
odaPassword	Login password
odaURL	URL that identifies the data source
odaDriverClass	Driver class for accessing the data source

To change these properties, add code similar to the following statements in the `DataSource.beforeOpen` method:

```
extensionProperties.odaUser = "JoeUser";
extensionProperties.odaPassword = "openSesame";
extensionProperties.odaURL = "jdbc:my_data_source:xxx";
extensionProperties.odaDriverClass =
    "com.companyb.jdbc.Driver";
extensionProperties.odaJndiName = "java:/MySQLDs";
extensionProperties.OdaConnProfileName = "myprofile";
extensionProperties.OdaConnProfileStorePath = "c:/conprof";
```

You can also set these properties using the following syntax:

```
this.setExtensionProperty("odaURL","jdbc:mysql://localhost/
mysql");
```

In addition they can be altered using script in the `beforeFactory` event as follows:

```
report = reportContext.getReportRunnable().designHandle
    .getDesignHandle();
dsHandle = report.findDataSource("Data Source");
dsHandle.setProperty("odaDriverClass", "myDriver");
dsHandle.setProperty("odaURL", "myUrl");
```

You can also set data source and data set properties using the property binding feature in the data source and data set editor.

Getting a parameter value

A script can get the value of a report parameter by passing the name of the parameter to the `getParameterValue()` method of the `reportContext` object. The following statement gets the value of the `UserID` parameter:

```
userID = reportContext.getParameterValue( "UserID" );
```

You can also retrieve parameter values using the `params` BIRT global variable in a statement that has the following syntax:

```
userID = params["UserID"].value;
```

BIRT 2.2.1 also supports dynamic parameters that allow selecting multiple values. You can access these parameters using the following syntax:

```
reportContext.getParameterValue( "MultiParm" )[0];
```

or

```
params["MultiParm"].value[0];
```

A statement that uses the syntax in the previous example retrieves the first value. You can determine the number of selected values with a statement that uses the following syntax:

```
reportContext.getParameterValue( "MultiParm" ).length;
```

or

```
params["MultiParm"].value.length;
```

You can use these features when implementing a beforeOpen event handler to apply an IN clause to the query as shown in the following example:

```
var parmcount = params["parmorders"].value.length  
var whereclause = "";  
if( parmcount > 0 ){  
    whereclause = " where customernumber in ( ";  
}  
for( i=0; i < parmcount; i++ ){  
    if( i == 0 ){  
        whereclause = whereclause +  
        params["parmorders"].value[i];  
    }else{  
        whereclause = whereclause + " , " +  
        params["parmorders"].value[i];  
    }  
}  
if( parmcount > 0 ){  
    this.queryText = this.queryText + whereclause + " ) ";  
}
```

Determining method execution sequence

You can determine the method execution sequence by writing code that generates a file containing a line for every method that you want to track.

To create an output file that contains the method execution sequence include initialization code in the ReportDesign.initialize method and finalization code in the ReportDesign.afterFactory method. In each method that you want to track, add code to write a line of text to the output file. It is easier to write the code in JavaScript than Java, but it is possible to write analogous code in Java.

The following sections show you how to use JavaScript to determine method execution sequence.

Providing the ReportDesign.initialize code

The following code in the ReportDesign.initialize method creates a file on your hard drive and adds one line to the file.

```
importPackage( Packages.java.io );
fos = new java.io.FileOutputStream( "c:\\logFile.txt" );
printWriter = new java.io.PrintWriter( fos );
printWriter.println( "ReportDesign.initialize" );
```

The preceding code performs the following tasks:

- Imports the Java package, java.io
- Creates a file output stream for the file you want to create
- Creates a PrintWriter object that every method can use to track method execution sequence

How to provide code for the ReportDesign.initialize method

You can provide code for the ReportDesign.initialize method by performing the following steps:

- 1 Choose the Script tab.
- 2 Choose the Outline view.
- 3 In Outline, select the top line, as shown in Figure 9-10.

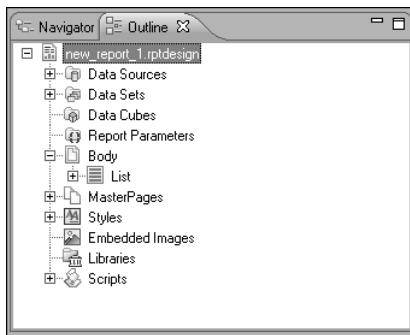


Figure 9-10 Selecting the report design

- 4 In Script, select the Initialize() method.
- 5 Type the code into the script editor.

The BIRT Report Designer appears, as shown in Figure 9-11.

A screenshot of a software window titled "new_report_1.rptdesign". The window shows a code editor with the following Java code:

```
importPackage( Packages.java.io );
fos = new java.io.FileOutputStream( "c:\\\\logFile.txt" );
printWriter = new java.io.PrintWriter( fos );
printWriter.println( "ReportDesign.initialize" );
```

Figure 9-11 Providing ReportDesign.initialize code

Providing code for the methods you want to track

For every method that you want to track, provide a single statement generating a line of output to your log file, as shown in the following statement:

```
printWriter.println( "Table.onRow" );
```

To provide code for a report item method you want to track, first select the appropriate object from Outline and select the appropriate method from the method selection list. Then use the same steps for entering code into a method, as shown in the preceding section.

To provide code for a data source or data set method, select the appropriate data source or data set from Data Explorer before selecting the method you want to track.

Providing the ReportDesign.afterFactory code

The following statement in the ReportDesign.afterFactory method closes the file.

```
printWriter.close( );
```

Using this method flushes all the buffers and ensures that all method output appears in the file.

To provide the ReportDesign.afterFactory code, select the top line of the outline and select the afterFactory method on the code page.

Tutorial 1: Writing an event handler in JavaScript

This tutorial provides instructions for writing a set of event handlers. The tutorial assumes that you have a basic report design based on the Classic Models, Inc. Sample Database. The only requirement for the starting report design is that it contains a table of customers with a column for the customer name. In this tutorial you count the customers whose names contain the string "Mini" and display the result in a pop-up window.

In this tutorial, you perform the following tasks:

- Open the report design
- Create and initialize a counter in the Table.onCreate() method
- Conditionally increment the counter in the Row.onCreate() method
- Display the result using the ReportDesign.afterFactory() method

Task 1: Open the report design

Open a report design that uses the Classic Car sample database and displays a table of customer names.

- 1 If necessary, open Navigator by choosing Window→Show View→Navigator.
- 2 Double-click the appropriate report design. The file opens in the layout editor, as shown in Figure 9-12.

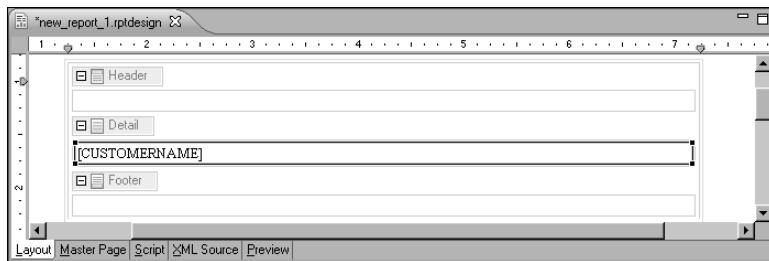


Figure 9-12 Report design in the layout editor

Task 2: Create and initialize a counter in the Table.onCreate() method

In order to count the number of customers whose names contain the string Mini, you must first declare a global counter and set its value to zero. The Table.onCreate() method is the most appropriate place to perform this task because Table.onCreate() executes before retrieving any rows. You conditionally increment this counter in the Row.onCreate() method.

- 1 In Layout, select the table by placing the cursor near the bottom left corner of the table. The table icon appears, as shown in Figure 9-13.

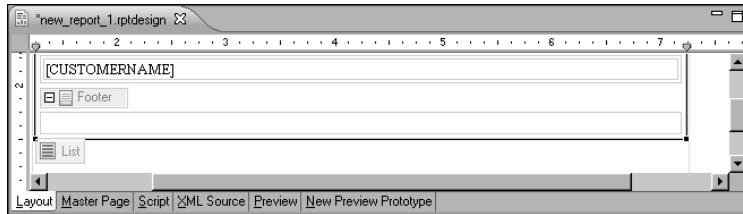


Figure 9-13 Table icon in the layout editor

- 2** Choose the Script tab. The script tab appears, as shown in Figure 9-14.

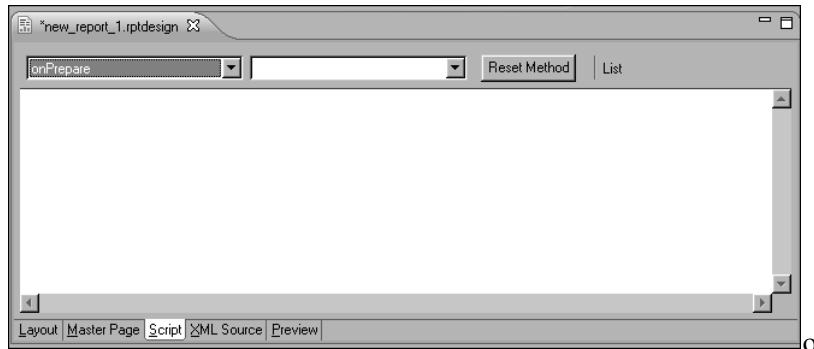


Figure 9-14 Script window

- 3** Type the following line of code in the script window for the onCreate() method:

```
countOfMinis = 0;
```

- 4** To run the report and verify that the code did not create any errors, choose Preview.
5 Scroll to the bottom of the report, where JavaScript error messages appear. If there are no errors, the report appears, as shown in Figure 9-15.

If you see an error message, you may have typed a statement incorrectly. If so, go back to the script window, select the method you just modified, correct the error, and choose Preview again.

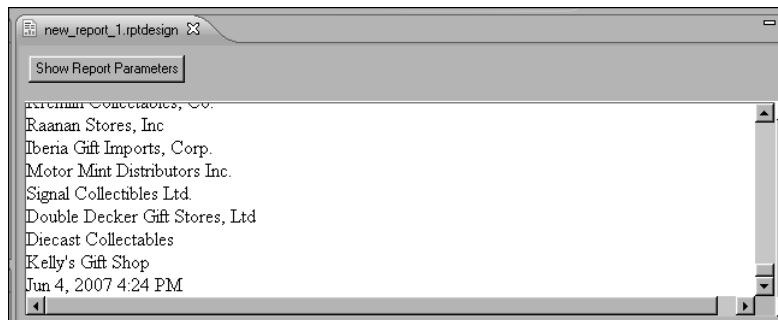


Figure 9-15 Report preview

Task 3: Conditionally increment the counter in the Row.onCreate() method

To count the number of customers with the string Mini in their names, you must examine each customer's name and add one to the counter for every occurrence. A logical place to perform this task is in the Row.onCreate()

method, which executes with every retrieval of a row of data from the data source.

- 1 In Layout, select the Row and choose Script.
- 2 Pull down the list of methods at the top of the script window and select onCreate, as shown in Figure 9-16.

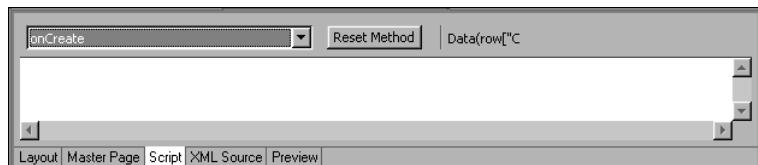


Figure 9-16 onCreate() in the script window

- 3 Enter the following line of JavaScript code in the Script window:

```
row=this.getRowData( );
```

Notice that when you enter the period after this, a pop-up appears containing all the available methods and properties, including getRowData. This line of code gets an instance of IRowData, which has a method, getExpressionValue(), to get the contents of a column of the row.

- 4 Type the following line of JavaScript below the line you just entered:

```
CustName=row.getExpressionValue( "row[CUSTOMERNAME]" );
```

This line of code returns the contents of the table column that comes from the CUSTOMERNAME column in the data set.

- 5 Type the following line of code to conditionally increment the counter you created in Task 2: “Create and initialize a counter in the Table.onCreate() method.”

```
if( CustName.indexOf( "Mini" ) != -1 ) countOfMinis += 1;
```

You can use the JavaScript palette to insert each of the following elements in the preceding line:

- indexOf()

Select Native (JavaScript) Objects→String Functions→indexOf()

- !=

Select Operators→Comparison→!=

- +=

Select Operators→Assignment→+=

- 6 Choose Preview to run the report again to verify that the code you entered did not create any errors.

Task 4: Display the result using the ReportDesign.afterFactory() method

To display the count of customers with the string Mini in their names, you insert code in a method that runs after the processing of all the rows in the table. One logical place for this code is in the ReportDesign.afterFactory() method.

- 1 In Outline, select the report design, as shown in Figure 9-17.

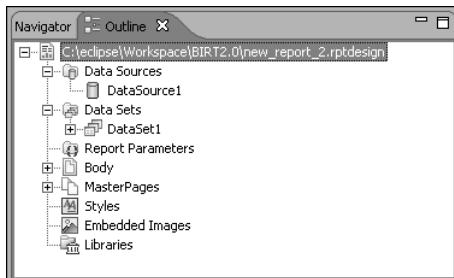


Figure 9-17 Selecting the report design in Outline

- 2 Select the afterFactory() method from the script window drop-down list.
- 3 Type the following code into the afterFactory() method:

```
importPackage( Packages.javax.swing );
frame = new JFrame( "Count of Minis = " + countOfMinis );
frame.setBounds( 310, 220, 300, 20 );
frame.show( );
```
- 4 Select Preview to see the results. If there are no errors in the code, you see a report similar to the one in Figure 9-18.

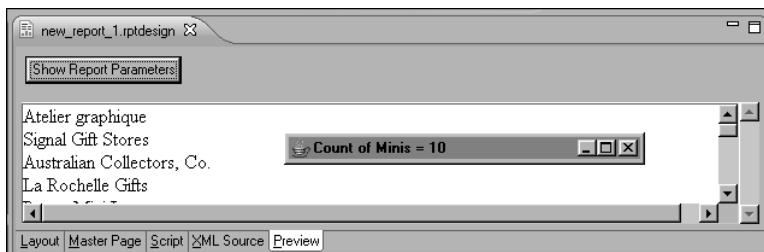


Figure 9-18 Result of changing the afterFactory() method

If you do not see the Count of Minis window, look for it behind the Eclipse window. If the Count of Minis window does not appear, the most likely reason is a scripting error caused by an error in one of your code entries.

If you suspect that a scripting error occurred, scroll to the bottom of the report where all scripting error messages appear. In most situations, there is a brief error message next to a plus sign (+). The plus sign indicates that there is a more detailed error message that is only visible after you expand the brief

error message. To expand the brief error message, choose the plus sign. Scroll down to see the more detailed error message.

JavaScript event handler examples

The following examples illustrate some of the common functions that you can use while writing JavaScript event handlers.

JavaScript onPrepare examples

The onPrepare event fires for all report elements during the preparation phase of the generation process. This location is ideal for changing the design of a particular report element before generating the individual report item instances.

For example, BIRT supports grouping in tables and lists. If you want to permit the user to group the data, you can implement this feature by performing the following developer tasks:

- Add a group to a table
- Select the group in the outline view of the report
- Create an onPrepare script that changes the group definition for the table

The following code example groups a table by country or city using a boolean parameter to determine which grouping column to use:

```
this.sortDirection = "desc"
if( params[ "grp_p" ].value == true ){
    this.keyExpr = "row[ 'CITY' ];";
    grpname = "City";
} else{
    this.keyExpr = "row[ 'COUNTRY' ];";
    grpname = "Country";
}
```

JavaScript onCreate examples

The onCreate event fires when the generation process creates an instance of the report element. Writing an event handler for an onCreate event is useful when changing an instance of a report item. The following examples illustrate this concept.

Image elements can source image data from a URL, an embedded image in a report, an image file in the resource folder, or a blob type from a database. You can also retrieve the data for the image at run time, using an onCreate event handler. The following example illustrates how to use Java to read the data from an image stored on the local file system.

```
importPackage( Packages.java.io );
```

```

importPackage( Packages.java.lang );

var file = new File( "c:/temp/test.png" );
var ist = new FileInputStream( file );
var lengthi = file.length();

bytesa = new ByteArrayOutputStream( lengthi );
var c;
while( ( c=ist.read( ) ) != -1 ){
    bytesa.write( c );
}
ist.close( );
this.data = bytesa.toByteArray( );

```

If the image element is in a table, you can choose the selected file based on the data retrieved from the dataset. For example, the following onCreate script selects an image from the resource folder based on calculations made on the table bound columns.

```

if ( ( ( this.getRowData( ).getColumnValue( "total2005" ) / 5 )
    / ( this.getRowData( ).getColumnValue( "total2004" ) / 12) ) >=
    .96 ) {
    this.file ="up.gif";
}
else if( ( ( this.getRowData( ).getColumnValue( "total2005" ) /
    5 ) / ( this.getRowData( ).getColumnValue( "total2004" )
    ) /
    12 ) ) <= .85 ) {
    this.file ="down.gif";
}
else {
    this.file ="even.gif";
}

```

BIRT supports conditional element formatting, using the highlight editor. You can also use the onCreate event handler to do conditional formatting. Consider the following example:

```

if( this.getRowData( ).getColumnValue( "QUANTITYORDERED" ) > 40
){
    this.getStyle( ).backgroundColor = "yellow";
    this.getStyle( ).verticalAlign = "Middle";
    this.height = "2cm";
}

```

This example is an onCreate event handler for a table row. If the table column QUANTITYORDERED is greater than forty, the row height enlarges to two centimeters, highlighted in yellow, and the content centers in the middle of the row.

JavaScript onRender examples

Rendering content may occur anytime after generation. Often users run a report and re-render it many times in different formats. The onRender events allow some customization at render time. For example, a user may want a label in the page footer to show the render time of the report or the day of the month the report rendered. You can achieve this effect by using the onRender method of a label element, as shown in the following the following script:

```
importPackage( Packages.java.util );
c1 = new GregorianCalendar();
this.text = c1.get( Calendar.DAY_OF_MONTH );
```

Calling Java from JavaScript

Rhino provides excellent integration with Java classes, allowing a BIRT script to work seamlessly with business logic written in Java. Wrapping Java in JavaScript allows the developer to write powerful scripts quickly by leveraging both the internal and external libraries of existing Java code. You can use static methods, non-static methods, and static constants of a Java class.

Understanding the Packages object

The Packages object is the JavaScript gateway to the Java classes. It is a top-level Rhino object that contains properties for every top-level Java package, such as java and com. The Packages object also contains a property for every package that fit finds in its classpath. You can use the Packages object to access a Java class for which Packages has a property by preceding the class name with Packages, as shown in the following statement:

```
var nc = new Packages(javax.swing.JFrame);
```

You can also use the Packages object to reference a Java class that is not a part of a package, as shown in the following statement:

```
var nc = new Packages.NumberConversion();
```

For BIRT to find a custom Java class or package, you must place it in the BIRT classpath, as discussed later in this chapter.

Understanding the importPackage method

You can avoid writing a fully qualified reference to a Java class by using the top-level Rhino method importPackage(). The importPackage() method functions like a Java import statement. Use the importPackage() method to specify one or more Java packages that contain the Java classes that you need to access, as shown in the following statement:

```
importPackage( Packages.java.io, Packages(javax.swing) );
```

You must prepend Packages to the name of each package. After the first time BIRT executes a method containing the importPackage() method, the specified packages are available to all succeeding scripts. For this reason, you should include the importPackage() method in the ReportDesign.initialize method, which is always the first method that BIRT executes.

Java imports java.lang.* implicitly. Rhino, on the other hand, does not import java.lang.* implicitly because JavaScript has several top-level objects with the same names as some classes defined in the java.lang package. These classes include Boolean, Math, Number, Object, and String. Importing java.lang causes a name collision with the JavaScript objects of the same name. For this reason, you should avoid using importPackage() to import java.lang.

Using a Java class

To use a Java class in a BIRT script, you set a JavaScript object equal to the Java object. You then call the Java class methods on the JavaScript object. The following example creates a Java Swing frame and sets the JavaScript object named frame to the Java JFrame object. Then the code calls the setBounds() and show() methods directly on the JavaScript object.

```
importPackage( Packages.java.awt );
frame = new JFrame( "My Frame" );
frame.setBounds( 300, 300, 300, 20 );
frame.show( );
```

The effect of this code example is to display a Java window on your desktop containing the title, My Frame. This example works only in the BIRT Report Designer.

Placing Java classes

For BIRT report viewer to find Java classes, the classes must be under:

```
$ECLIPSE_INSTALL\plugins org.eclipse.birt.report
.viewer_*\birt\WEB-INF\classes
```

or package the classes as a JAR file put in into

```
$ECLIPSE_INSTALL\plugins org.eclipse.birt.report
.viewer_*\birt\WEB-INF\lib
```

If you deploy the example Viewer to an application server, you must also deploy these classes or JAR files to the run-time environment.

Place event handlers and Java classes in the SCRIPTLIB directory as defined in the web.xml of the web viewer application. If using the SCRIPTLIB directory, the classes should be in JAR format.

If you are using the Report Engine API, the classes must be in the classpath. Alternatively, you can add a system variable that the report engine adds to the classpath as follows:

```
System.setProperty( EngineConstants.WEBAPP_CLASSPATH_KEY,
```

```
"c:/myjars/class.jar" );
```

If you are using the Report Engine API, you can set the parent classloader for the engine, by setting the APPCONTEXT_CLASSLOADER_KEY in the application context, as shown in the following code:

```
config = new EngineConfig();
HashMap hm = config.getAppContext();
hm.put( EngineConstants.APPCONTEXT_CLASSLOADER_KEY,
        YourClass.class.getClassLoader() );
config.setAppContext( hm );
```

Issues with using Java in JavaScript code

There are many nuances to writing Java code, such as how to handle overloaded methods, how to use interfaces, and so forth. For more information on these topics, refer to the Rhino page on scripting Java at <http://www.mozilla.org/rhino/ScriptingJava.html>.

Calling the method of a class that resides in a plug-in

Both Java and JavaScript event handlers have access to all the public methods of any class that resides in an Eclipse plug-in. The plug-in can be one of the core Eclipse plug-ins, a plug-in supplied by a third party, or one of your own creations. As long as the plug-in is available to the BIRT report at run time, a BIRT script has access to all the public methods of all the classes within that plug-in.

The following JavaScript code example shows how to call a method of a Java class that resides in an Eclipse plug-in:

```
importPackage( Packages.org.eclipse.core.runtime );
mybundle = Platform.getBundle( "org.eclipse.myCorp.security" );
validateClass = mybundle.loadClass(
    "org.eclipse.myCorp.security.Validate" );
validateInstance = validateClass.newInstance();
var password = validateInstance.getPass( loginID );
```

In the example, the first statement makes the Eclipse core package, org.eclipse.core.runtime, available to the JavaScript program. This package contains two classes, Platform and Bundle, which are necessary to the rest of the program.

The Platform class contains a static method, getBundle(), that returns a Bundle object. The sole argument to getBundle() is the name of the plug-in that contains the target class.

The `Bundle` class contains a `loadClass()` method that returns a `java.lang.Class` object. The only argument to `loadClass()` is a fully qualified class name, which in this case is `org.eclipse.myCorp.security.Validate`.

The `java.lang.Class` class represents the target class and contains a `newInstance()` method that returns an instance of the class. The `newInstance()` method creates the instance using the default constructor, which has no arguments. The final statement in the example calls the target method of the newly instantiated object of the target class.

The Java equivalent of the previous JavaScript example is:

```
#import org.eclipse.core.runtime.Bundle;
#import org.eclipse.core.runtime.Platform;
#import org.eclipse.myCorp.security.Validate;

Bundle mybundle = Platform.getBundle(
    "org.eclipse.myCorp.security" );
java.lang.Class validateClass = mybundle.loadClass(
    "org.eclipse.myCorp.security.Validate" );
Validate validateInstance = validateClass.newInstance();
String password = validateInstance.getPass( loginID );
```

10

Using Java to Write an Event Handler

Creating a Java event handler is slightly more complex than creating a JavaScript event handler. You cannot simply enter Java code directly in the BIRT Report Designer.

To create a Java event handler class, you must compile the source for the Java class and make certain that the class is visible to BIRT. Creating a Java event handler for BIRT is simplified, however, by the fact that Eclipse is a robust Java development environment and supports integrating a Java project with a BIRT project.

Writing a Java event handler class

When you provide one or more Java event handlers for a scriptable BIRT element, you must create one class that contains all the Java event handlers for that element. Creating a class that contains event handler methods for more than one element is not advisable.

BIRT provides a set of Java interfaces and Java adapter classes to simplify the process of writing a Java event handler class. There is one interface and one adapter class for every scriptable BIRT element.

An element's event handler interface defines all the event handler methods for that element. A handler class must implement every method defined in the interface, even if some of the methods are empty. You only provide code for the event handlers that you want to implement.

Locating the JAR files that an event handler requires

There are two JAR files that contain all the classes and interfaces that an event handler requires. One of the JAR files is a part of BIRT Report Designer and SDK and the other one is a part of BIRT Report Engine. You can use either when developing a custom event handler.

The first JAR file that you can use for developing a Java event handler is `org.eclipse.birt.report.engine_<version>.jar`, which is located in the Eclipse plugins directory for BIRT Report Designer and SDK.

The second JAR file that you use when you develop and deploy your report is `scriptapi.jar`, which is located in the `\WebViewerExample\WEB-INF\lib` directory of BIRT Report Engine.

All JAR files in the `\WebViewerExample\WEB-INF\lib` directory are in the deployed report classpath, so there is no need to do anything special to make `scriptapi.jar` accessible at run time. If you are using the `scriptapi.jar` file in development phase, you need to download the Report Engine download and reference `scriptapi.jar` from within the buildpath.

The required JAR files are also in the `ReportEngine\lib` directory of BIRT Report Engine. If you are using the Report Engine API to run your reports, add the jars in this directory to the classpath and buildpath. If you are using the Report Engine plug-in in an Eclipse application, this step is not required, because the plug-in already contains the dependency and classpath entries.

Extending an adapter class

An element adapter class implements the element interface and provides empty stubs for every method. To use the adapter class, extend the adapter class and override the methods for which you are providing handler code. Eclipse recommends extending an adapter class rather than implementing an interface directly.

BIRT naming conventions for the event handler interfaces and adapter classes are discussed later in this chapter.

How to create an event handler class and add it to the Java project

This section describes the process for using the Eclipse Java development environment to create an event handler class for a scriptable BIRT element.

- 1 Add `org.eclipse.birt.report.engine_<version>.jar` to your Java project, as outlined in the following steps:
 - 1 Select your Java project and choose `File->Properties->Java Build Path->Libraries`. Java Build Path appears, as shown in Figure 10-1.

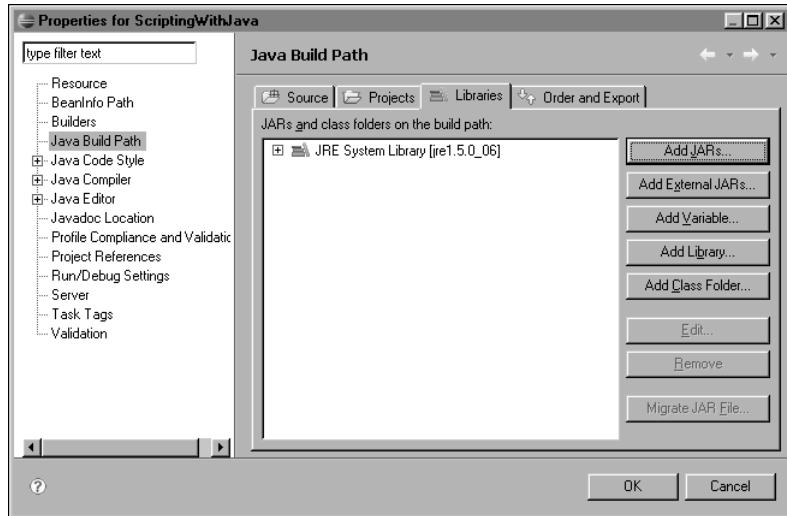


Figure 10-1 Adding a JAR file to the compiler's classpath

- 2 Choose Add External JARs. JAR Selection appears.
- 3 Navigate to Eclipse /plugins directory. In a default Eclipse installation, this directory is in the following location:
`<ECLIPSE_INSTALL>\eclipse\plugins`
- 4 Select `org.eclipse.birt.report.engine_<version>.jar`. Choose Open. Java Build Path reappears.
- 5 Choose OK.

- 2 Select your Java project and choose **File**→**New**→**Other**. Select a wizard appears.
- 3 Expand Java and select Class, as shown in Figure 10-2.

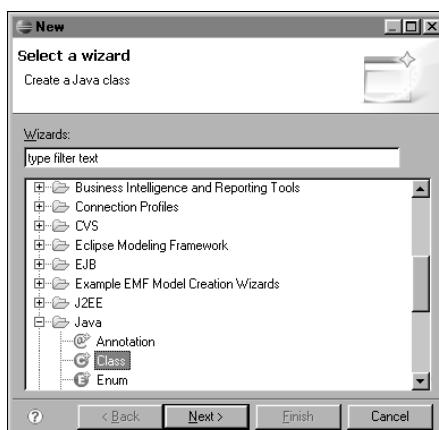


Figure 10-2 The Select a wizard dialog

Choose Next. New Java Class appears, as shown in Figure 10-3.

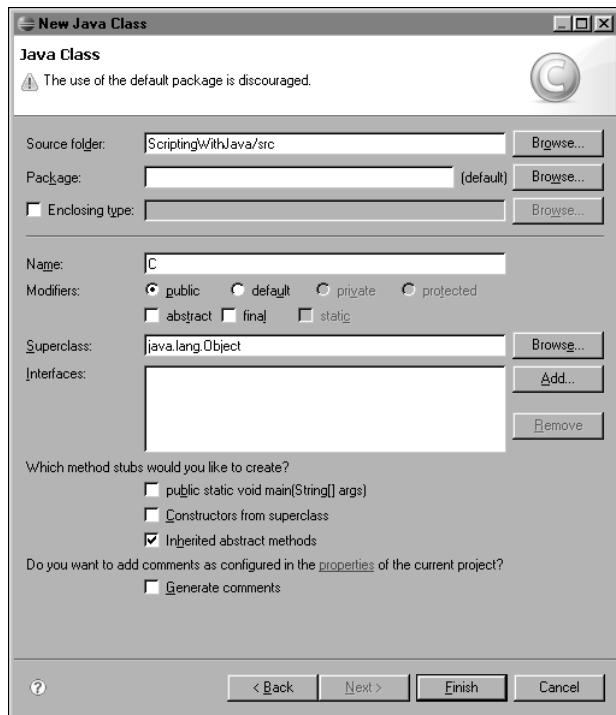


Figure 10-3 New Java Class

- 4 Navigate to the folder where you want the Java source file to reside by choosing the Browse button beside Source Folder.
- 5 If your new Java class is a part of a package, type the fully qualified package name in Package.
- 6 In Name, type a name for your class.
- 7 In Modifiers, choose Public.
- 8 Choose the Browse button beside Superclass. Superclass Selection appears, as shown in Figure 10-4.

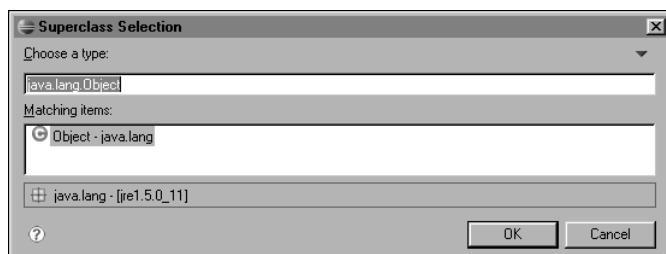


Figure 10-4 Superclass Selection

- 9 In Choose a type, type the name of the adapter class for the ROM element. For example, enter Label EventAdapter for the Label element. Choose OK. New Java Class reappears.
- 10 Select Generate comments. Choose Finish. A Java editor view appears, similar to the one shown in Figure 10-5.

```

LabelEventHandler.java

import org.eclipse.birt.report.engine.api.script.eventadapter.LabelEventAdapter;
import org.eclipse.birt.report.engine.api.script.IReportContext;
import org.eclipse.birt.report.engine.api.script.element.ILabel;
import org.eclipse.birt.report.engine.api.script.eventadapter.LabelEventAdapter;

public class LabelEventHandler extends LabelEventAdapter {
}

```

Figure 10-5 The Java editor

- 11 Add the event handler method for your new event handler class. Figure 10-6 shows the addition of an `onPrepare()` method that sets the background color of the label to red.

```

LabelEventHandler.java

import org.eclipse.birt.report.engine.api.script.IReportContext;
import org.eclipse.birt.report.engine.api.script.element.ILabel;
import org.eclipse.birt.report.engine.api.script.eventadapter.LabelEventAdapter;
import org.eclipse.birt.report.engine.api.script.eventadapter.LabelEventAdapter;

public class LabelEventHandler extends LabelEventAdapter {
    public void onPrepare(ILabel arg0, IReportContext arg1) {
        try{
            arg0.getStyle().setBackgroundColor("red");
        }
        catch(Exception e){}
    }
}

```

Figure 10-6 The `onPrepare()` method in the Java editor

Making the Java class visible to BIRT

One way to make a Java event handler class visible to the BIRT report designer is to create a Java development project for compiling the class in the same workspace as your BIRT report project. The other option is to place the class in a directory or JAR file specified in the BIRT classpath. When you deploy the report to an application server, however, you must copy the Java class to the appropriate location on the server.

Associating the Java event handler class with a report element

After you create the Java event handler class and code the appropriate handler methods, you must associate the class with the appropriate report element.

How to associate a Java class with a report element

The example in this procedure makes the following assumptions:

- The report design includes a scriptable report item, such as a label.
 - A Java class containing event handler methods for the scriptable report item is visible to BIRT.
- 1 In Outline, select the report element for which an event handler class is visible to BIRT, as shown in Figure 10-7.

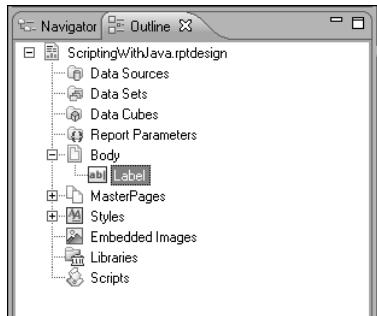


Figure 10-7 Selecting a report element

- 2 In Property Editor for the selected report element, select Event Handler and enter the fully qualified name of the event handler class, as shown in Figure 10-8.



Figure 10-8 The event handler class name

BIRT Java interface and class naming conventions

BIRT event handler classes and interfaces use the following naming conventions for consistency:

- Event handler interfaces

All BIRT ROM element interface names begin with the letter I, which is followed by the name of the ROM element and then EventHandler. For example, the interface for the Label element is ILabelEventHandler.

- Event handler adapter classes

All BIRT ROM element adapter class names begin with the name of the element, followed by EventAdapter. For example, the name of the adapter class for a Label element is LabelEventAdapter.

- ROM element instance interfaces

All BIRT ROM element instance interface names begin with the letter I, followed by the name of the element and then Instance. For example, the ROM element instance interface for a Label element is ILabelInstance.

- ROM element design interfaces

All BIRT ROM element instance design interface names begin with the letter I, followed by the name of the element. For example, the design interface for a Label element is ILabel.

Writing a Java event handler

Most scriptable elements have more than one event for which you can write a handler. If you write an event handler for any event of an element, the event handler class must include methods for all the events for that element. You can leave empty those methods that do not require handler code but the empty methods must appear in the class.

You can give an event handler class any name you choose. You associate the class with a report element in BIRT Report Designer in the Properties view, as explained earlier in this chapter. The Java event handler class can either extend an adapter class or implement an event handler interface. The following sections explain adapter classes and handler interfaces.

Using event handler adapter classes

BIRT provides event handler adapter classes for every scriptable report element. An event handler adapter class contains empty methods for every event handler method for the element. If your class extends an adapter class, you need to override only the methods for the events for which you want to provide handler code.

One advantage of using an adapter class instead of implementing an interface is that your class compiles even if methods are added to the interface in a future release. If the signature of an event handler method changes in a future release, however, you must change your implementation of that method to reflect the signature change. The class compiles even if you do not change the method with the changed signature, but the method with the wrong signature is never called.

Using event handler interfaces

BIRT provides event handler interfaces for every scriptable report element. If your event handler class extends an adapter class, the adapter class implements the correct interface. If your class does not extend an adapter class, then your class must implement the appropriate interface for the report element you are scripting.

There are some advantages of specifying an interface instead of extending an adapter class. Eclipse generates stubs for every method the interface specifies. The stubs show the method arguments, so you can see the argument types of the methods you must implement.

If your class extends an adapter class, there are no generated stubs for you to examine. You also have more freedom in the design of your class structure if you avoid using an adapter class.

For example, you might want two or more event handler classes to extend a single base class. Because Java does not support multiple inheritance, the event handler class cannot extend both the adapter class and the base class. However, if the event handler class implements an interface instead of extending an adapter class, there is nothing to prevent the event handler class from extending the base class.

The disadvantage of using an interface over an adapter class is that if additional methods are added to an interface in a future release, a class that implements the interface fails to compile.

About the Java event handlers for report items

You can write an event handler for any or all the events that BIRT fires for a report item. Table 10-1 describes the events BIRT fires for each report item.

Table 10-1 Report item event handler methods

Method	Description
onPrepare()	The onPrepare() method for every report element contains the following two arguments: <ul style="list-style-type: none">■ Element design interface■ Report context interface

Table 10-1 Report item event handler methods (*continued*)

Method	Description
onCreate()	The arguments to the onCreate() method depend on the particular element. Every onCreate() method contains at least the following two arguments: <ul style="list-style-type: none">■ Element instance interface■ Report context interface
onPageBreak()	The onPageBreak() method for every report element contains the following two arguments: <ul style="list-style-type: none">■ Element instance interface■ Report context interface
onRender()	The onRender() method for every report element contains the following two arguments: <ul style="list-style-type: none">■ Element instance interface■ Report context interface

Using Java event handlers for the **DataSource** element

The DataSource event handler interface has four methods that you can implement to respond to events. A Java class to handle these events must implement the **IDataSourceEventHandler** interface or extend the **DataSourceAdapter** class.

All the event methods receive an **IReportContext** object. All the methods except the **afterClose()** method also receive an **IDataSourceInstance** object. These interfaces are discussed later in this chapter. Table 10-2 lists the methods that you can implement for a **DataSource** element.

Table 10-2 **DataSource** event handler methods

Method	Description
beforeOpen(IDataSourceInstance dataSource, IReportContext reportContext)	The beforeOpen event fires immediately before opening the data source. This handler is often used to change the connection properties, such as user name and password.
afterOpen(IDataSourceInstance dataSource, IReportContext reportContext)	The afterOpen event fires immediately after opening the data source.
beforeClose(IDataSourceInstance dataSource, IReportContext reportContext)	The beforeClose event fires immediately before closing the data source.

(continues)

Table 10-2 *DataSource event handler methods (continued)*

Method	Description
afterClose(IReportContext reportContext)	The afterClose event fires immediately after closing the data source.

Using Java event handlers for the **DataSet** element

BIRT fires five events for the **DataSet** element. A Java class to handle these events must implement the **IDataSetEventHandler** interface or extend the **DataSetAdapter** class. All **DataSet** event handler methods receive an **IReportContext** object. Additionally, all **DataSet** event handler methods except the **afterClose()** method receive an **IDataSetInstance** object. The **onFetch()** method receives a third object, an **IDataSetRow** object. These interfaces are described later in this chapter. Table 10-3 lists the methods that you can implement for a **DataSet** element.

Table 10-3 *DataSet event handler methods*

Method	Description
beforeOpen(IDatasetInstance dataSet, IReportContext reportContext)	The beforeOpen event fires immediately before opening the data set. This event handler is often used to change the query text for a data set.
afterOpen(IDatasetInstance dataSet, IReportContext reportContext)	The afterOpen event fires immediately after opening the data set.
onFetch(IDatasetInstance dataSet, IDatasetRow row, IReportContext reportContext)	The onFetch event fires upon fetching each row from the data source.
beforeClose(IDatasetInstance dataSet, IReportContext reportContext)	The beforeClose event fires immediately before closing the data set.
afterClose(IReportContext reportContext)	The afterClose event fires immediately after closing the data set.

Using Java event handlers for the **ScriptedDataSource** element

The **ScriptedDataSource** interface extends the **IDataSourceEventHandler** interface, which has four methods. The **ScriptedDataSource** interface adds two new methods to the four methods of the **IDataSourceEventHandler** interface. A Java class that provides the **ScriptedDataSource** event handlers must implement **IScriptedDataSourceEventHandler** interface or extend the **ScriptedDataSourceAdapter** class. A Java class that provides the

ScriptedDataSource event handlers must implement the two methods of the IScriptedDataSourceEventHandler interface plus the four methods of the IDataSourceEventHandler interface, which it extends.

Both of the two event handler methods of IScriptedDataSourceEventHandler receive an IDataSourceInstance object. Table 10-4 lists the two additional methods that you must implement for a ScriptedDataSource element.

Table 10-4 ScriptedDataSource event handler methods

Method	Description
open(IDataSourceInstance dataSource)	Use this method to open the data source.
close(IDataSourceInstance dataSource)	Use this method to close the data source and perform cleanup tasks.

Using Java event handlers for the ScriptedDataSet element

The ScriptedDataSet interface extends the IDataSetEventHandler interface, which has four methods. The ScriptedDataSet interface adds four new methods to the four of the IDataSourceEventHandler interface. Of the four new methods, three must be fully implemented and the fourth may be empty. A Java class that provides the ScriptedDataSet event handlers must implement IScriptedDataSetEventHandler interface or extend the ScriptedDataSetAdapter class. A Java class that provides the ScriptedDataSet event handlers must implement the four methods of the IScriptedDataSetEventHandler interface plus the four methods of the IDataSourceEventHandler interface, which it extends.

Table 10-5 lists the four additional methods that you must implement for a ScriptedDataSet element.

Table 10-5 ScriptedDataSet event handler methods

Method	Description
open(IDataSetInstance dataSet)	Called when the data set is opened. Use this method to initialize variables and to prepare for fetching rows.
fetch(IDataSetInstance dataSet, IUpdatableDataSetRow dataSetRow)	Called at row processing time. Use this method to fetch data with which to populate the row object. This method must return true if the fetch is successful and false if it is not.

(continues)

Table 10-5 ScriptedDataSet event handler methods (*continued*)

Method	Description
close(IDatasetInstance dataSet)	Called upon completion of processing a data set. Use this method to perform cleanup operations.

Using Java event handlers for the ReportDesign

BIRT fires several events that the ReportDesign element handles. A Java class to handle these events must implement the IReportEventHandler interface or extend the ReportEventAdapter class. All of the event handler methods receive an IReportContext object. The beforeFactory() method also receives an IReportDesign object. Table 10-6 lists the methods that you can implement for a ReportDesign element in the order in which they run.

Table 10-6 ReportDesign event handler methods

Method	Description
initialize(IReportContext reportContext)	The initialize event fires twice, once before the generation phase begins and once before the render phase begins.
beforeFactory(IReportDesign report, IReportContext reportContext)	The beforeFactory event fires before the generation phase begins.
afterFactory(IReportContext reportContext)	The afterFactory event fires after the generation phase ends.
beforeRender(IReportContext reportContext)	The beforeRender event fires before the presentation phase begins.
afterRender(IReportContext reportContext)	The afterRender event fires after the presentation phase ends.

Understanding the BIRT interfaces

A developer of Java event handlers needs to be familiar with several Java interfaces. Most of the handler method parameters and return values are Java interfaces rather than classes.

The most important Java interfaces for developing Java event handlers are:

- element design
- IReportElement

- element instance
- report context
- IColumnMetaData
- IDatasetInstance
- IDatasourceInstance
- IDatasetRow
- IRowData

About the element design interfaces

Every element has a unique element design interface. The element design is a Java interface that specifies methods for accessing and setting specific features of the element design. Every element design interface inherits methods from `IReportElement`.

About the methods for each report element

Besides the methods defined in `IDesignElement`, each report element has methods that are only relevant for that report element. For example, `ICell`, the design interface for a Cell object, includes the following methods in addition to those defined in `IDesignElement`:

- `getColumn()`
- `getColumnSpan()`
- `getDrop()`
- `getHeight()`
- `getRowSpan()`
- `getWidth()`
- `setColumn(int column)`
- `setColumnSpan(int span)`
- `setDrop(java.lang.String drop)`

In contrast, the methods for `ITextItem`, the design interface for a `TextItem` element, includes these additional methods:

- `getContent()`
- `getContentKey()`
- `getContentType()`
- `getDisplayContent()`
- `setContent(java.lang.String value)`

- `setContentKey(java.lang.String resourceKey)`
- `setContentType(java.lang.String contentType).`

About the **IReportElement** interface

The **IReportElement** interface is the base interface for all the report element interfaces. **IReportElement** has the following methods:

- `getComments()`
- `getCustomXml()`
- `getDisplayName()`
- `getDisplayNameKey()`
- `getName()`
- `getNamedExpression(java.lang.String name)`
- `getParent()`
- `getQualifiedName()`
- `getStyle()`
- `getUserProperty(java.lang.String name)`
- `setComments(java.lang.String theComments)`
- `setCustomXml(java.lang.String customXml)`
- `setDisplayName(java.lang.String displayName)`
- `setDisplayNameKey(java.lang.String displayNameKey)`
- `setName(java.lang.String name)`
- `setNamedExpression(java.lang.String name, java.lang.String exp)`
- `setUserProperty(java.lang.String name, java.lang.Object value)`

About the element instance interfaces

The element instance interfaces are available at run time, but not at design time. They contain the run-time instance of the element. The element instance interface passes to both `onCreate()`, the generation phase event handler, and to `onRender()`, the presentation phase event handler.

Through instance interfaces, you have access to a different set of properties than you do at design time. There is no superinterface from which all element instance interfaces inherit. Like the element design interface, the set of methods in the instance interfaces vary from element to element.

For example, **ICellInstance**, the Cell instance interface, contains the following methods:

- `getColSpan()`

- getColumn()
- getRowSpan()
- setColSpan(int colSpan)
- setRowSpan(int rowSpan)

By comparison, IRowInstance, the Row instance interface, contains these methods:

- getBookmarkValue()
- getHeight()
- getStyle()
- setBookmark()
- setHeight()

Using the IReportContext interface

An object of type IReportContext passes to all event handlers except those for ScriptedDataSource and ScriptedDataSet objects. The IReportContext interface includes the methods shown in Table 10-7.

Table 10-7 IReportContext interface methods

Method	Task
deleteGlobalVariable(java.lang.String name)	Removes a global variable created using the setGlobalVariable() method.
deletePersistentGlobalVariable(java.lang.String name)	Removes a persistent global variable created using the setPersistentGlobalVariable() method.
getAppContext()	Retrieves the application context object as a java.util.Map object. The report application can use the application context object to pass any information that is application-specific.
getGlobalVariable(java.lang.String name)	Returns the object saved with the setGlobalVariable() method. The string argument is the key used when saving the object.
getHttpServletRequest()	Returns the HttpServletRequest object associated with the URL requesting the report. The HttpServletRequest object provides access to the request URL and any parameters appended to the request. It also provides access to the HTTP session object.

(continues)

Table 10-7 IReportContext interface methods (*continued*)

Method	Task
getLocale()	Returns the locale associated with the report execution or rendering task. This locale can be different from the local machine system or user locale.
getMessage(java.lang.String key)	Returns a message from the default properties file.
getMessage(java.lang.String key, java.util.Locale locale, java.lang.Object [] params)	Returns a message from the properties file for a specified locale, using a parameters array.
getMessage(java.lang.String key, java.lang.Object[] params)	Returns a message from the default properties file, using a parameters array.
getOutputFormat()	Returns a string containing either html, pdf, postscript, ppt, xls, or doc, depending on the format specified in the __format parameter of the request URL.
getParameterValue(java.lang.String name)	Returns the value of the parameter named in the name argument. The value returned is a java.lang.Object.
getPersistentGlobalVariable(java.lang.String name)	Returns the serializable object saved with the setPersistentGlobalVariable() method. The string argument is the key used when saving the serializable object.
getRenderOption()	Gets the render options used to render the report.
getReportRunnable()	Gets the report runnable used to create or render this report.
getTaskType()	Gets the type of the current task.
setGlobalVariable(java.lang.String name, java.lang.Object obj)	Saves an object that can be retrieved in the same execution phase as it is saved. The setGlobalVariable() method takes a string argument and an Object argument. You use the string argument as a key with which to later retrieve the saved object.
setParameterDisplayText(java.lang.String name, java.lang.Object value)	Sets the display text for a parameter.
setParameterValue(java.lang.String name, java.lang.Object value)	Sets the value of a named parameter with the value contained in the value parameter.

Table 10-7 IReportContext interface methods (*continued*)

Method	Task
setPersistentGlobalVariable(java.lang.String name, java.io.Serializable obj)	Saves an object that can be retrieved in a different execution phase than it is saved. The setPersistentGlobalVariable() method takes a string argument and a serializable object argument. You use the string argument as a key with which to later retrieve the serializable object. The object is serializable because it must be persisted between phases to support executing the two phases at different times and possibly on different machines. The serializable object is saved in the report document.

Using the IColumnMetaData interface

The IColumnMetaData interface provides information about the columns of the data set. Table 10-8 lists the methods in the IColumnMetaData interface class.

Table 10-8 IColumnMetaData interface methods

Method	Returns
getColumnAlias(int index)	Alias assigned to the column at the position indicated by the index argument
getColumnCount()	Count of columns in the data set
getColumnLabel(int index)	Label assigned to the column at the position indicated by the index argument
getColumnName(int index)	String containing the name of the column at the position indicated by the index argument
getColumnNativeTypeName(int index)	Name of the type of data in the column at the position indicated by the index argument
getColumnType(int index)	Data type of the column at the position indicated by the index argument
getColumnTypeName(int index)	Name of the type of data in the column at the position indicated by the index argument
isComputedColumn(int index)	True or false, depending on whether the column at the position indicated by the index argument is a computed field

Using the **IDataSetInstance** interface

The **IDataSetInstance** interface provides access to many aspects of the data set and associated elements. An **IDataSetInstance** object passes to every **DataSet** event handler method.

Table 10-9 describes the methods in the interface **IDataSetInstance**.

Table 10-9 **IDataSetInstance** interface methods

Method	Returns
<code>getAllExtensionProperties()</code>	The data set extension properties in the form of a <code>java.util.Map</code> object. The map object maps data extension names to their values.
<code>getColumnMetaData()</code>	An <code>IColumnMetaData</code> object that provides the data set's metadata.
<code>getDataSource()</code>	A <code>DataSource</code> object associated with the data set.
<code>getExtensionID()</code>	The unique ID that identifies the type of the data set, assigned by the extension that implements this data set.
<code>getExtensionProperty(java.lang.String name)</code>	The value of a data set extension property.
<code>getName()</code>	The name of this data set.
<code>getQueryText()</code>	The query text of the data set.
<code>setExtensionProperty(java.lang.String name, java.lang.String value)</code>	The value of an extension property.
<code>setQueryText(java.lang.String queryText)</code>	The query text of the data set.

Using the **IDataSetRow** interface

An object of the **IDataSetRow** type passes to the **DataSet.onfetch()** event handler method. Table 10-10 lists the methods in the **IDataSetRow** interface. Note that there are two `getColumnValue()` methods. The two methods differ only in the argument that specifies the column containing the value. They both return a `java.lang.Object` object, which you must cast to the appropriate type for the column.

Table 10-10 **IDataSetRow** interface methods

Method	Returns
<code>getColumnValue(int index)</code>	The column data by index. This index is 1-based.

Table 10-10 **IDataSetRow interface methods (continued)**

Method	Returns
getColumnValue(java.lang.String name)	The column data by column name.
getDataSet()	An IDDataSetInstance object representing the data set that contains this row.

Using the IRowData interface

An object of the IRowData type returns from the getRowData() method of IReportElementInstance, which every report element instance interface extends.

IRowData provides access to the bound values that appear in the table. The IRowData interface has two getExpressionValue() methods. Both methods return the display value for a specific column in the table. The two methods differ in the argument you pass to specify the column that you require.

Table 10-11 lists the methods in the IRowData interface.

Table 10-11 **IRowData interface methods**

Method	Returns
getColumnCount()	Return the count of the bounding expressions.
getColumnName(int index)	Return the name of the bounding expression by id.
getColumnValue(int index)	Return the value of the bounding expression by id. This index is 1-based.
getColumnValue(String name)	Return the value of the bounding expression by name.

Java event handler example

Listed below are some common examples that illustrate event handlers written in Java. The examples illustrated in Chapter 9, “Using JavaScript to Write an Event Handler,” can be used as reference as well.

Report level events

Report level events include initialize, beforeFactory, afterFactory, beforeRender, and afterRender. When these events are called depends on the type of report execution occurring.

The beforeFactory event is an often overridden event, because changes to the report design can occur in this event. The following example checks a boolean parameter. If this value is true, a table named Mytable drops from the report design.

```
package my.event.handlers;

import org.eclipse.birt.report.engine.api.script.IReportContext;
import org.eclipse.birt.report.engine.api.script.element
    .IReportDesign;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .ReportEventAdapter;
import org.eclipse.birt.report.model.api.*;
import org.eclipse.birt.report.model.api.activity
    .SemanticException;

public class MyReportEvents extends ReportEventAdapter {
    @Override
    public void beforeFactory(IReportDesign report,
        IReportContext reportContext) {
        if((Boolean)reportContext.getParameterValue(
            "DropTable" ) ){
            ReportDesignHandle rdh =
                ( ReportDesignHandle )reportContext
                    .getReportRunnable( ).getDesignHandle( );
            try{
                rdh.findElement( "Mytable" ).drop( );
            }catch( SemanticException e ){
                e.printStackTrace( );
            }
        }
    }
}
```

In this example, we use the Design Engine API, so we have to add modelapi.jar and coreapi.jar to the buildpath and classpath. The following example also makes use of the Design Engine API to add a data source, data set, and table to a report using the beforeFactory event.

```
package my.event.handlers;

import org.eclipse.birt.report.engine.api.script.IReportContext;
import org.eclipse.birt.report.engine.api.script.element
    .IReportDesign;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .ReportEventAdapter;
import org.eclipse.birt.report.model.api.ReportDesignHandle;
import org.eclipse.birt.report.model.api.LibraryHandle;
import org.eclipse.birt.report.model.api.DesignElementHandle;
import org.eclipse.birt.report.model.core.DesignSession;
```

```

public class MyReportAddTableEvent extends ReportEventAdapter {
    @Override
    public void beforeFactory( IReportDesign report,
        IReportContext reportContext ) {
        ReportDesignHandle rdh =
            ( ReportDesignHandle )reportContext
                .getReportRunnable( )
                .getDesignHandle( );
        DesignSession ds =rdh.getModule( ).getSession( );
        try{
            String rsf = ds.getResourceFolder( );
            LibraryHandle libhan = ds.openLibrary(
                rsf + "/mylibrary.rptlibrary" ).handle( );
            DesignElementHandle deh1 =
                libhan.findDataSource( "mydatasource" );
            DesignElementHandle deh2 =
                libhan.findDataSet( "mydataset" );
            DesignElementHandle deh3 =
                libhan.findElement( "mytable" );
            rdh.getDataSources( ).add( deh1 );
            rdh.getDataSets( ).add( deh2 );
            rdh.getBody( ).add( deh3 );
            libhan.close( );
        }catch(Exception e){
            e.printStackTrace( );
        }
    }
}

```

In the example, the library mylibrary.rptlibrary, located in the resource folder, opens. The data source named mydatasource, the data set named mydataset, and the table named mytable are all added to the current report design.

Report item events

Report item events allow the developer to change the default behavior of item. Changes made in the onPrepare event can change the design of the item, changes made in the onCreate event can change the particular instance of a item at generation time, and the onRender event can change properties of an instance of the item at render time. Consider the following image item example:

```

package my.event.handlers;

import org.eclipse.birt.report.engine.api.script
    .IReportContext;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .ImageEventAdapter;
import org.eclipse.birt.report.engine.api.script.instance
    .IImageInstance;

```

```

import org.eclipse.birt.report.engine.content
    .IImageContent;
import java.io.*;

public class myImageHandler extends ImageEventAdapter {

    public void onRender( IImageInstance image, IReportContext
        reportContext ) {
        if( image.getImageSource( ) == IImageContent.IMAGE_URL ){
            if( reportContext.getOutputFormat( )
                .equalsIgnoreCase( "html" ) ){
                image.setURL(
                    "http://us.i1.yimg.com/us.yimg.com/i/ww/
                     beta/y3.gif" );
            }else{
                image.setURL( "http://www.google.com/intl/en_ALL/
                     images/logo.gif" );
            }
        }
        if( image.getImageSource( ) ==
            IImageContent.IMAGE_FILE ){
            String rpl = image.getFile( );
            if( rpl.contains( "up" ) ){
                String newstr = rpl.replaceAll( "up", "down" );
                image.setFile( newstr );
            }
        }
        if( image.getImageSource( ) ==
            IImageContent.IMAGE_NAME ){
            if( ( Boolean )reportContext.getParameterValue(
                "SwapImage" ) ){
                if( image.getImageName( ).compareToIgnoreCase(
                    "tocico.png" ) == 0 ){
                    image.setImageName( "clientprintico.PNG" );
                }
            }
        }
        if( image.getImageSource( ) ==
            IImageContent.IMAGE_EXPRESSION){
            try{
                File myfile = new File( "c:/temp/test.png" );
                FileInputStream ist = new FileInputStream( myfile );
                long lengthi = myfile.length( );
                byte[ ] imageData = new byte[ ( int )lengthi ];
                ist.read( imageData );
                ist.close( );
                image.setData( imageData );
            }catch( Exception e ){
                e.printStackTrace( );
            }
        }
    }
}

```

```
}
```

```
}
```

This example illustrates changing image sources for different types of image items. If the image type is a URL image, the output format is checked and the URL for the image changes. If the image type is a file from the resource folder, the filename is searched for the string up. If this string is found the image is replaced with an image with the name down. If the image type is an embedded image, the report parameter, SwapImage, is checked. If the value is true, the image is swapped to another embedded image. If the image is a BLOB type image from a database, the bytes for the image are swapped to the bytes read from the local file system.

As stated earlier, onPrepare event handlers can effect the design of a particular report item. In this example, an onPrepare event handler adds a hyperlink and a table of contents entry to a data element design. The onCreate event is overridden to modify the hyperlink based on the value of the data item instance.

```
package my.event.handlers;

import org.eclipse.birt.report.engine.api.script
    .IReportContext;
import org.eclipse.birt.report.engine.api.script.element
    .IAction;
import org.eclipse.birt.report.engine.api.script.element
    .IDataItem;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .DataItemEventAdapter;
import org.eclipse.birt.report.engine.api.script.instance
    .IActionInstance;
import org.eclipse.birt.report.engine.api.script.instance
    .IDataItemInstance;
import org.eclipse.birt.report.model.api.elements
    .DesignChoiceConstants;

public class MyDataElementEvent extends DataItemEventAdapter {

    @Override
    public void onCreate( IDataItemInstance data,
        IReportContext reportContext ) {
        IActionInstance ai = data.getAction( );
        if( ( Integer )data.getValue( ) == 10101 ){
            ai.setHyperlink( "http://www.yahoo.com","_blank" );
        }
    }

    @Override
    public void onPrepare(IDataItem dataItemHandle,
        IReportContext reportContext) {
        IAction act = dataItemHandle.getAction( );
```

```

        try{
            act.setTargetWindow( "_blank" );
            act.setURI( "http://www.google.com" );
            act.setLinkType( DesignChoiceConstants
                .ACTION_LINK_TYPE_HYPERLINK );
            dataItemHandle.setTocExpression(
                "row[ \"ORDERNUMBER\" ]");
        }catch( Exception e ){
            e.printStackTrace( );
        }
    }
}

```

Using an onCreate event handler for a row allows access to the table columns defined in the binding tab. As example, the following code retrieves the QUANTITYORDERED column for each row of data in a table element. If the value is greater than forty, the background for the row is set to green.

```

package my.event.handlers;

import org.eclipse.birt.report.engine.api.script
    .IReportContext;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .RowEventAdapter;
import org.eclipse.birt.report.engine.api.script.instance
    .IRowInstance;

public class MyRowEvents extends RowEventAdapter {
    @Override
    public void onCreate( IRowInstance rowInstance,
        IReportContext reportContext ) {
        try{
            Integer qty =
                ( Integer )rowInstance.getRowData( ).getColumnValue(
                    "QUANTITYORDERED" );
            if( qty > 40 ){
                rowInstance.getStyle( ).setBackgroundColor( "green" );
            }
        }catch( Exception e ){
            e.printStackTrace( );
        }
    }
}

```

Event handlers can share data by using the reportContext setPersistentGlobal variable method. This method writes the variable to the report document if generating a report with two processes. Suppose you have an order listing report and you want to have the last order number presented in the page header. You can perform this operation when using two processes to generate and render the report with the following solution.

Add a dynamic text element to the master page header with the following expression:

```
"placeholder"+pageNumber;
```

The generated report produces placeholder1, placeholder2, and so forth.

Next create an event handler for the onPageBreak event for the data item that appears in the header, as shown in the following code:

```
package my.event.handlers;

import org.eclipse.birt.report.engine.api.script
    .IReportContext;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .DataItemEventAdapter;
import org.eclipse.birt.report.engine.api.script.instance
    .IDataItemInstance;
import java.util.*;

public class MyCustomHeaderDataItem extends
    DataItemEventAdapter {
    @Override
    public void onPageBreak( IDataItemInstance data,
        IReportContext reportContext) {
        ArrayList ar =
            (ArrayList)reportContext.getPersistentGlobalVariable(
                "MyArrayList" );
        if( ar == null ){
            ar = new ArrayList( );
        }
        ar.add( "Page Ends with: " + data.getValue( ) );
        reportContext.setPersistentGlobalVariable(
            "MyArrayList", ar);
    }
}
```

This code adds an array list item for each page. When the onPageBreak event fires, the data item contains the last value for the page. The array list is saved to the report document using the setPersistentGlobalVariable method.

Finally, create an onRender event handler for the dynamic text element added to the master page header with the following code:

```
package my.event.handlers;

import
    org.eclipse.birt.report.engine.api.script.IReportContext;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .DynamicTextEventAdapter;
import org.eclipse.birt.report.engine.api.script.instance
    .IDynamicTextInstance;
import java.util.*;
```

```

public class MyCustomHeaderDynamicTextItem extends
    DynamicTextEventAdapter {
    @Override
    public void onRender( IDynamicTextInstance text,
        IReportContext reportContext) {
        String cmp = "nomatch";
        if( text.getText( ).length( ) > 10 ){
            cmp = text.getText( ).substring( 0,10 );
        }
        if( cmp.compareToIgnoreCase("placeholder" ) == 0 ){
            ArrayList ar =
                ( ArrayList )reportContext
                    .getPersistentGlobalVariable( "MyArrayList" );
            if( ar == null ){
                return;
            }
            Integer ccount =
                Integer.parseInt( text.getText( ).substring( 11 ) )-1;
            text.setText( ( String )ar.get( ccount ) );
        }
    }
}

```

This code first verifies that you are working with the proper dynamic text element by looking for the placeholder text in the value of the dynamic text item. The array list produced by the data item is retrieved from the report document using the getPersistentGlobalVariable method. The specific page string is retrieved from the array list by getting the page number from the current value of the dynamic text item.

It is important to realize that this method only works when generation and presentation occur in two separate processes. For example, when you use the frameset mapping in the BIRT Web Viewer or a run task and a render task with the Report Engine API. The reason is that the onRender event for the dynamic text item in the master page fires after the onCreate and onPageBreak events for all report items when using two processes. When using one process, the onRender event fires immediately after the onCreate event for the dynamic text item in the master page header.

Debugging a Java event handler

One of the main advantages of writing an event handler in Java is the ability to debug the code using Eclipse. You can debug by opening the event handler in the Java Perspective, setting appropriate break points, and selecting Run->Open Debug Dialog. BIRT supplies a BIRT Report launch configuration.

Select BIRT Report from the available configurations and choose Launch Configuration. Select the projects that contain reports using your event

handler from the list of available projects and choose Debug to launch a separate instance of Eclipse.

In the new instance, you can navigate to a report that contains a reference to the event handler and choose Preview. Any breakpoints in the event handler fire.

This page intentionally left blank

11

Working with Chart Event Handlers

BIRT supports an event handler model for the Chart Engine. The model is similar to the model for standard BIRT report elements and supports both the Java and JavaScript environments. This chapter provides details on both averments. The Chart Engine also supports this event model when used outside of BIRT.

Chart events overview

A chart is a graphical representation of data. A chart event occurs before, during, or after the drawing of a chart. Table 11-1 lists the chart event handler methods and describes when these methods are called.

Table 11-1 Chart event handler methods

Method	Called
afterDataSetFilled(Series series, DataSet dataSet, IChartScriptContext icsc)	After populating the series data set
afterDrawAxisLabel(Axis axis, Label label, IChartScriptContext icsc)	After rendering each label on a given axis
afterDrawAxisTitle(Axis axis, Label label, IChartScriptContext icsc)	After rendering the title of an axis
afterDrawBlock(Block block, IChartScriptContext icsc)	After drawing each block
afterDrawDataPoint(DataPointHints dph, Fill fill, IChartScriptContext icsc)	After drawing each data point graphical representation or marker

(continues)

Table 11-1 Chart event handler methods (*continued*)

Method	Called
afterDrawDataPointLabel(DataPointHints dph, Label label, IChartScriptContext icsc)	After rendering the label for each data point
afterDrawFittingCurve(CurveFitting cf, IChartScriptContext icsc)	After rendering curve fitting
afterDrawLegendItem(LegendEntryRenderingHints lerh, Bounds bounds, IChartScriptContext icsc)	After drawing each entry in the legend
afterDrawMarkerLine(Axis axis, MarkerLine mLine, IChartScriptContext icsc)	After drawing each marker line in an axis
afterDrawMarkerRange(Axis axis, MarkerRange mRange, IChartScriptContext icsc)	After drawing each marker range in an axis
afterDrawSeries(Series series, ISeriesRenderer isr, IChartScriptContext icsc)	After rendering the series
afterDrawSeriesTitle(Series series, Label label, IChartScriptContext icsc)	After rendering the title of a series
afterGeneration(GeneratedChartState gcs, IChartScriptContext icsc)	After generation of a chart model to GeneratedChartState
afterRendering(GeneratedChartState gcs, IChartScriptContext icsc)	After the chart renders
beforeDataSetFilled(Series series, IDatasetProcessor idsp, IChartScriptContext icsc)	Before populating the series data set using the DataSetProcessor
beforeDrawAxisLabel(Axis axis, Label label, IChartScriptContext icsc)	Before rendering each label on a given axis
beforeDrawAxisTitle(Axis axis, Label label, IChartScriptContext icsc)	Before rendering the title of an axis
beforeDrawBlock(Block block, IChartScriptContext icsc)	Before drawing each block
beforeDrawDataPoint(DataPointHints dph, Fill fill, IChartScriptContext icsc)	Before drawing each datapoint graphical representation or marker
beforeDrawDataPointLabel(DataPointHints dph, Label label, IChartScriptContext icsc)	Before rendering the label for each datapoint
beforeDrawFittingCurve(CurveFitting cf, IChartScriptContext icsc)	Before rendering curve fitting
beforeDrawLegendItem(LegendEntryRenderingHints lerh, Bounds bounds, IChartScriptContext icsc)	Before drawing each entry in the legend
beforeDrawMarkerLine(Axis axis, MarkerLine mLine, IChartScriptContext icsc)	Before drawing each marker line in an axis

Table 11-1 Chart event handler methods (*continued*)

Method	Called
beforeDrawMarkerRange(Axis axis, MarkerRange mRange, IChartScriptContext icsc)	Before drawing each marker range in an axis
beforeDrawSeries(Series series, ISeriesRenderer isr, IChartScriptContext icsc)	Before rendering the series
beforeDrawSeriesTitle(Series series, Label label, IChartScriptContext icsc)	Before rendering the title of a series
beforeGeneration(Chart cm, IChartScriptContext icsc)	Before generating a chart model to GeneratedChartState
beforeRendering(GeneratedChartState gcs, IChartScriptContext icsc)	Before the chart renders

The following sections describe how to implement the chart events based on the event handler model for the BIRT Chart Engine.

Understanding when chart events trigger

The Chart Engine generator processes charts in four phases in the BIRT reporting environment:

- Prepare
- Data binding
- Build
- Render

These phases correspond to the following methods called when the BIRT Chart Engine generates a chart:

- Generator.prepare()
- Generator.bindData()
- Generator.build()
- Generator.render()

The following sections describes these phases and provides more information about the context in which chart events trigger.

Prepare phase

The prepare phase sets up the chart script context. This phase sets the class loader for the scripting environment to the same classloader that the BIRT scripting environment uses. This phase does not trigger any event handlers.

Data binding phase

The data binding phase prepares all data sets associated with a series. The Chart Engine supports both static and dynamic data. This phase is required when using dynamic data, which is the case when rendering a chart in a BIRT report.

Static data

The Chart Engine requires placing all data in an object that implements the chart org.eclipse.birt.chart.model.data.DataSet interface before generating the chart. The following interfaces extend the DataSet interface:

- BubbleDataSet
- DateTimeDataSet
- DifferenceDataSet
- GanttDataSet
- NumberDataSet
- StockDataSet
- TextDataSet

Each interface has a corresponding implementation class that supplies a static create method for initializing a data structure. For example, when using the Chart Engine API, you can create a NumberDataSet using the following code:

```
NumberDataSet seriesOneValues =  
    NumberDataSetImpl.create( new double[ ]{ 15, 23, 55, 76 } );
```

This code creates a static DataSet for a series in a chart. The Chart Engine also supports extension points to extend this list of data sets. When using static data sets, the Chart Engine does not support sorting or grouping. When using static data, you do not need a data binding phase since the data is already prepared in the necessary format.

Dynamic data

The Chart Engine supports sourcing data from a dynamic source, which includes BIRT data sets and java.sqlResultSet objects. In these cases, the binding phase generates the chart data sets based on the expressions defined in the model for each series in the chart. After completing this phase, each Chart series binds to one of the chart data sets listed in the static data section.

A series is a set of plotted values in a chart. BIRT uses a series definition object to define what data a series contains. For example, row["month"] populates a series with the month column from a BIRT data set bound to a chart. This series definition object results in a run-time series that is bound to a supported chart data set type when the chart generates.

BIRT also supports defining groups and sorting in a series definition. When using these features, BIRT creates additional run time series automatically, each with its own chart data set.

For example, if a BIRT report contains a data set with three columns, such as the product name, amount sold, and month sold, you can create a bar chart to display the data. The category series contains the product and the value series contains the amount.

Using the category series as the x-axis and the value series as the y-axis, the chart model produces two run-time series when generating the chart. You can add the month to the y-axis series to group the data and produce multiple run-time series. The category series and the value series can contain up to twelve run-time series, one for each month.

All run-time series for a chart must have the same number of data points. To get the run-time series for a specific series definition, call the `getRunTimeSeries()` method for the specific series definition. Listing 11-1 gets a run-time series and sets the first bar series to have a riser type of triangle.

Listing 11-1 Getting a run-time series

```
function beforeGeneration( chart, icsc )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute );

    var xAxis = chart.getBaseAxes( )[0];
    var yAxis = chart.getOrthogonalAxes( xAxis, true )[0]
    var seriesDef = yAxis.getSeriesDefinitions( ).get( 0 )
    var runSeries = seriesDef.getRunTimeSeries( );
    var firstRunSeries = runSeries.get( 0 );
    firstRunSeries.setRiser( RiserType.TRIANGLE_LITERAL );
}
```

Some chart types use nested `SeriesDefinitions` objects. For example, a pie chart creates a top-level series definition that stores category series information. This top-level series definition also contains another nested series definition that stores the information for the value series.

Binding phase script events

The binding phase triggers the following event types:

- `beforeDataSetFilled`
- `afterDataSetFilled`

The `beforeDataSetFilled` event receives two parameters:

- `IDataSetProcessor` implementation
- Chart script context

The `IDataSetProcessor` implementation must contain a `populate` method that creates one of the static chart data sets. This `populate` method is called just after this event triggers.

The default implementation of the `IDataSetProcessor` is done with the `DataSetProcessorImpl` class, which currently provides no useful methods for this event. You can extend this interface using the chart extension points to add your own data set processor to add calls to the `beforeDataSet` event.

The Chart Engine calls the `afterDataSetFilled` event after creating and populating the run-time series chart data set. This event receives three parameters, the Chart script context, the specific run-time series, and the populated chart data set.

You can manipulate the chart data set at this point. For example, Listing 11-2 replace null values in the data set.

Listing 11-2 Replacing null values in a data set

```
function afterDataSetFilled( series, dataSet, icsc )
{
    var list = dataSet.getValues();
    for ( i=0; i<list.length; i=i+1 )
    {
        if ( list[i] == null )
        {
            list[i]= 0;
        }
    }
}
```

In the example, the chart data set passed to the method is the chart data set object for the selected series. If `series` is numeric, the object is an instance of `NumberDataSetImpl`.

The `Series` object is an implementation of the specific run-time series type being created. This object is a `SeriesImpl` object for the category series or a chart-specific type for other series. When rendering a bar chart, it is a `BarSeriesImpl` object.

If you are implementing a JavaScript event handler, you do not need to cast to a specific type. If you are using a Java event handler, you may need to cast to a specific type if the method you want to call in the `series` object is not defined by the `Series` interface.

This event triggers for each run-time series. To identify which series you are using, use the `series` identifier, as shown in the following code.

```
if( series.getSeriesIdentifier( ) == "series one" ){
    ..
}
```

By default, this value is undefined. You can set it in the Chart Builder Wizard under series title. If you create more than one run-time series from a series definition, this identifier is the same on all these generated run-time series.

You can also check the class type to determine the type of series you are using, as shown in Listing 11-3.

Listing 11-3 Checking the class type to determine the type of series

```
function afterDataSetFilled( series, dataSet, icsc )  
{  
    importPackage( Packages.java.io );  
    importPackage(  
        Packages.org.eclipse.birt.chart.model.type.impl );  
    if( series.getClass( ) == LineSeriesImpl ){  
        series.getLineAttributes( ).setThickness( 5 );  
    }  
    if( series.getClass( ) == BarSeriesImpl ){  
        ...  
    }  
}
```

For more information on the chart script context parameter, see “Chart script context,” later in this chapter.

Building phase

The chart building phase starts when a chart model binds to a data set to produce a GeneratedChartState object. This object contains most of the information required to render the chart.

Do not confuse the chart building phase with the BIRT report generation phase. In BIRT, the report generation phase runs in the rendering phase.

A chart consists of blocks, which are the rectangular sub-regions of the chart that act as containers for specific chart information. The chart model contains the following blocks:

- Outermost block that contains all other blocks
- Title block for the chart title
- Plot block to render the series and optional axis
- Legend block to display the legend

The building phase calculates the following items:

- Bounds for all blocks
- Optional axis with minimum, maximum, and scale values

This phase also performs the following additional tasks:

- Initializes all series renderers

- Creates the rendering hints for the legend, each run-time series, and the data points in each series
- Stores all calculations in the GeneratedChartState object in the chart model

The building phase triggers the following script events:

- beforeGeneration
- afterGeneration

The beforeGeneration event receives the chart model bound to data from the binding phase and the Chart script context. In the Chart Engine, all charts are subclasses or an instance of ChartWithAxesImpl or ChartWithoutAxesImpl.

This object passes to the beforeGeneration event. If modifications are required and you are writing your event in Java, it may be necessary to cast to the specific implementation of the chart interface. You can also check the type of chart when writing script. For example, assume you are writing a beforeGeneration event handler for a pie chart and you want to explode slices that contain large values. You can use script similar to the example in Listing 11-4.

Listing 11-4 Writing a beforeGeneration event handler

```
function beforeGeneration( chart, icsc )
{
    importPackage( Packages.org.eclipse.birt.chart.model.impl );
    importPackage(
        Packages.org.eclipse.birt.chart.model.type.impl );
    if( chart.getClass( ) == ChartWithAxesImpl ){
        //Do something if a line chart...
    }
    if( chart.getClass( ) == ChartWithoutAxesImpl ){
        seriesDef = chart.getSeriesDefinitions( ).get( 0 );
        catRunSeries = seriesDef.getRunTimeSeries( );
        //Pie Charts use nested series definitions
        //for the value series
        valSeriesDef =
            seriesDef.getSeriesDefinitions( ).get( 0 );
        valRunSeries = valSeriesDef.getRunTimeSeries( ).get( 0 );
        if( valRunSeries.getClass( ) == PieSeriesImpl ){
            valRunSeries.setExplosion( 10 );
            valRunSeries.setExplosionExpression(
                "valueData > 1500000" );
        }
    }
}
```

In the example, the values series with a pie chart nests under a second-level series definition. The category series nests below the top-level series definition.

After all the calculations process, the afterGeneration event occurs and the chart is ready to render. This event receives the Chart script context and the GeneratedChartState object. This event is called just prior to entering the rendering phase. Modifications to the chart can be made by using the GeneratedChartState object to retrieve the chart model. For example, Listing 11-5 adds a blue border around the plot area.

Listing 11-5 Writing an afterGeneration event handler

```
function afterGeneration( gcs, icsc )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute );
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute.impl );
    var outline = gcs.getChartModel().getPlot().getOutline();
    outline.setColor( ColorDefinitionImpl.BLUE() );
    outline.setStyle( LineStyle.SOLID_LITERAL );
    outline.setVisible( true );
}
```

Rendering phase

The rendering phase takes the GeneratedChartState object and renders the chart. This phase also sets up chart interactivity. The rendering phase uses the following main object types to handle rendering a chart:

- Device renderer
- Display server
- Model renderers

The device renderer is an object that implements the IDeviceRenderer interface and provides methods for drawing primitives like drawPolygon, drawRectangle, fillArc, and drawText. The charting engine has device renderers for SWT and Swing. The Swing renderer is extended to supply device renderers for BMP, JPEG, PNG, SVG, and PDF formats. The Chart Engine provides an extension point to extend this list too.

The display server provides generic services to the device render for such operations as getting the dpi resolution and providing text metrics. The display server must implement the IDisplayServer interface. The charting engine provides two implementations. One for Standard Widget Toolkit (SWT) and the other for Swing.

The display server links to a specific device renderer based on an Eclipse extension point entry for the specific device renderer. Currently all Swing-related renderers use the Swing display server and the SWT device renderer uses the SWT display server. The Chart Engine provides an extension point to extend the list of display servers.

Model renderers render a specific series and rely on the device render to actually render the graphic primitives that make up the chart series data points. All chart series renderers extend from the BaseRenderer or AxesRenderer abstract class and must implement the ISeriesRenderer Interface, which provides the following methods:

- compute() is called prior to ending the building phase to allow the renderer to do prerendering calculations
- renderLegendGraphic() is called during the render phase to allow the renderer to draw the graphic markers in the legend
- renderSeries() receives the device renderer and the series rendering hints and is responsible for drawing the specific series.

Table 11-2 lists the series-specific renderers. The Chart Engine provides an extension point to extend this list.

Table 11-2 Series-specific renderers

AxesRenderer	BaseRenderer
Bar	Dial
Bubble	Pie
Scatter	
Line	
Difference	
Gantt	
Stock	

The Rendering phase essentially loops through all the run-time series defined for a chart and renders each specific series. In Listing 11-6, the pseudo-code illustrates the order of the basic rendering operations. In the pseudo-code, the axis rendering statements only apply to chart types that contain axes.

Listing 11-6 Pseudo-code illustrating the order of the rendering operation

```
Loop for All Renders
  *Render Main Block
    *Render Title Block
    Render Plot Block
      *Render Background
      *Render Axis Structure
        Render Series (Series Specific ie Line or Bar)
        **Render Axis Labels
        **Render Legend Block (call to specific series Renderer
          for graphic)
      End Loop
      * Only on first Series
      ** Only on last Series
```

Rendering phase script events

The bulk of the chart events trigger during the rendering phase. Some available events are chart-type specific and not called for all series renderers. In Listing 11-7, the pseudo-code describes the order for most event triggers and corresponds to the rendering pseudo-code presented in the last section.

Listing 11-7 Pseudo-code describing the order for event triggers

```
beforeRendering
Loop all run-time series
If first series
    beforeDrawBlock - Main Block
    afterDrawBlock - Main Block
    beforeDrawBlock - Title Block
    afterDrawBlock - Title Block
    beforeDrawBlock - Plot Block
        Loop all marker ranges
            beforeDrawMarkerRange
            afterDrawMarkerRange
        End Loop
        Loop all marker lines
            beforeDrawMarkerLine
            afterDrawMarkerLine
        End Loop
        beforeDrawSeries - Category Series
        afterDrawSeries - Category Series
        afterDrawBlock - Plot Block
    End If
    beforeDrawBlock - Plot Block
        beforeDrawSeries - For Specific Series
            Loop all data points - For Specific Series
                beforeDrawDataPoint
                afterDrawDataPoint
            End Loop
            Loop all data point labels - For Specific Series
                beforeDrawDataPointLabel
                afterDrawDataPointLabel
            End Loop
            beforeDrawFittingCurve
            afterDrawFittingCurve
        afterDrawSeries
        If last series and chart contains axes
            Loop for each axis
                beforeDrawAxisLabel
                afterDrawAxisLabel
            End Loop
            beforeDrawAxisTitle
            afterDrawAxisTitle
        End If
```

```

afterDrawBlock - Plot Block
If last series
    beforeDrawBlock - Legend
        Loop all legend entries
            beforeDrawLegendItem
            afterDrawLegendItem
        End Loop
    afterDrawBlock - Legen
End If
End Loop
afterRender

```

The actual order can vary slightly based on the specific series renderer. This chapter does not describe every event in detail. The common objects used in each script are presented in the next section.

Rendering blocks

The before and afterDrawBlock events trigger multiple times for the Plot block. These two events receive BlockImpl objects, which can be manipulated using the various methods and properties of the block, such as setting the outline, anchor points, and the background fill. The block class provides methods for determining which block triggered the event. For example, Listing 11-8 shows how to set the anchor point for the legend in a legend block.

Listing 11-8 Setting the anchor point for the legend in a legend block

```

function beforeDrawBlock( block, scriptContext )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute.impl );
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute );
    if ( block.isLegend( ) )
    {
        block.getOutline( ).setVisible( true );
        block.getOutline( ).getColor( ).set( 21,244,231 );
        block.setBackground( ColorDefinitionImpl.YELLOW( ) );
        block.setAnchor( Anchor.NORTH_LITERAL );
    }
    else if ( block.isPlot( ) )
    {
        ...
    }
    else if ( block.isTitle( ) )
    {
        ...
    }
    else if ( block.isCustom( ) ) //Main Block

```

```
{  
    ....  
}  
}
```

Rendering data points

The build phase creates a DataPointHints object for every run-time series value. This object passes to the before and afterDrawDatapoint and the before and afterDrawDataPointLabel events. The DataPointHints object contains data used by the series renderer to actually render the object. For example, when rendering a two-dimensional bar chart, the script in Listing 11-9 displays the x and y locations in the plot bounds, the category and series values, and size of the bar for each value in the run-time series.

Listing 11-9 Displays x and y locations in a two-dimensional bar chart

```
function beforeDrawDataPoint( dph, fill, icsc )  
{  
    importPackage( Packages.java.io );  
    out = new PrintWriter( new FileWriter(  
        "c:/data/datapoints.txt", true ) );  
    out.println( "BaseValue X-Axis Value " + dph.getBaseValue( ) );  
    out.println( "Orthogonal Y-Axis Value " +  
        dph.getOrthogonalValue( ) );  
    out.println( " X location " + dph.getLocation( ).getX( ) );  
    out.println( " Y Location " + dph.getLocation( ).getY( ) );  
    out.println( " Size " + dph.getSize( ) );  
    out.close( );  
}
```

You can use this data to modify the chart before rendering the data point. For example, using an instance of the ColorDefinitionImpl class, which implements the Fill interface, allows the developer to change the color of a data point. Listing 11-10 uses a script to change a color bar in a data point.

Listing 11-10 Using a script to change a color bar in a data point

```
function beforeDrawDataPoint( dph, fill, icsc )  
{  
    importPackage(  
        Packages.org.eclipse.birt.chart.model.attribute.impl );  
    if( dph.getBaseValue( ) == "Memory" )  
    {  
        var mycolor = ColorDefinitionImpl.ORANGE( );  
        var r = mycolor.getRed( );  
        var g = mycolor.getGreen( );  
        var b = mycolor.getBlue( );  
        fill.set( r, g, b );  
    }  
}
```

```
    }  
}
```

The before and after drawDataPointLabel events receive the DataPointHints object and the label object. The label values can be modified based on the data point values as well. The label object is an instance of the LabelImpl class and offers many methods for altering the label. Listing 11-11 shows how to change the values for a label on a data point.

Listing 11-11 Changing the values for a label on a data point

```
function beforeDrawDataPointLabel(dph, label, icsc)  
{  
    value = dph.getOrthogonalValue( ).doubleValue( );  
    if ( ( value >= 0 ) & ( value <= 500 ) )  
    {  
        label.getCaption( ).getColor( ).set( 32, 168, 255 );  
        label.getCaption().getFont().setItalic(true);  
        label.getCaption().getFont().setRotation(5);  
        label.getCaption().getFont().setStrikethrough(true);  
        label.getCaption().getFont().setSize(22);  
        label.getCaption().getFont().setName("Arial");  
        label.getOutline().setVisible(true);  
        label.getOutline().setThickness(3);  
    }  
    else if ( value >= 500 )  
    {  
        label.getFont().setRotation(5);  
        label.getOutline().setVisible(true);  
    }  
    else if ( value < 0 )  
    {  
        label.getCaption( ).getColor( ).set( 0, 208, 32 );  
    }  
}
```

Rendering legend items

The before and afterDrawLegendItem events are called while the legend is rendering. These events receive an instance of the LegendEntryRenderingHints class and an instance of the BoundsImpl class.

The LegendEntryRenderingHints object contains the following items:

- Data index
- Fill
- Label
- Value label

The data index corresponds to the index for the entry being rendered. The fill object is an instance of a class that implements the Fill interface, which fills the legend entry graphic. The label is an instance of the LabelImpl class and represents the label generated for the specific entry. The value label is an instance of LabelImpl. This label is only used when coloring the legend by values and the show values check box is selected.

The BoundsImpl object contains the top, left, width, and height values for the specific legend entry graphic. You can these values, but be careful to avoid making the legend entry bounds wider or taller than the legend block. Listing 11-12 shows how to make modifications to the legend entry label and alter the legend entry graphic size.

Listing 11-12 Make modifications to the legend entry label

```
function beforeDrawLegendItem( l erh, bounds, icsc )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute.impl );

    label = l erh.getLabel();
    labelString = label.getCaption().getValue();
    if( labelString == "true" )
    {
        label.getCaption().getColor().set( 32, 168, 255 );
        label.getCaption().getFont().setItalic( true );
        label.getCaption().getFont().setRotation( 5 );
        label.getCaption().getFont().setStrikethrough( true );
        label.getCaption().getFont().setSize( 12 );
        label.getCaption().getFont().setName( "Arial" );
        label.getOutline().setVisible( true );
        label.getOutline().setThickness( 3 );

        var mycolor = ColorDefinitionImpl.BLUE();
        r = mycolor.getRed();
        g = mycolor.getGreen();
        b = mycolor.getBlue();
        l erh.setFill().set( r, g, b );

        var graphicwidth = bounds.getWidth();
        var graphicheight = bounds.getHeight();
        var chartModel = icsc.getChartInstance();
        var legendBounds = chartModel.getLegend().getBounds();
        var legendInsets = chartModel.getLegend().getInsets();
        //This does not account for the label text
        var availablewidth = legendBounds.getWidth() -
            legendInsets.getLeft() - legendInsets.getRight();
        if( availablewidth > ( graphicwidth + 15 ) )
        {
            bounds.setWidth( graphicwidth + 15 );
        }
    }
}
```

```

        bounds.setHeight( graphicheight + 15 );
    }
}
}

```

Note that making these same changes in the afterDrawLegendItem has no effect, since the entry is already rendered.

Rendering axes

Many of the Chart scripting events receive an axis object. The axis object is an instance of the AxisImpl class and provides many methods and properties for use in the scripting environment. This features include retrieving the run-time series associated with the axis and setting the min, max, and scale values.

The first thing to do when working with an axis object is determine which axis it is by getting the axis type or getting the axis caption, as shown in Listing 11-13.

Listing 11-13 Determining the axis when working with an axis object

```

function beforeDrawAxisLabel( axis, label, scriptContext )
{
importPackage(
    Packages.org.eclipse.birt.chart.model.attribute );
if ( axis.getType( ) == AxisType.TEXT_LITERAL )
{
    ...
}
}

```

In Listing 11-13, the getType method returns one of the following values:

- LINEAR_LITERAL
- LOGARITHMIC_LITERAL
- TEXT_LITERAL
- DATE_TIME_LITERAL

When the beforeDrawAxisLabel and beforeDrawAxisTitle events trigger, the axis lines and grids are already rendered, so if you want to modify the basic attributes of the axis, you must make these changes in an earlier event. You can use the beforeDrawAxisLabel and beforeDrawAxisTitle events to modify the respective label elements and use the code in Listing 11-13 to determine the axis event being triggered.

If you have multiple y-axes all defined by a specific type such as LINEAR_LITERAL, you can use the axis title to determine which one you are using. The title does not need to be visible, but the value must set. Listing 11-14 illustrates how to check the axis type to modify the label colors.

Listing 11-14 Checking the axis type to modify the label colors

```
function beforeDrawAxisLabel( axis, label, icsc )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute.impl );
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute );
    if( axis.getTitle().getCaption().getValue( ) ==
        "myYaxisTitle" )
    {
        label.getCaption( ).setColor(
            ColorDefinitionImpl.BLUE( ) );
    }
    if ( axis.getType( ) == AxisType.DATE_TIME_LITERAL )
    {
        label.getCaption( ).setColor(
            ColorDefinitionImpl.RED( ) );
    }
}
```

If you want to modify the axis properties do so before you render the axis. Listing 11-15 shows how to retrieve the axis using this approach.

Listing 11-15 Retrieving the axis to modify axis properties

```
function beforeGeneration( chart, icsc )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute.impl );
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute );
    //Currently the chart model only supports one base axis
    xAxis = chart.getBaseAxes( )[ 0 ];
    yAxis = chart.getOrthogonalAxes( xAxis, true )[ 0 ];
    yAxisNumber2 = chart.getOrthogonalAxes( xAxis, true )[ 1 ];
    if ( yAxis.getType( ) ==
        AxisType.DATE_TIME_LITERAL )
    {
        xAxis.setFormatSpecifier(
            JavaDateFormatSpecifierImpl.create( "MM//dd/yyyy" ) );
    }
}
```

In Listing 11-15, the code retrieves all the axes and checks the x-axis to verify that it is a date-and-time axis. After the check, a format specifier is sets for use by the axis labels.

Chart script context

All chart event handler methods for both Java and JavaScript receive a chart script context argument in the form of a ChartScriptContext object. The chart script context object provides access to the following objects:

- Chart instance
- Locale
- ULocale
- Logging
- External context

When using the Chart Engine in a report, the external context object is the reportContext object described in the chapter, “Using JavaScript to Write an Event Handler.” You can call the variables and methods of the reportContext object. For example, you can set the chart axis title to a report parameter using a JavaScript event handler, as shown in Listing 11-16.

Note that retrieving the reportContext in Java and JavaScript is slightly different. To get the reportContext in a chart event handler written in JavaScript, use the context.getExternalContext().getScriptable() method. To get the reportContext in a chart event handler written in Java, use the context.getExternalContext().getObject() method.

Listing 11-16 Setting the chart axis title using a JavaScript event handler

```
function beforeDrawAxisTitle ( axis, title, context )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute );
    if ( axis.getType( ) == AxisType.LINEAR_LITERAL )
    {
        title.getCaption( ).setValue
            ( context.getExternalContext( ).getScriptable( )
                .getParameterValue( "chartTitle" ) );
    }
}
```

You can set the chart axis title using a Java event handler, as shown in Listing 11-17.

Listing 11-17 Setting the chart axis title using a Java event handler

```
public void beforeDrawAxisTitle( Axis axis, Label label,
    IChartScriptContext icsc )
{
    IReportContext rc = ( IReportContext )
        icsc.getExternalContext( ).getObject( );
    String mytitle =
        ( String )rc.getParameterValue( "chartTitle" );
```

```

if ( axis.getType( ) == AxisType.TEXT_LITERAL )
{
    label.getCaption().setValue(mytitle);
}
}

```

Table 11-3 lists the methods of the chart script context object and its functions.

Table 11-3 Chart script context methods

Method	Function
getChartInstance()	Returns the chart instance object.
getExternalContext()	Returns the IExternalContext object that provides access to a scriptable external object. External scriptable objects are defined in the user application.
getLocale()	Returns the Locale object for the locale currently in use.
getLogger()	Returns the Logger object for use in logging messages and errors.
getULocale()	Returns the ULocale object for the locale currently in use.
setChartInstance(Chart)	Sets the chart instance.
setExternalContext(IExternalContext)	Sets the external context.
setLogger(ILogger)	Sets the logger.
setULocale(ULocale)	Sets the ULocale.

Chart instance object

You get a chart instance object from the chart script context object. The chart instance object contains methods that provide access to chart modification functionality. Use the chart instance object to get, change, and test properties.

Chart instance getter methods

The chart instance getter methods allow you to get the properties of a chart. Table 11-4 lists the chart instance getter methods and returning property values.

Table 11-4 Chart instance getter methods

Method	Gets
getBlock()	Value of the Block containment reference
getDescription()	Value of the Description containment reference

(continues)

Table 11-4 Chart instance getter methods (*continued*)

Method	Gets
getDimension()	Value of the Dimension attribute
getExtendedProperties()	Value of the Extended Properties containment reference list
getGridColumnCount()	Value of the Grid Column Count attribute
getInteractivity()	Value of the Interactivity containment reference
getLegend()	Legend block
getPlot()	Plot block
getSampleData()	Value of the Sample Data containment reference
getScript()	Value of the Script attribute
getSeriesForLegend()	Array of series containing captions or markers that render in the Legend
getSeriesThickness()	Value of the Series Thickness attribute
getStyles()	Value of the Styles containment reference list
getSubType()	Value of the Sub Type attribute
getTitle()	Title block for the chart
getType()	Value of the Type attribute
getUnits()	Value of the Units attribute
getVersion()	Value of the Version attribute

Chart instance setter methods

The chart instance setter methods allow you to set various properties of a chart. Table 11-5 lists the chart instance setter methods and the values set by these methods.

Table 11-5 Chart instance setter methods

Method	Sets
setBlock(Block value)	Value of the Block containment reference
setDescription(Text value)	Value of the Description containment reference
setDimension(Chart Dimension value)	Value of the Dimension attribute
setGridColumnCount(int value)	Value of the GridColumnCount attribute
setInteractivity(Interactivity value)	Value of the Interactivity containment reference

Table 11-5 Chart instance setter methods (*continued*)

Method	Sets
setSampleData()	Value of the Sample Data containment reference
setScript()	Value of the Script attribute
setSeriesThickness()	Value of the Series Thickness attribute
setSubType()	Value of the Sub Type attribute
setType()	Value of the Type attribute
setUnits()	Value of the Units attribute
setVersion()	Value of the Version attribute

Miscellaneous chart instance methods

Table 11-6 lists miscellaneous chart instance method chart instance methods and describes the action performed by each method. Use these chart instance methods when the new simple charting API is not sufficient to meet your requirements.

Table 11-6 Miscellaneous chart instance methods

Method	Action
clearSections(int iSectionType)	Walks through the model and clears sections of the specified type
createSampleRuntimeSeries()	Builds run-time series instances for each design-time series based on the sample data contained in the model
setDimension(Chart Dimension value)	Sets the value of the Dimension attribute
unsetDimension()	Unsets the value of the Grid_Column_Count attribute
unsetGridColumn()	Unsets the value of the Dimension attribute
unsetSeriesThickness()	Unsets the value of the Series_Thickness attribute
unsetVersion()	Unsets the value of the Version attribute

The chart instance is also available in the beforeGeneration event, which occurs after the data set events just before creating the chart. In Listing 11-18, the JavaScript example contains a beforeGeneration event handler that sets the unit spacing for the entire chart. The beforeDrawSeries event handler uses the chart context to retrieve the chart instance and set the unit spacing for just one series, which is titled series one.

If you use this script with a chart containing a two bar series, this process makes one of the bar series wider than the other. The afterDrawSeries resets the unit spacing for the series. Note that setUnitSpacing only works with a ChartWithAxes instance, which extends the Chart object.

Listing 11-18 Setting unit spacing in the beforeGeneration event handler

```
function beforeGeneration( chart, context )
{
    chart.setUnitSpacing( 20 );
}
var oldSpacing;

function beforeDrawSeries( series, seriesRenderer, context )
{
    oldSpacing =
        context.getChartInstance().getUnitSpacing();
    if( series.getSeriesIdentifier() == "series one" ){
        context.getChartInstance().setUnitSpacing( 70 );
    }
}

function afterDrawSeries( series, seriesRenderer, context )
{
    context.getChartInstance().setUnitSpacing(oldSpacing);
}
```

Writing a Java chart event handler

You write a Java chart event handler the same way you write a Java event handler for any other kind of report item. The only exception is that the chart event handler must contain all the events for the chart that you want to implement in the same handler class. Writing an event handler in Java for a chart also requires additional JAR files in the build path.

Setting up the chart event handler project

To create a new Java chart event handler project you must have the following JAR files in the build path and classpath:

- chartengineapi.jar
- org.eclipse.emf.ecore_version.jar
- org.eclipse.emf.common_version.jar

When accessing the reportContext, you need the scriptapi.jar. If you are using the ULocale methods, you also need com.ibm.icu_version.jar.

When writing a Java event handler for a chart, you have the option of implementing the `IChartEventHandler` interface or extending the `ChartEventHandlerAdapter`. The examples in this chapter extend the adapter.

After writing a chart event handler, you can apply it by selecting the chart in the report design and selecting Properties. On Properties, select the event handler property and enter the fully qualified class name or select Browse to locate the class. Only the classes in your workspace or classpath that meet the requirements of the `IChartEventHandler` or `ChartEventHandlerAdapter` contract display.

You can debug chart event handlers written in Java in the same way that you debug other Java event handlers.

Chart Java event handler examples

Charts are composed of blocks. These blocks are rectangular sub-regions, in the chart that contain sub-components. For example, the plot block contains the actual rendering of the data points, and the title block contains the chart title. The `beforeDrawBlock` event fires before drawing each block. Using this event handler, you can modify many properties of the block before the chart renders.

This section provides some common examples of chart Java event handlers. The code in Listing 11-19 outlines the legend block in red and the plot block in green, and sets the background color of the title block to cream and its outline to blue.

Listing 11-19 Setting outline and background colors in a chart

```
package my.chart.events;

import org.eclipse.birt.chart.model.layout.Block;
import org.eclipse.birt.chart.model.attribute.impl
    .ColorDefinitionImpl;
import org.eclipse.birt.chart.script.ChartEventHandlerAdapter;
import org.eclipse.birt.chart.script.IChartScriptContext;

public class BlockScript extends ChartEventHandlerAdapter
{
    public void beforeDrawBlock( Block block,
        IChartScriptContext icsc )
    {
        if ( block.isLegend( ) )
        {
            block.getOutline( ).setVisible( true );
            block.getOutline( ).getColor( ).set( 255, 0, 0 );
        }
        else if ( block.isPlot( ) )
        {
```

```

        block.getOutline( ).setVisible( true );
        block.getOutline( ).getColor( ).set( 0, 255, 0 );
    }
    else if ( block.isTitle( ) )
    {
        block.getOutline( ).setVisible( true );
        block.setBackground( ColorDefinitionImpl.CREAM( ) );
        block.getOutline( ).getColor( ).set( 0, 0, 255 );
    }
}

```

The beforeDrawSeries event triggers before drawing each series in a chart. Listing 11-20 sets the series labels to red and also applies a curve fitting line to the series. If the series is a line series, the marker type changes to a triangle.

Listing 11-20 Setting a series label color and applying a curve fitting line

```

package my.chart.events;

import org.eclipse.birt.chart.model.component.Series;
import org.eclipse.birt.chart.model.component.impl
    .CurveFittingImpl;
import org.eclipse.birt.chart.model.attribute.MarkerType;
import org.eclipse.birt.chart.model.attribute.Marker;
import org.eclipse.birt.chart.model.type.LineSeries;
import org.eclipse.birt.chart.render.ISeriesRenderer;
import org.eclipse.birt.chart.script.ChartEventHandlerAdapter;
import org.eclipse.birt.chart.script.IChartScriptContext;

public class SeriesScript extends ChartEventHandlerAdapter
{

    public void beforeDrawSeries( Series series, ISeriesRenderer
        isr, IChartScriptContext icsc )
    {
        if( series instanceof LineSeries )
        {
            Marker mk =
                ( ( Marker )( ( LineSeries )series )
                    .getMarkers( ).get( 0 ) );
            mk.setType( MarkerType.TRIANGLE_LITERAL );
        }
        series.setCurveFitting( CurveFittingImpl.create( ) );
        series.getLabel( ).getCaption( ).getColor( )
            .set( 255, 0, 0 );
    }
}

```

In Listing 11-21, the code creates two event handlers, one for the beforeGeneration event and the other for beforeDrawSeries event. The

beforeGeneration event handler sets the plot background color to grey. If the chart has axes, the event handler sets the x-axis labels to forty-five degree angles and places the labels below the axis. This event handler also sets the major grid lines for the y-axis to visible. The beforeDrawSeries event handler adds a mouse-over event to the series to show the value of the current data point.

Listing 11-21 Creating beforeGeneration and beforeDrawSeries event handlers

```
package my.chart.events;

import org.eclipse.birt.chart.model.Chart;
import org.eclipse.birt.chart.model.impl.ChartWithAxesImpl;
import org.eclipse.birt.chart.render.ISeriesRenderer;
import org.eclipse.birt.chart.script.ChartEventHandlerAdapter;
import org.eclipse.birt.chart.script.IChartScriptContext;
import org.eclipse.birt.chart.model.component.Series;
import org.eclipse.birt.chart.model.component.Axis;
import org.eclipse.birt.chart.model.attribute.AttributeType;
import org.eclipse.birt.chart.model.attribute.LegendItemType;
import org.eclipse.birt.chart.model.attribute.TriggerCondition;
import org.eclipse.birt.chart.model.data.impl.TriggerImpl;
import org.eclipse.birt.chart.model.data.impl.ActionImpl;
import org.eclipse.birt.chart.model.attribute.impl
    .ColorDefinitionImpl;
import org.eclipse.birt.chart.model.attribute.Position;
import org.eclipse.birt.chart.model.data.Action;
import org.eclipse.birt.chart.model.attribute.impl
    .TooltipValueImpl;
import org.eclipse.birt.chart.model.attribute.TooltipValue;

public class ChartModScript extends ChartEventHandlerAdapter {

    public void beforeGeneration(Chart cm,
        IChartScriptContext icsc)
    {
        cm.getPlot().getClientArea().setBackground(
            ColorDefinitionImpl.GREY());
        if (cm instanceof ChartWithAxesImpl)
        {
            // x axis
            Axis xaxis = ((ChartWithAxesImpl)cm)
                .getPrimaryBaseAxes()[0];
            xaxis.setLabelPosition(Position.BELOW_LITERAL);
            xaxis.getLabel().getCaption().getFont()
                .setRotation(45);
            Axis yaxis = ((ChartWithAxesImpl)cm)
                .getPrimaryOrthogonalAxis(xaxis);
            yaxis.getMajorGrid().getLineAttributes()
```

```

        .setVisible( true );
    }

    public void beforeDrawSeries( Series series, ISeriesRenderer
        isr, IChartScriptContext icsc )
    {
        TooltipValue tt = TooltipValueImpl.create( 500, null );
        Action ac = ActionImpl.create(
            ActionType.SHOW_TOOLTIP_LITERAL, tt );
        series.getTriggers( ).add(
            TriggerImpl.create(
                TriggerCondition.ONMOUSEOVER_LITERAL, ac ) );
    }
}

```

Writing a JavaScript chart event handler

The process of writing a JavaScript chart event handler differs from the process of writing a JavaScript event handler for other report items. In the BIRT Report Designer, Script displays only the `onRender` event in the list of JavaScript events for a chart report item. The list next to the JavaScript events list contains a list of the active chart events, as shown in Figure 11-1.

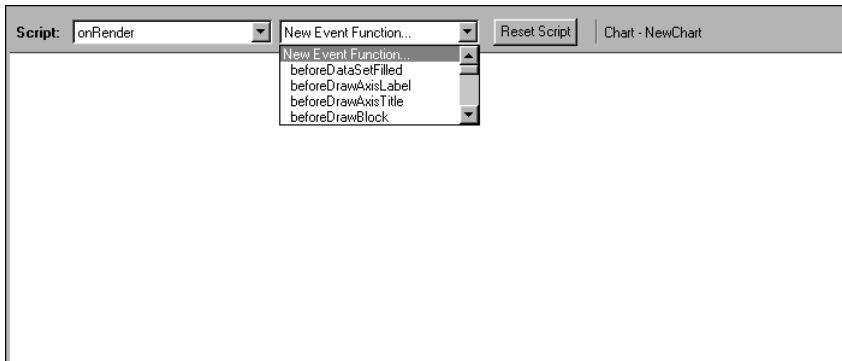


Figure 11-1 Script for a chart in BIRT Report Designer

When coding chart events, you must include every event handler script for the chart in the same location. When you select one of the events from the list, BIRT adds a stub for that event in the `onRender` script window, as shown in Figure 11-2.

The screenshot shows a software interface for writing chart scripts. At the top, there's a toolbar with buttons for 'Script:' (set to 'onRender'), 'New Event Function...', 'Reset Script', and 'Chart - NewChart'. Below the toolbar is a code editor area containing the following JavaScript code:

```


/**
 * Called before populating the series dataset using the DataSetProcessor.
 *
 * @param series
 *     Series
 * @param idsp
 *     IDataSetProcessor
 * @param icsc
 *     IChartScriptContext
 */

function beforeDataSetFilled(series, idsp, icsc)
{
}


```

Figure 11-2 Chart script stub

The list contains all chart events. To write handler code for an event, type the handler code between the parentheses, as shown in Figure 11-3.

The screenshot shows a software interface for writing chart scripts. At the top, there's a toolbar with buttons for 'Script:' (set to 'onRender'), 'New Event Function...', 'Reset Script', and 'Chart - NewChart'. Below the toolbar is a code editor area containing the following JavaScript code:

```


/**
 * Called before populating the series dataset using the DataSetProcessor.
 *
 * @param series
 *     Series
 * @param dataSetProcessor
 *     DataSetProcessor
 * @param context
 *     IChartScriptContext
 */

function beforeDataSetFilled( series, dataSetProcessor, context )
{
    logger.logFromScript("Logging before data set filled");
}


```

Figure 11-3 Chart event handler stub

The beforeDrawAxisLabel event triggers for each label rendered on a chart axis. Providing an event handler for this event allows you to modify the labels. In Listing 11-22, the modified labels provide an abbreviated number format, making the chart more readable.

Listing 11-22 Modifying labels in a beforeDrawAxisLabel event

```


function beforeDrawAxisLabel( axis, label, context )
{
    value = label.getCaption().getValue();
    if ( value >= 1000 && value < 1000000 )
        value = value/1000 + "k";
    else if ( value >= 1000000 )
        value = value/1000000 + "M";
}


```

```
    label.getCaption( ).setValue( value );
}
```

The beforeDrawDataPoint and afterDrawDataPoint events trigger before and after rendering a data point in a chart. You can use these events to modify how the data point is rendered. In Listing 11-23, the event handler renders a bar chart with positive and negative bars and colors the negative bars red.

Listing 11-23 Rendering a bar chart with positive and negative bars

```
previousFill = null;
function beforeDrawDataPoint( dph, fill, icsc )
{
    val = dph.getOrthogonalValue( );
    if ( val < 0 ){
        previousFill = new Object( );
        previousFill.r = fill.getRed( );
        previousFill.g = fill.getGreen( );
        previousFill.b = fill.getBlue( );
        fill.set( 255, 0, 0 );
    }
    else{
        previousFill = null;
    }
}

function afterDrawDataPoint( dph, fill, icsc )
{
    if ( previousFill != null ){
        fill.set( previousFill.r, previousFill.g,
                  previousFill.b );
    }
}
```

The beforeDrawDataPoint event handler retrieves the value and determines if the bar is negative. If the bar is negative, the event handler saves the current fill color. The afterDrawDataPoint uses these settings to restore the color, setting the color back to red.

The beforeDrawAxisTitle event triggers before rendering an axis title. Writing an event handler for this event allows modification of the rendered title. In Listing 11-24, the event handler modifies the y-axis title bar in a bar chart to display the string, Modified Title, in blue.

Listing 11-24 Modifying the y-axis title bar in a bar chart

```
function beforeDrawAxisTitle ( axis, title, context )
{
    importPackage(
        Packages.org.eclipse.birt.chart.model.attribute );
    if ( axis.getType( ) == AxisType.LINEAR_LITERAL )
    {
```

```

        title.getCaption( ).setValue( "Modified Title" );
    }
    title.getCaption( ).getColor( ).set( 0, 0, 255 );
}

```

The beforeDrawSeries event triggers prior to rendering a series in a chart. You can modify a series using an event handler for this event. In Listing 11-25, the event handler sets the visibility for the series identifier, SalesBalance, to false, so the series does not render in the chart.

Listing 11-25 Using the beforeDrawSeries event to visibility in a chart

```

function beforeDrawSeries( series, seriesRenderer, context )
{
    if ( series.getSeriesIdentifier() == "SalesBalance" )
    {
        series.setVisible( false );
    }
}

```

The afterDataSetFilled event triggers for each series after it populates with data from the data set. You can use an event handler here to modify the data before rendering the series. In Listing 11-26, the event handler sets a threshold on values in a series, so that, if a value is less than the threshold, the value is set to zero. If you use this script on a bar chart, the bar shows nothing for any value under 240000.

Listing 11-26 Setting a threshold on values in a series

```

function afterDataSetFilled( series, dataSet, icsc )
{
    // insert if code to identify your series here if needed.
    var list = dataSet.getValues( );
    for ( i=0; i<list.length; i=i+1 )
    {
        if ( list[i] < 240000 )
        {
            list[i]= 0;
        }
    }
}

```

Using the simplified charting API

BIRT version 2.2 introduced a simplified API for manipulating chart properties. In earlier versions, accessing the chart model was less straightforward and the differences between the chart model and the report model often led to confusion. The new simplified charting API integrates the

chart model with the report model script API, and the new API uses an interface that is more like the API for other report items.

Listing 11-27 shows how earlier versions of BIRT set color by category.

Listing 11-27 Setting color by category in earlier versions of BIRT

```
public void beforeGeneration( Chart cm,
    IChartScriptContext icsc )
{
    //Set color by category
    cm.getLegend().setItemtype(
        LegendItemtype.CATEGORIES_LITERAL );
}
```

Listing 11-28 shows how you can implement the same functionality using the simplified charting API in the beforeFactory event handler for a report that contains a chart.

Listing 11-28 Setting color by category using the simplified charting API

```
var chart1 = this.getReportElement( "Chart1" )
chart1.setColorByCategory( true );
```

Typically, you use the simplified charting API when programmatically modifying an existing chart that you created and formatted in the BIRT Report Designer. Given a report design that contains a formatted chart, your Java application can use the simplified charting API to modify the chart's content.

Getting an instance of a chart item

An important part of the simplified charting API is the IChart interface. Like the other report item interfaces, IChart extends IReportItem. Because of the more complex nature of the Chart item, the IChart interface is in a different package and has a more extensive set of methods than the other report item interfaces.

Use the following code to get an IChart object in a Java application that is using the Design Engine API or when developing a Java event handler:

```
IChart chart = (IChart) rptdesign.getReportElement( "Chart
name" );
```

If you are using a JavaScript event handler, use the following code to retrieve the IChart object.

```
var chart1 = this.getReportElement( "Chart1" )
```

Be certain to name the chart using the name property to access the chart in this fashion.

Understanding the sub-interfaces of IChart

There are two sub-interfaces of IChart:

- IChartWithAxes
- IChartWithoutAxes

IChartWithoutAxes only adds one method to IChart, while IChartWithAxes adds several. When using Java, you typically cast the chart report item to an appropriate sub-interface for the chart, as shown in the following code:

```
IChartWithAxes chart = ( IChartWithAxes )  
    rptdesign.getReportElement( "Name of an Axis-containing  
    Chart" );
```

Table 11-7 lists the methods of the IChart interface.

Table 11-7 IChart, IChartWithAxes, and IChartWithout Axes methods

Method	Action
getDescription()	Returns an IText containing the chart description, enables changing the description
getDimension()	Gets the current dimension setting.
setDimension(String type)	Set the dimension for the chart: <ul style="list-style-type: none">■ TwoDimensional■ TwoDimensionalWithDepth■ ThreeDimensional
getTitle()	Returns an IText containing the chart title. Allows changing the title.
isColorByCategory()	Returns color by category setting.
getLegend()	Returns an ILegend object containing the chart legend.
getCategory()	Returns an ICategory object containing the category. You can use this object to change settings, such as grouping and sorting for a chart category.
setColorByCategory(boolean flag)	Indicates whether to set the color by category or not.
getOutputType()	Returns a String containing the output type.

(continues)

Table 11-7 IChart, IChartWithAxes, and IChartWithout Axes methods (*continued*)

Method	Action
setOutputType(String type)	Sets the output type of the chart to one of the following formats: <ul style="list-style-type: none">■ PNG■ SVG■ JPG■ BMP
getFactory()	Returns an IComponentFactory object for creating chart components.

Table 11-8 lists the methods of the IChartWithAxes interfaces.

Table 11-8 IChartWithAxes methods

Method	Action
getCategoryAxis()	Returns an IAxis object containing the category axis.
getValueAxes()	Returns an array of IAxis objects containing the set of value axes.
isHorizontal()	Returns a boolean indicating whether the chart orientation is horizontal or not.
setHorizontal(boolean flag)	Sets the orientation of the chart to horizontal when true or vertical when false.
getValueSeries()	Returns a two dimensional array of IValueSeries objects containing the value series.

Table 11-9 lists the methods of the IChartWithoutAxes interfaces.

Table 11-9 IChartWithout Axes methods

Method	Action
getValueSeries()	Returns a one dimensional array of IValueSeries objects containing the value series.

You can also use the simplified charting API in a report script. You use the API most frequently in a beforeFactory event handler, which you call prior to generating the report. You can still modify the design of the report in this event handler. Listing 11-29 uses the following script in the beforeFactory of

the report, locates a chart named Chart1, sets the chart title to the string, My New Title, and renders the title in red.

Listing 11-29 Using the beforeFactory event handler

```
var chart1 = this.getReportElement( "Chart1" );
var color1 = chart1.getTitle().getCaption().getColor();

chart1.setColorByCategory( true );
chart1.getTitle().getCaption().setValue( "My New Title" );

color1.setRed( 255 );
color1.setGreen( 0 );
color1.setBlue( 0 );
chart1.getTitle().getCaption().setColor( color1 );
```

Adding the following line to the script changes the chart to a three dimensional chart:

```
chart1.setDimension( "ThreeDimensional" );
```

You can also set the chart output type using the following line of script:

```
chart1.setOutputType("PNG");
```

You can set chart dimensions by using the setWidth and setHeight methods since IChart extends from ReportItem. For example, to set the width and height to six inches, use the following script:

```
chart1.setWidth("6in");
chart1.setHeight("6in");
```

This page intentionally left blank

12

Accessing Data Programmatically

BIRT supports accessing a data source using JavaScript code. A data source that you access using JavaScript is called a scripted data source. Using a scripted data source, you can access objects other than an SQL, XML, or text file data source.

Because the JavaScript code for accessing and managing a scripted data source can wrap Java objects, a scripted data source can be an EJB, an XML stream, a Hibernate object, or any other Java object that retrieves data. A scripted data source must return data in tabular format, so that BIRT can perform sorting, aggregation, and grouping.

Using a Scripted Data Source

Creating a scripted data source and creating a non-scripted data source are similar tasks. The differences between creating a scripted data source and a non-scripted data source are

- The report developer must select Scripted Data Source from the list of data source types when creating a scripted data source.
- The report developer can provide code for two event handler methods, `open()` and `close()`, that are only available for a scripted data source.

Every scripted data source must have at least one scripted data set. The differences between creating a scripted data set and a non-scripted data set are

- The report developer must associate the scripted data set with a scripted data source.

- The report developer must provide code for the scripted data set `fetch()` event handler method.
- The report developer uses a different dialog for identifying the columns of a scripted data set than the dialog used for a non-scripted data set.

When you use BIRT Report Designer to create a scripted data source, you must perform the following tasks:

- Create a scripted data source

Right-click on Data Sources in Data Explorer and select Scripted Data Source in the list of data source types.

- Create a scripted data set

Right-click on Data Sets in Data Explorer and select a scripted data source from the list of available data sources.

- Define output columns

Define the names and types of output columns, using the scripted data set editor.

- Supply code for the data source `open()` and `close()` methods

There are two scripted data source event handler methods, `open()` and `close()`. It is not mandatory that you implement either method, but most applications require the use of the `open()` method.

Use the `open()` method to initialize a data source. Typically, you create a Java object for accessing the data source in the `open()` method.

Use the `close()` method to clean up any loose ends, including setting object references to null to ensure that the objects are deleted during garbage collection.

- Supply code for the data set methods

There are three scripted data set event handler methods, `open()`, `fetch()`, and `close()`. Implementing the `fetch()` method is mandatory.

Use the `open()` method to initialize variables and to prepare the data source for fetching data.

Use the `fetch()` method to get a row of data from the data source and to populate the columns of the row object. The `fetch()` method must return either true or false. A true value tells BIRT that there is another row to process. A false return value signifies that there are no more rows to process.

Use the `close()` method to perform cleanup operations.

- Place the columns on the report layout

Place a data set column on a report layout the same way you place a column for a non-scripted data set.

The following tutorial guides you through the procedure required to perform each task in this process.

Tutorial 2: Creating and scripting a scripted data source

This tutorial provides instructions for creating and scripting a simulated scripted data source. Although this tutorial does not use an actual data source, you learn the process.

In this tutorial, you perform the following tasks:

- Create a new report design
- Create a scripted data source
- Create a scripted data set
- Supply code for the open() and close() methods of the data source
- Supply code for the open() method of the data set
- Define output columns
- Place the columns on the report layout
- Supply code for the fetch() method of the data set

Task 1: Create a new report design

In this task, you create a new report in BIRT Report Designer and name it ScriptedDataSrc.rptdesign.

- 1 Choose File->New->Report.
- 2 In File Name in New Report, type:
`ScriptedDataSrc.rptdesign`
- 3 In Enter or Select the Parent Folder, accept the default folder. Choose Next.
- 4 In Report Templates, select My First Report. Choose Finish. The BIRT report design screen appears. If a Cheat Sheet tab appears, close it.

Task 2: Create a scripted data source

In this task you create the new data source.

- 1 In Data Explorer, right-click Data Sources and choose New Data Source. Select a Data Source type appears.
- 2 In New Data Source, select Scripted Data Source.

- 3** In Data Source Name, type:

ScriptedDataSource

- 4** Choose Finish.

Data Explorer and the code window for ScriptedDataSource appear, as shown in Figure 12-1.



Figure 12-1 Data Explorer and ScriptedDataSource code window

Task 3: Create a scripted data set

In this task, you create the new data set.

- 1** In Data Explorer, right-click Data Sets. Choose New Data Set. New Data Set appears, as shown in Figure 12-2.

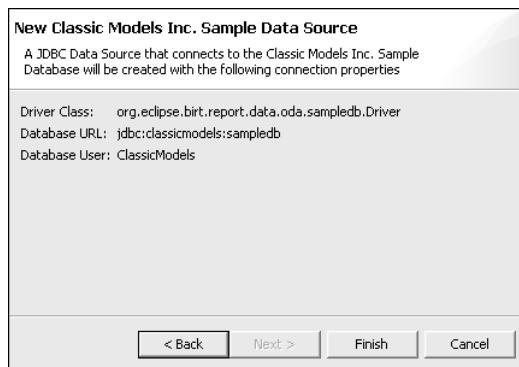


Figure 12-2 New data set for a scripted data source

- 2** In Data Set Name, type:

ScriptedDataSet

- 3** Choose Finish.

- 4** In Data Explorer, select ScriptedDataSet. The script window for the data set appears, as shown in Figure 12-3.



Figure 12-3 Code window for ScriptedDataSet

Task 4: Supply code for the open() and close() methods of the data source

In the open() method, you open the data source. In the close() method, you do cleanup tasks. In this tutorial, there is no actual data source, but typically you need to place some code in these methods. The open() method is the default selected method upon creating a data set.

- 1 If necessary, select open from the pull-down list of methods.
- 2 Type the following code into the code window for the open() method:

```
dummyObject = new Object();
```

The previous example code is placeholder code for this simplified example. In a typical application, you use this method to initialize a Java object that provides access to the data for the report.

- 3 Select close from the pull-down list of methods.
- 4 Type the following code into the code window:

```
dummyObject = null;
```

Task 5: Supply code for the open() method of the data set

When you create the data set, the open() method is selected by default. Use the open() method of the data set to do initialization, such as defining a counter and setting it to zero.

- 1 Select open from the pull-down list of methods.
- 2 Type the following code into the code window:

```
recordCount = 0;
```

Task 6: Define output columns

To create the output columns for a scripted data set, you must edit the data set. The columns you create in the data set editor are the columns that the data set fetch() method generates.

- 1 In Data Explorer, double-click ScriptedDataSet. Edit Data Set—ScriptedDataSet appears, as shown in Figure 12-4.

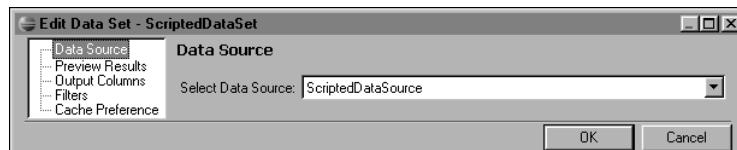


Figure 12-4 Edit Data Set for a scripted data source

- 2 Select Output Columns. Output Columns appears.

- 3** In the Name column of the first row, type:

col1

- 4** Select the Type column of the first row. Select Integer from the drop-down list. Output Columns contains the definition of one output column, as shown in Figure 12-5.

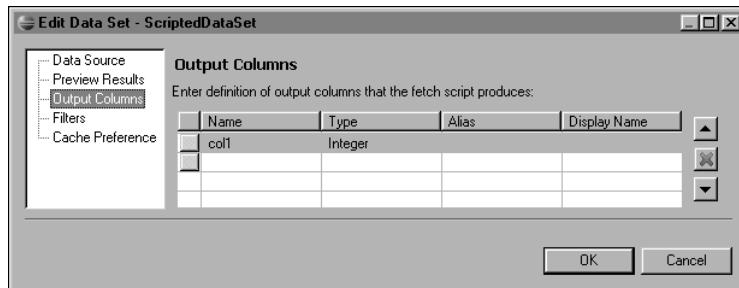


Figure 12-5 Column name and type in Output Columns

- 5** In the Name column of the second row, type:

col2

- 6** In the Type column of the second row, select String.

- 7** In the Name column of the third row, type:

col3

In the Type column of the third row, select Float. Output Columns contains the definition of three output columns, as shown in Figure 12-6. Choose OK.

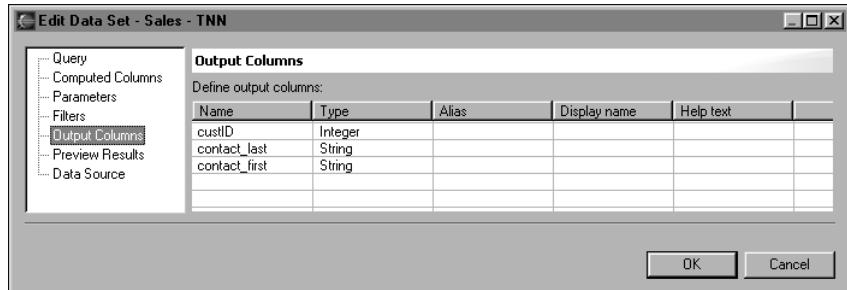


Figure 12-6 Column definitions

Task 7: Place the columns on the report layout

You place columns for a scripted data set in the same way as for a non-scripted data set.

- 1 On ScriptedDataSrc.rptdesign, select Layout.
- 2 Drag Table from Palette into the report layout.

- 3 Accept the default table size of three columns and one detail row.
- 4 In Data Explorer, expand ScriptedDataSet. The three columns you created appear in Data Explorer, as shown in Figure 12-7.

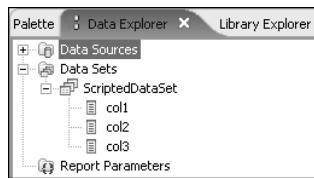


Figure 12-7 New columns in ScriptedDataSet

- 5 Add the columns to the report detail row:
 - 1 Drag col1 from Data Explorer to the first column of the report detail row.
 - 2 Drag col2 from Data Explorer to the second column of the report detail row.
 - 3 Drag col3 from Data Explorer to the third column of the report detail row.

Figure 12-8 shows the three columns of the data set in the layout editor.

col1	col2	col3
row["col1"]	row["col2"]	row["col3"]
Footer Row		

Figure 12-8 New columns in the report design

- 6 Choose Preview.

The preview of the report appears, as shown in Figure 12-9.

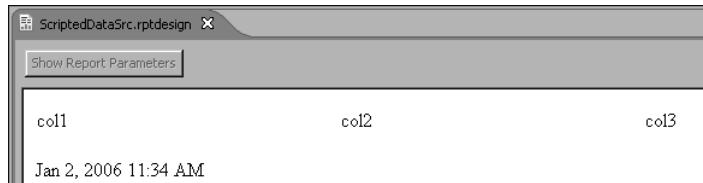


Figure 12-9 Report preview, showing the new columns

Task 8: Supply code for the fetch() method of the data set

Use the `fetch()` method to process row data. The `fetch()` method must return either true or false. `Fetch()` returns true to indicate that there is a row to process. `Fetch()` returns false to indicate that there are no more rows to process. The `fetch()` method also calculates the values of computed fields. The report only has column headings at this point. To include data, you must add code to the `fetch()` method.

- 1 Choose the Layout tab.
- 2 Right-click ScriptedDataSet in Data Explorer. Choose Edit Script.
- 3 Select fetch in the drop-down list of methods.
- 4 Select fetch from the pull-down list of methods in the data set code window.
- 5 Type the following code into the code window. This code limits the number of rows that appear in the report to 19.

```
if(recordCount < 20) {  
    recordCount++;  
    row.col1 = recordCount;  
    row["col2"] = "Count = " + recordCount;  
    row[3] = recordCount * 0.5;  
    return true;  
}  
else return false;
```

- 6 Choose Preview.

The report now contains 20 rows and 3 columns of data, as shown in Figure 12-10.

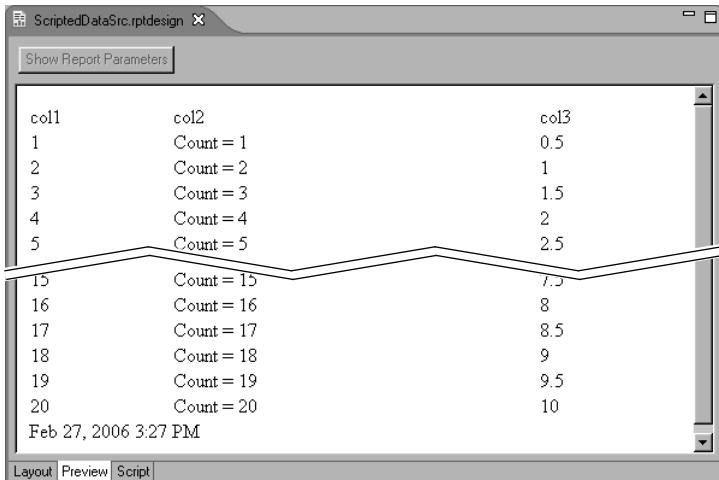


Figure 12-10 Report preview

Writing the scripted data set in Java

You can also implement this example in Java. Setup the Java project in the same workspace as the BIRT report project. In the report project, repeat the previous tasks, omitting tasks 4, 5, and 8.

In the Java project, add the Java class file in Listing 12-30. Finally, in the report project, select the scripted data set and set the event handler class property to the class in Listing 12-30.

Listing 12-30 MyScriptedDataSet.java

```
import org.eclipse.birt.report.engine.api.script
    .IScriptedDataSetMetaData;
import org.eclipse.birt.report.engine.api.script
    .IUpdatableDataSetRow;
import org.eclipse.birt.report.engine.api.script.eventadapter
    .ScriptedDataSetEventAdapter;
import org.eclipse.birt.report.engine.api.script.instance
    .IDataSetInstance;

public class MyScriptedDataSet extends
    ScriptedDataSetEventAdapter {

    public int recordCount = 0;

    @Override
    public boolean fetch( IDatasetInstance dataSet,
        IUpdatableDataSetRow row ) {
        try{
            if( recordCount < 20) {
                recordCount++;
                row.setColumnValue( "col1", recordCount );
                row.setColumnValue( "col2", "Count =
                    " + recordCount );
                row.setColumnValue( "col3", recordCount*.05 );
                return true;
            }else{
                return false;
            }
        }catch( Exception e ){
            e.printStackTrace( );
            return false;
        }
    }

    @Override
    public void open( IDatasetInstance dataSet ) {
        recordCount = 0;
    }
}
```

Selecting Browse for the event handler property displays the scripted data set class. This example shows how to implement the scripted data set in Java. You can implement a scripted data source in a similar way.

In this example, the recordCount is stored as a global variable of the MyScriptedDataSet object. Using the global variable in this way is only valid for the data source or data set event handlers and should not be used when

extending other event adapters. All other event adapters create a new instance of the event handler class for each instance of a report item. For example, when extending the RowEventAdapter, a new instance of the extending class is created for each row that uses the extended adapter.

Using a Java object to access a data source

A common use of a scripted data set is to access a Java object that accesses or generates the data for a report. This section shows how to access a Java class in the JavaScript code for a scripted data set.

Performing initialization in the data set open() method

Use the data set open() method to perform initialization tasks. A typical initialization task is to get an instance of the Java object that provides the data for the report.

When referring to a Java object, first import its package into the JavaScript environment. For example, the following code imports the package com.yourCompany.yourApplication:

```
importPackage( Packages.com.yourCompany.yourApplication );
```

This statement is like the import statement in Java and allows you to omit the package name when referencing a class. This statement is normally the first line in the open() method. You typically follow the importPackage statement with code to create the Java object instance, as shown in the following code:

```
var myList = myListFactory.getList( );
```

A typical way of getting rows of data from a Java object is to use an iterator object. The open() method is the proper place to create an iterator object. For example, the following statement gets an iterator from myList:

```
var iterator = myList.getIterator( );
```

Getting a new row of data in the data set fetch() method

Once you have a way to get rows of data from your Java object, use the fetch() method to call the Java method that returns the rows. The fetch() method determines if there are any more rows of data and returns false if there are none, as shown in the following code:

```
if( iterator.hasNext( ) == false ){
    return false;
}
```

At this point, the fetch() method can populate a row with the data that it gets from the iterator, as shown in the following code:

```
var node = iterator.next( );
row[1] = node.getFirstCol( );
row[2] = node.getSecondCol( );
row[3] = node.getThirdCol( );
```

You must return true to signal BIRT that there is a valid row of data to process, as shown in the following code:

```
return true;
```

Cleaning up in the data set close() method

You can perform any cleanup in the close() method. This method is a good place to set to null any objects that you created. For example, the following code sets three object references to null:

```
myList = null;  
iterator = null;  
node = null;
```

Deciding where to locate your Java class

If a scripted data source uses a custom Java class, that class must reside in a location where BIRT can find it. BIRT can find the Java class if its location meets any of the following requirements:

- The Java class is in the classpath of the Java Runtime Environment (JRE) under which Eclipse runs.
Consider using this option if your Java class is in this location for other reasons.
- The Java class is in <ECLIPSE_INSTALL>\plugins\org.eclipse.birt.report.viewer\birt\WEB-INF\lib.
Consider using this option if your Java class is built, tested, and ready to deploy.
- The Java class is a part of an Eclipse Java project that is in the same workspace as the BIRT report project.
Consider using this option if you are developing your Java class simultaneously with developing your BIRT report.

Deploying your Java class

Before you deploy your BIRT report to an application server, you must place your Java class in a JAR file. You must then deploy that JAR file to the proper location on the application server, so that the BIRT report viewer can find it at run time.

Using input and output parameters with a scripted data set

The scripted data set JavaScript event handler methods have two arrays you can use to access parameters, `inputParams` and `outputParams`. The `inputParams` array contains one string for every parameter whose direction is

defined as input. The outputParams array contains one string for every parameter whose direction is defined as output.

For example, assume that you have a scripted data set with an input and an output parameter, as shown in Figure 12-11.

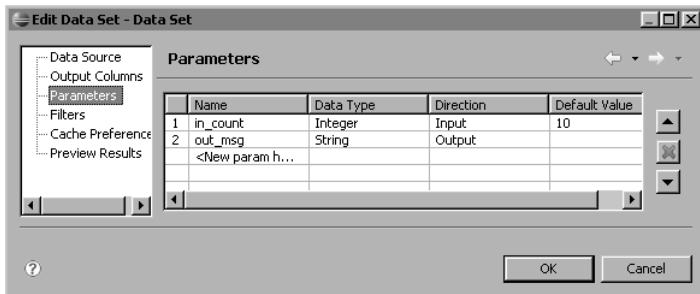


Figure 12-11 A scripted data set, with input and output parameters

You can get and set the values of the out_msg and in_count parameters by using the inputParams and outputParams arrays as in the following example:

```
outputParams[ "out_msg" ] = "Total rows: " +  
    inputParams[ "in_count" ];
```

You can access a parameter in the array either by the name of the parameter or by a 1-based index value. The inputParams and outputParams arrays are not accessible to Java event handlers.

Creating a web services data source using a custom connection class

As stated earlier, you can use a custom connection class to create a Web Services data source. The custom connection class is responsible for returning an Input Stream that contains a SOAP XML response. This class must implement a connect() method. This method has to return an Object, implementing a executeQuery() and disconnect() method.

The connect() method accepts two parameters, which contain the connection properties and application context.

BIRT uses an Application Context Map to store values and objects for use in all phases of report generation and presentation. You can reference objects in the Application Context from Script, the Expression Builder, in the ODA layer, and so forth. The application context Map contains specific name value pairs that are passed to all generation and rendering processes.

You can use the application context to pass a security identifier that can be validated in the connection class and passed in as part of the SOAP request. In many cases the Web Services require such identifiers.

The Connection Class

If the connectionClass public property of the data source is set to a non-empty string, the run-time driver uses a custom connection class to create connections to the web service. The custom connection class is also responsible for executing the web services queries for Web Services data sets associated with this data source.

The connection class property is the fully qualified name of a Java class, which must implement the following class method to establish a web services connection. The Java class must be in the application class path.

```
public static Object connect(  
    java.util.Map connectionProperties,  
    java.util.Map appContext );
```

The connectionProperties parameter specifies the run-time values of all public connection properties available to the driver as a (String, String) map keyed by the connection property name. The map may contain any or all of the following map keys: soapEndPoint, connectionTimeOut, connectionClass, OdaConnProfileName, and OdaConnProfileStorePath.

The appContext parameter provides all the application context values as (String, Object) pairs. Its value is never null, but its collection may be empty.

The connection Instance

The connect(...) method of the custom driver class, after establishing a successful connection, returns a non-null object which implements the following two methods.

```
public Object executeQuery(  
    java.lang.String queryText,  
    java.util.Map parameterValues,  
    java.util.Map queryProperties  
);  
  
public void disconnect( );
```

The executeQuery() Method

The queryText parameter specifies the query text. It can be null if the connection class does not require the report design to provide a query text.

The parameterValues parameter specifies values of all the data set parameters as a (String, Object) map keyed by parameter name. It can be null if the data set does not define any parameters.

The queryProperties parameter specifies values of all the data set public properties as a (String, String) map keyed by property name. The map can contain the queryTimeOut map key.

The query method must return a value of either of the following data types:

- java.lang.String

- The returned String is the complete SOAP response.
- `java.io.InputStream`
The returned stream is SOAP response stream.

The disconnect() method

This method closes the connection. The driver implementation of the disconnect method is optional. If it is implemented, BIRT calls this method when the associated data source closes.

Method calling and error handling

The driver calls the custom driver class using Java reflection. The class and connection object do not need to implement any predefined interface. Any exception thrown by any of the defined methods is treated as error and results in a failure and an ODAException being thrown.

Custom connection class example

Listing 12-31 is an example of a custom class that accesses a set of connection properties, then instantiates a query object.

Listing 12-31 Custom Connection class

```
import java.util.Iterator;
import java.util.Map;

public class MyConnectionClass {
    public static Object connect( Map connProperties,
        Map appContext )
    {
        Iterator it = connProperties.keySet( ).iterator( );
        while ( it.hasNext( ) ) {
            Object key = it.next( );
        }

        it = connProperties.values( ).iterator( );
        while ( it.hasNext( ) ) {
            // Get value
            Object value = it.next( );
        }

        MyWSQuery msg = new MyWSQuery( );
        return msg;
    }
}
```

Listing 12-32 is an example of a query class implementation that executes a query by opening a file input stream and disconnects, closing the file.

Listing 12-32 Query class implementation

```
import java.io.FileInputStream;
import java.util.Map;

public class MyWSQuery {

    public Object executeQuery( String queryText,
        Map parameterValues, Map queryProperties )
    {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream( "c:/ExchangeRates.xml" );
        } catch ( Exception e ) {

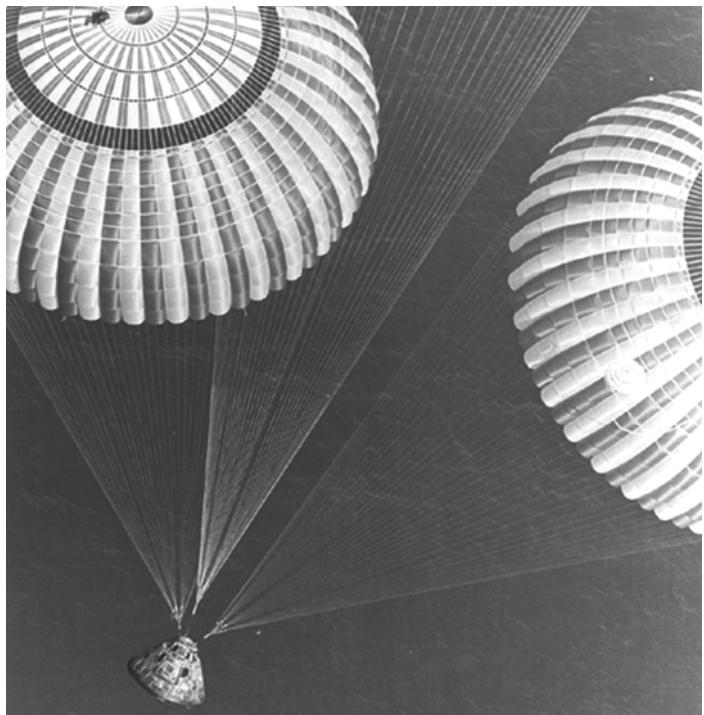
        }
        return fis;
    }

    public void disconnect( )
    {
        If( fis != null )
        {
            fis.close( );
            fis = null;
        }
    }
}
```

This page intentionally left blank

IV

Integrating BIRT Functionality into Applications



This page intentionally left blank

13

Understanding the BIRT APIs

The Eclipse BIRT code consists of many hundreds of Java classes and interfaces, but most of them are private, for use by contributors to the BIRT open source project. Developers of applications use the classes and interfaces that are in the public API. The public API consists of the classes and interfaces in the following package hierarchies:

- Report Engine API

The `org.eclipse.birt.report.engine.api` package hierarchy contains the API that a developer of a custom report generator uses. This API provides the most commonly used functionality for a reporting application. The key class in the Report Engine API is `ReportEngine`. This class provides access to all the tasks that create a report from a report design or a report document. The Report Engine API also includes the classes and packages that support the scripting capabilities of a report design.

- Design Engine API

The `org.eclipse.birt.report.model.api` package hierarchy is by far the larger of the two reporting APIs. This API provides access to the content and structure of a report design, a template, or a library. A reporting application can call this API to change the structure of a design. The Design Engine API is also the API that a developer of a custom report designer uses.

- Chart Engine API

The `org.eclipse.birt.chart` package hierarchy contains the API that a developer of a custom chart generator uses. A reporting application can also use this API in conjunction with the Report Engine and Design Engine APIs to create and modify chart elements in a report design.

- Extension APIs

BIRT also provides a set of extension APIs for creating custom report items, custom data sources and data sets, custom rendering formats, and custom charts. Part V, “Working with the Extension Framework” provides detailed examples of how to use these extensions.

For information about class and interface methods, see the API Javadoc, which you can access from the BIRT Report Designer main menu at Help→Help Contents→BIRT Programmer Reference→Reference→API Reference.

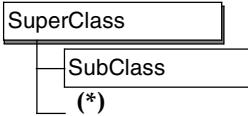
Package hierarchy diagrams

This chapter contains hierarchical diagrams for the packages in the BIRT APIs. These diagrams show the hierarchy of the classes in the package and interfaces local to the package or implemented by classes in the package. Classes and interfaces preceded by a package name are not local to the package. In these hierarchical diagrams, the graphics shown in Table 13-1 indicate different attributes and relationships of the classes and interfaces.

Table 13-1 Conventions for the hierarchy diagrams

Item	Symbol
Abstract class	
Class that has one or more subclasses	
Class that has no subclasses	
Final class	
Interface	
Solid lines indicate a superclass-subclass relationship	
Broken lines indicate an implementation relationship	

Table 13-1 Conventions for the hierarchy diagrams (*continued*)

Item	Symbol
An asterisk indicates too many subclasses to list	

About the BIRT Report Engine API

The BIRT Report Engine supports report generation and rendering in several different environments, such as:

- Stand-alone engine
A Java developer uses a stand-alone engine to render a BIRT report from an existing report design (.rptdesign) file. In this environment, the Java developer creates a command line application to write a complete report to disk, in HTML, Adobe PDF, Adobe PostScript (PS), Microsoft Excel (XLS), Microsoft PowerPoint (PPT), and Microsoft Word (DOC) formats.
- BIRT report viewer
BIRT Report Designer uses the BIRT report viewer to view a report as paginated HTML. The BIRT report viewer is a web application that runs in the Tomcat Application Server, which is embedded in Eclipse. This viewer contains an embedded report engine.
- Custom report designer with an embedded engine
A custom desktop reporting application integrates the BIRT Report Engine for the purpose of previewing the report.
- Web application that embeds the engine
A web application similar to the BIRT report viewer can use the BIRT Report Engine to generate a web-based report.

The BIRT Report Engine supports running and rendering reports in these diverse environments. It does not perform environment-dependent processing such as URL construction, image storage, and design file caching. The reporting application that uses the engine API must provide such context information to the engine.

The BIRT Report Engine API consists of a set of interfaces and implementation classes. The BIRT Report Engine API supports integrating the run-time part of BIRT into your application. The API provides a set of task classes that support the following operations:

- Discovering the set of parameters defined for a report
- Getting the default values for parameters

- Running a report design to produce an unformatted report document
- Running a report design or report document to produce any of the supported output formats
- Extracting data from a report document

Creating the BIRT ReportEngine instance

Each application, whether it is stand-alone or web-based, only needs to create one ReportEngine instance. Since the BIRT Report Engine is thread-safe, the single-instance recommendation is not a restriction.

Create the ReportEngine instance with a constructor that takes an EngineConfig object as an argument. If the configuration object is null, a default engine configuration is used. At termination, the application must call `destroy()` to unload extensions and delete temporary files.

Using the BIRT Report Engine API

The BIRT Report Engine API supports the following key tasks to generate reports:

- Setting options for the report engine using an EngineConfig object
- Starting the Platform
- Creating a ReportEngine object using the ReportEngineFactory
- Opening an existing report design using one of the `openReportDesign()` methods of ReportEngine or opening an existing report document using the `openReportDocument()` method
- Optionally, when design details of the report parameters are required, using an IGetParameterDefinitionTask object to obtain this information
- Running and rendering a report using IRunAndRenderTask or IRunTask followed by IRenderTask
- Cleaning up the report engine by calling `destroy()` on the engine instance

A few primary classes and interfaces provide the core functionality of the BIRT Report Engine. The following sections provide an overview of these classes.

EngineConfig class

The EngineConfig class wraps configuration settings for a report engine. Use the EngineConfig object to set global options for the environment of the report engine, including:

- Specifying the BIRT home, the location of the engine plug-ins and Java archive (.jar) files
- Setting OSGi arguments

- Setting the Platform context
- Setting resource locations
- Adding application-wide scriptable objects
- Setting the directory where the report engine writes temporary files
- Managing logging

ReportEngine class

The ReportEngine class represents the BIRT Report Engine. You instantiate the ReportEngine object by using a factory method that takes an argument of an EngineConfig object. If the configuration object is null, the environment must provide a BIRT_HOME variable that specifies the BIRT home. You use a ReportEngine object to perform the following tasks:

- Getting the configuration object
- Opening a report design or a report document
- Creating an engine task to get parameter definitions
- Creating an engine task to access the data from a report item
- Getting supported report formats and MIME types
- Creating an engine task to run a report or render a report to an output format
- Creating an engine task to extract data from a report document
- Changing the logging configuration
- Cleaning up and destroying the engine

IReportRunnable interface

To work with the report design with the engine, you must load the design using one of the openReportDesign() methods in the ReportEngine class. These methods return an IReportRunnable instance that represents the engine's view of the report design. You use an IReportRunnable object to perform the following tasks:

- Getting standard report design properties such as the report title and report author
- Getting any images embedded within the report design
- Getting a handle to the report design

IReportDocument interface

To use a report document, you must load the document using one of the ReportEngine.openReportDocument() methods. These methods return an IReportDocument instance. You use an IReportDocument object to render a

report to a supported output format with an IRenderTask object. You can use the table of contents markers in the IReportDocument to determine the pages to render. The IReportDocument interface also supports retrieving page counts, parameter values used while creating the report document, and bookmarks.

IEngineTask interface

The IEngineTask interface provides the framework for the tasks that the report engine performs. The IEngineTask interface manages the scripting context, getting and setting parameter values, setting the report's locale, getting the current status of a task, and cancelling a task. The other task interfaces extend IEngineTask.

IGetParameterDefinitionTask interface

The IGetParameterDefinitionTask interface extends IEngineTask to provide access to information about parameters. The engine factory method to create an IGetParameterDefinitionTask object takes an IReportRunnable argument. Parameter definitions provide access to:

- Information that BIRT Report Designer specified at design time
- Static or dynamic selection lists
- User-supplied values
- The grouping structure of the parameters
- Custom XML
- User-defined properties

IDataExtractionTask interface

The IDataExtractionTask interface extends IEngineTask to provide access to the data stored in an IReportDocument object. You can use an IDataExtractionTask object to examine the metadata for a set of data rows. Using the metadata, you can select a set of columns to extract, sort, or filter. This interface can extract the data from:

- The whole report document
- A single report item
- A single instance of a report item

IRunTask interface

The IRunTask interface provides the methods to run a report design. This task saves the result as a report document (.rptdocument) file to disk.

An IRunTask object takes parameter values as a HashMap. Call the validateParameters() method to validate the parameter values before you run the report.

IRenderTask interface

The IRenderTask interface provides the methods to render a report document to one of the supported output formats. This task can save the report to a file on disk or to a stream.

You can set options for rendering using the RenderOption class. You can set specific options for HTML and PDF using the HTMLRenderOption and PDFRenderOption classes respectively, which are subclasses of the RenderOption class. Pass the appropriate render option object to the IRenderTask object before rendering your report.

IRunAndRenderTask interface

The IRunAndRenderTask interface provides the methods to run a report and render it in one of the supported output formats. This task can save the report to disk or to a stream. No intermediate report document (.rptdocument) file is created.

An IRunAndRenderTask object takes parameter values as a HashMap or individually. Call the validateParameters() method to validate the parameter values before you run the report.

You can set the same rendering options on an IRunAndRenderTask object as on the IRenderTask object. Pass the appropriate render option object to the IRunAndRenderTask object before running the report.

Report engine class hierarchy

The class hierarchy in Figure 13-1 illustrates the organization of the classes within the report engine package. Unless otherwise specified, all classes and interfaces in this diagram are in the org.eclipse.birt.report.engine.api package.

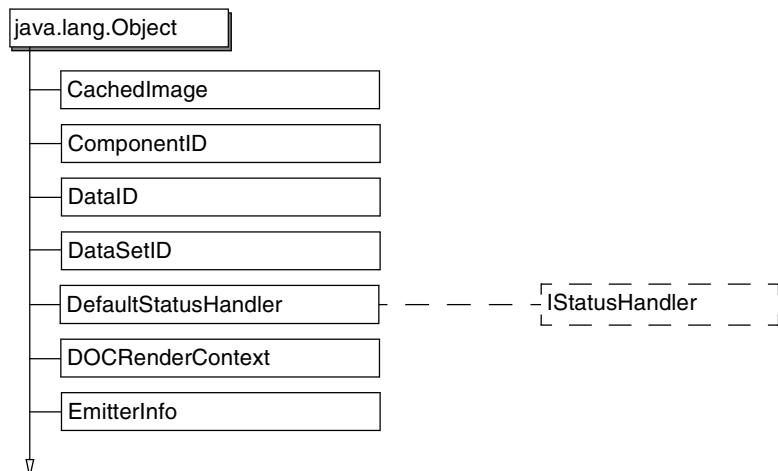


Figure 13-1 Classes within the report engine package (*continues*)

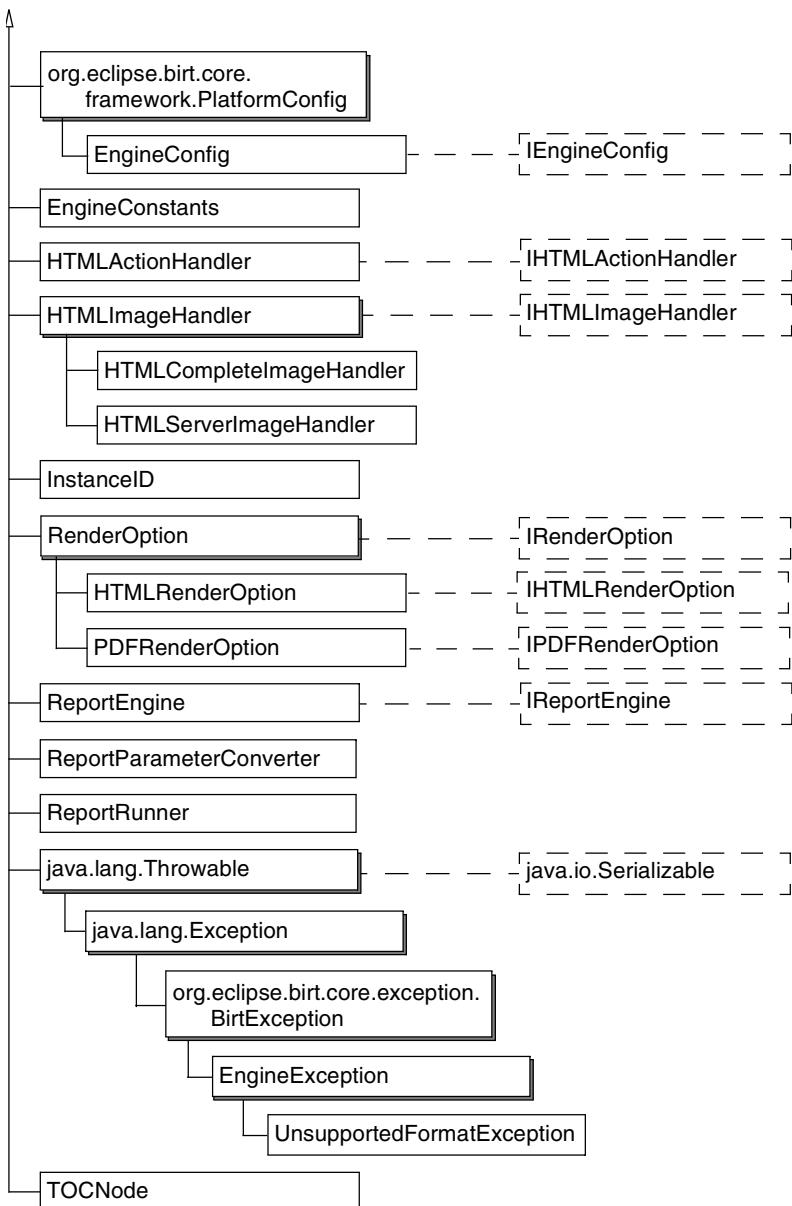


Figure 13-1 Classes within the report engine package (*continued*)

Report engine interface hierarchy

Figure 13-2 contains the interface hierarchy for the Report Engine API. All interfaces in this diagram are in the `org.eclipse.birt.report.engine.api` package.

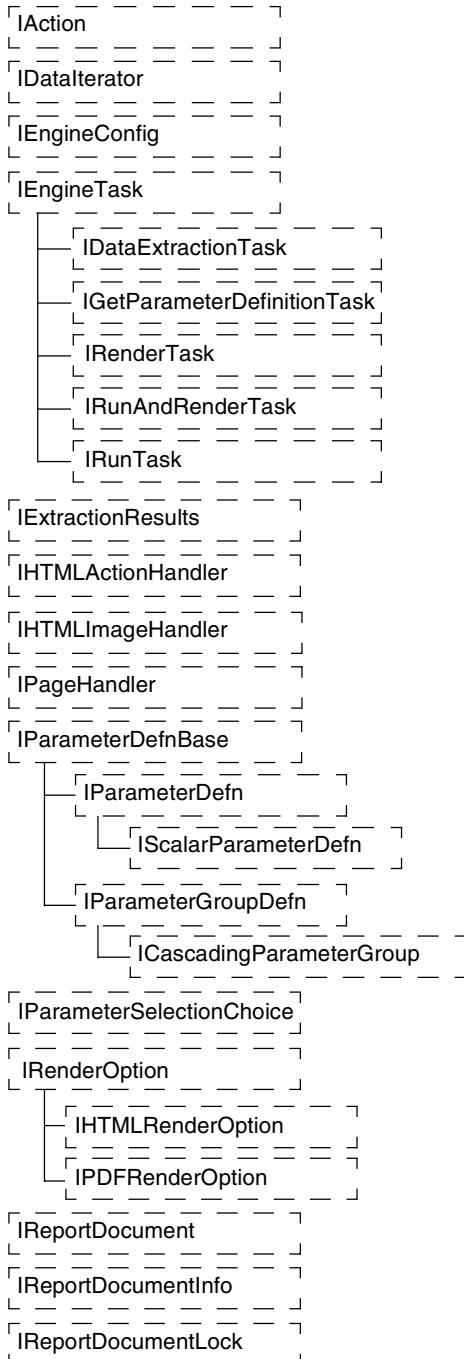


Figure 13-2 Interface hierarchy for the report engine package (*continues*)

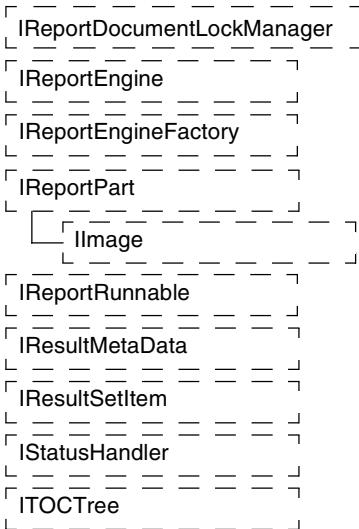


Figure 13-2 Interface hierarchy for the report engine package (*continued*)

About the Design Engine API

The Design Engine API is also known as the report model API. The Design Engine API is the API that a tool writer uses to build a design tool. The Design Engine API contains classes and methods to create, access, and validate a report design.

The `org.eclipse.birt.report.model.api` package contains the interfaces and classes that the tool writer uses to access the design model objects. Through the Design Engine API, you can do the following tasks:

- Reading and writing design files
- Maintaining the command history for undo and redo
- Providing a rich semantic representation of the report design
- Providing metadata about the ROM
- Performing property value validation
- Notifying the application when the model changes

Using the BIRT Design Engine API

The purpose of the BIRT Design Engine API is to modify or create a report design file that the BIRT report engine can use to generate a report. BIRT Report Designer, for example, uses the BIRT Design Engine API for this purpose. A custom report design tool, written for the same general purpose as BIRT Report Designer, can also use the BIRT Design Engine API to

generate a design file. The Design Engine API also supports libraries and templates in the same way as report designs. The BIRT Design Engine API does not include any user interface classes. A custom report design tool must provide its own user interface code.

With the design engine, you can create or modify a BIRT report design by performing the following tasks:

- Setting options for the design engine by using a `DesignConfig` object
- Starting the Platform, if not already started
- Creating a `DesignEngine` object using the `DesignEngineFactory`
- Beginning a user session by using the `DesignEngine.newSessionHandle()` method to instantiate a `SessionHandle` object
- Setting session parameters and loading the property definitions of the report elements by using the `SessionHandle` object to create an instance of the `ReportDesignHandle` class
- Using the `ReportDesignHandle` to create an `ElementFactory`, which can create report elements
- Using the `ReportDesignHandle` to add new elements to the report design or modify existing elements
- Saving the report design file by using the `ReportDesignHandle`

The following sections describe the primary classes of the BIRT Design Engine API.

DesignConfig class

The `DesignConfig` class wraps configuration settings for a design engine. Use a `DesignConfig` object to set global options for the design engine, including:

- Specifying the location of engine plug-ins and Java archive (.jar) files
- Setting the Platform Context
- Specifying configuration variable
- Set OSGi arguments

DesignEngine class

The `DesignEngine` class represents the BIRT design engine. You create the `DesignEngine` with a factory method that takes a `DesignConfig` object. If the configuration object is null, the environment must provide the path to the BIRT home, the directory that contains the engine plug-ins and JAR files. The `DesignEngine` class is the gateway to creating the other objects you need to build a report design tool. Use the methods of the `DesignEngine` class to create a locale-specific `SessionHandle` object. Use the `newSession()` method to perform this task. The `SessionHandle` provides a gateway to report design objects.

SessionHandle class

The SessionHandle class represents the user session. A SessionHandle object provides access to the set of open designs. A session has a set of default values for style properties and a default unit of measure. The session also has methods to create and open report designs, templates, and libraries, as well as setting the path and algorithm used to locate resources. The methods to create or open a report design return a ReportDesignHandle object.

ModuleHandle class

ModuleHandle provides access to the common structure and functionality of report designs, templates, and libraries. It is the parent class of the ReportDesignHandle and LibraryHandle classes. ModuleHandle provides access to the generic properties, such as author and comments. You also use ModuleHandle for many tasks on the file, including:

- Saving the module to a file
- Accessing the command stack for undo and redo
- Navigating to the various parts of the module
- Retrieving the module location
- Getting configuration variables

The ModuleHandle also has methods to get handles to the individual report items and all the other elements in a report design, template, or library. These elements and supporting components include:

- Report items. These elements are visual report elements such as tables, grids, images, and text elements.
- Code modules. These modules are global scripts that apply to the file as a whole.
- Parameters.
- Data sources, data sets, and cubes.
- Color Palette. This component is a set of custom color names.
- CSS files that the module uses.
- Theme. The theme is a group of styles that the module uses for formatting report elements.
- Master page. This element defines the layout of pages in paginated report output.
- Libraries. Any module can use one or more libraries to provide predefined elements.
- Resources. External files provide lists of messages in localized forms.
- Embedded images.

ReportDesignHandle class

ReportDesignHandle provides access to the report design-specific properties such as the scripts that execute when generating or rendering a report. This class also provides access to properties that templates use, such as the cheat sheet, display name, and icon file.

ReportDesignHandle is a subclass of ModuleHandle, so supports all that class's functionality. You also use ReportDesignHandle to get handles to the individual report items and for many report design-specific tasks, including:

- Navigating to the various parts of the design
- Setting the event-specific scripts that execute when the report engine runs and renders the report

The ReportDesignHandle also has methods to gain access to the following report components:

- Styles, the list of user-defined styles for formatting report elements
- Base directory, the location of file system resources with relative paths
- Body, a list of the report sections and report items in the design
- Scratch Pad, a temporary place to hold report items while restructuring a report

LibraryHandle class

LibraryHandle is a subclass of ModuleHandle, so supports all that class's functionality. LibraryHandle also provides access to the following library-specific properties:

- Name space, the name that a module including a library uses to identify the elements that the library defines
- The set of themes that the library defines
- Imported CSS styles used by themes

DesignElementHandle class

The DesignElementHandle class is the base class for all report elements, both visual report item elements and non-visual ones, such as data sets and cubes. DesignElementHandle provides generic services for all elements, such as:

- Adding a report item to a slot
- Registering a change event listener
- Getting and setting properties, names, and styles
- Getting available choices for specific properties
- Dropping an element from the design
- Copying, pasting and moving report items

Individual element handle classes

The individual element handle classes derive from ReportElementHandle. Each report element has its own handle class. To work with operations unique to a given report element, you cast the ReportElementHandle to the appropriate subclass for the element. For example, the CellHandle class has methods such as getColumn(), and the DataSourceHandle class has methods such as setBeforeOpen().

Design engine class hierarchy

Figure 13-3 illustrates the hierarchy of the classes within the design engine package. Unless otherwise specified, all classes and interfaces in this diagram are in the org.eclipse.birt.report.model.api package.

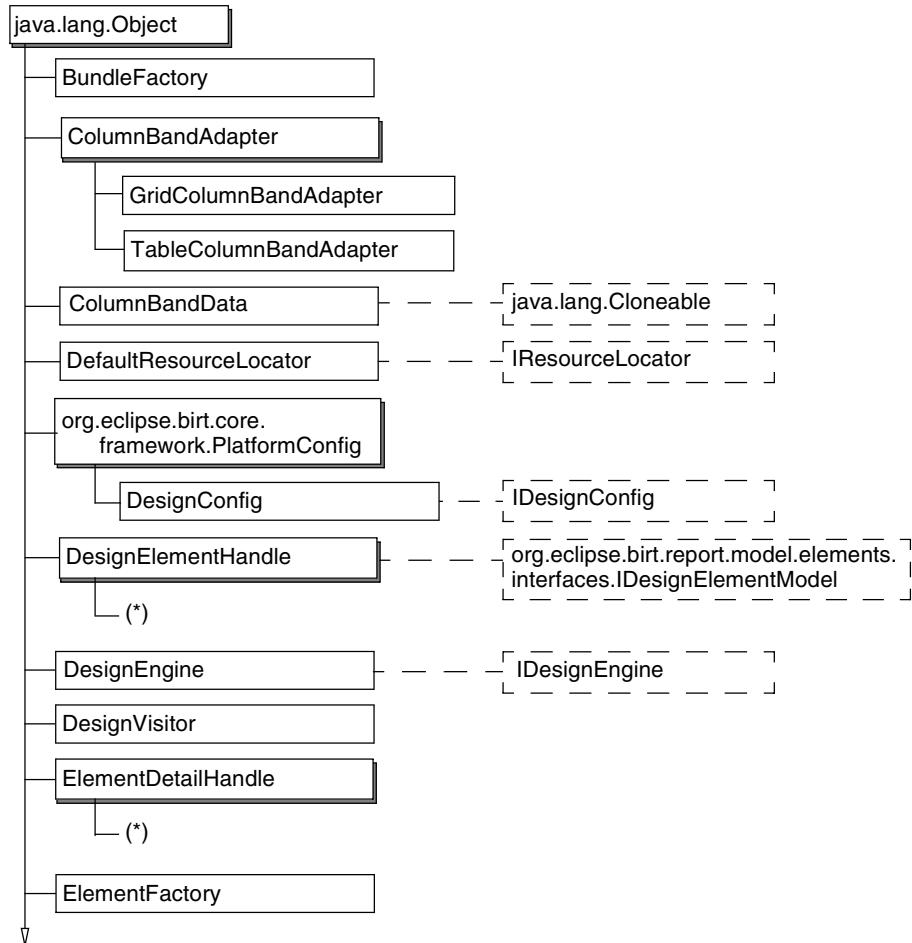


Figure 13-3 Classes within the report model package

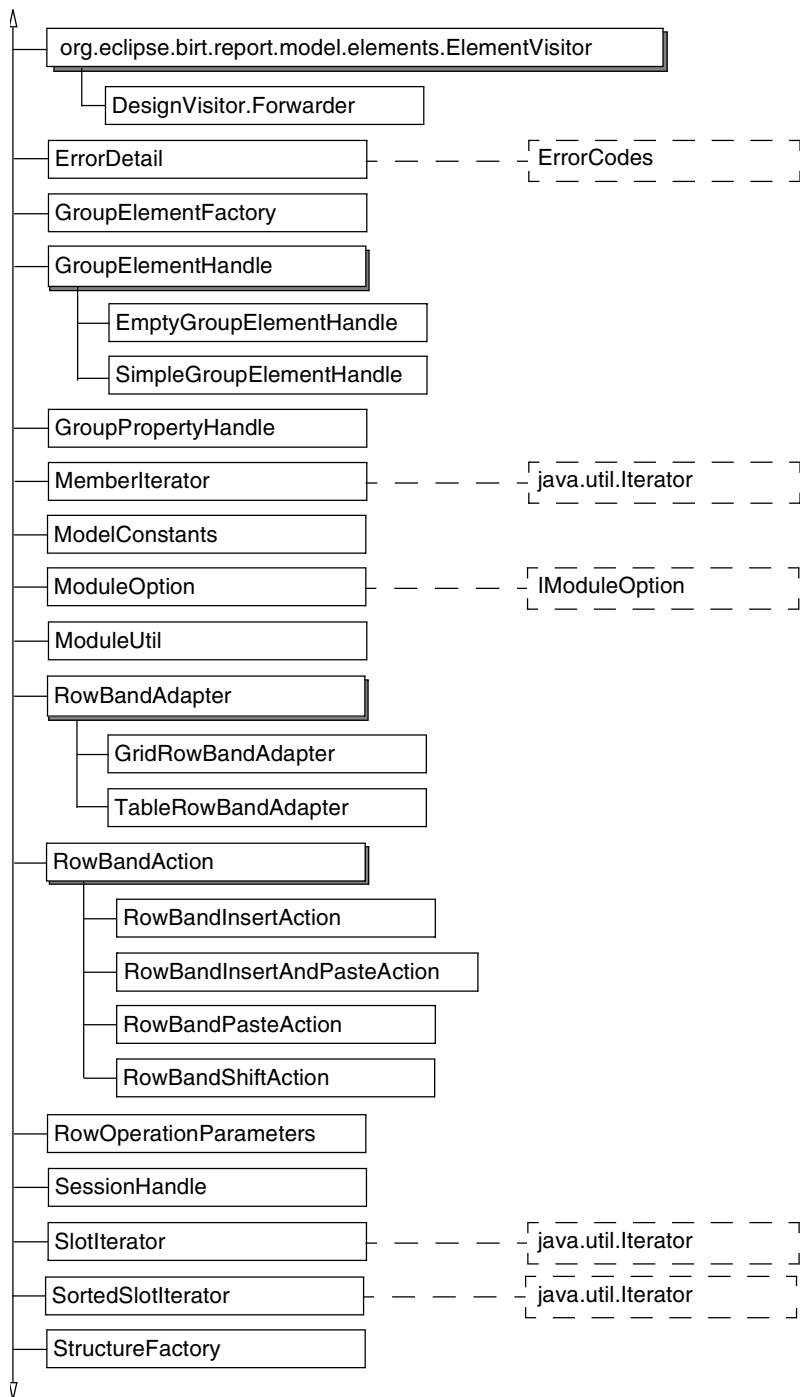


Figure 13-3 Classes within the report model package (*continues*)

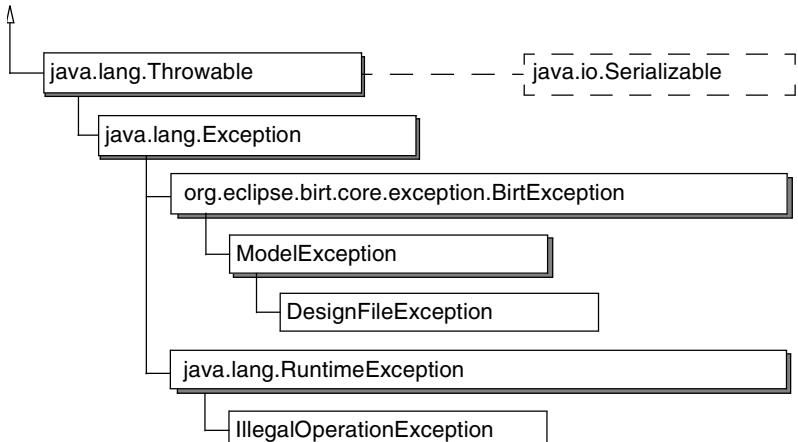


Figure 13-3 Classes within the report model package (*continued*)

DesignElementHandle hierarchy

Figure 13-4 contains the class hierarchy for DesignElementHandle in the org.eclipse.birt.report.model.api package and the classes that derive from it.

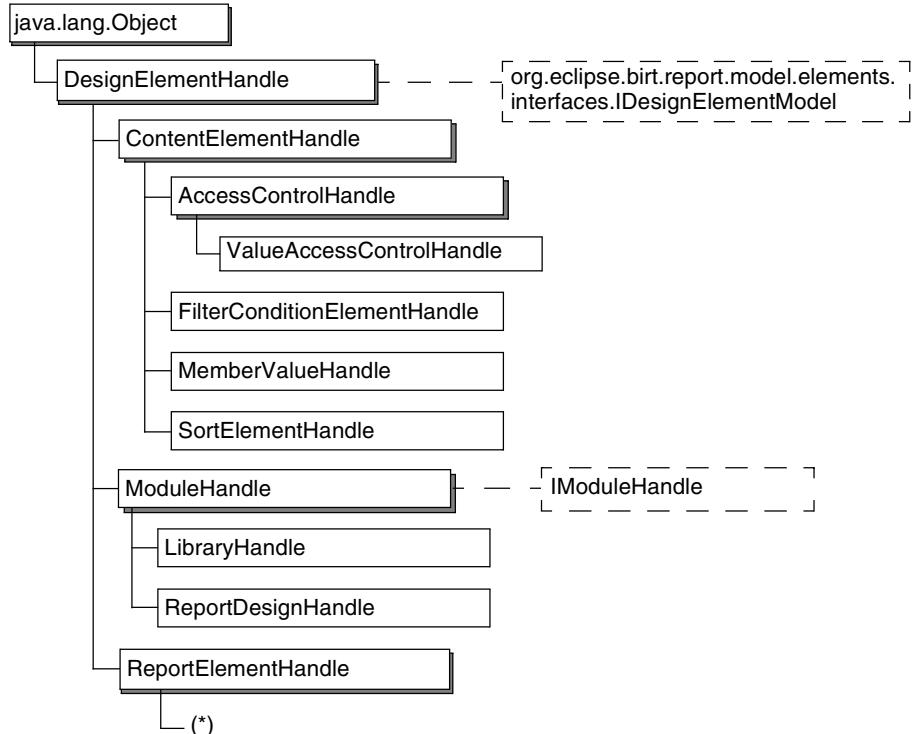


Figure 13-4 DesignElementHandle class hierarchy

ReportElementHandle hierarchy

Figure 13-5 contains the class hierarchy for ReportElementHandle in the org.eclipse.birt.report.model.api package and the classes that derive from it. The interfaces that the classes implement are all in the org.eclipse.birt.report.model.elements.interfaces and org.eclipse.birt.report.model.elements packages. Classes with names that have a prefix of olap are in the org.eclipse.birt.report.model.olap package.

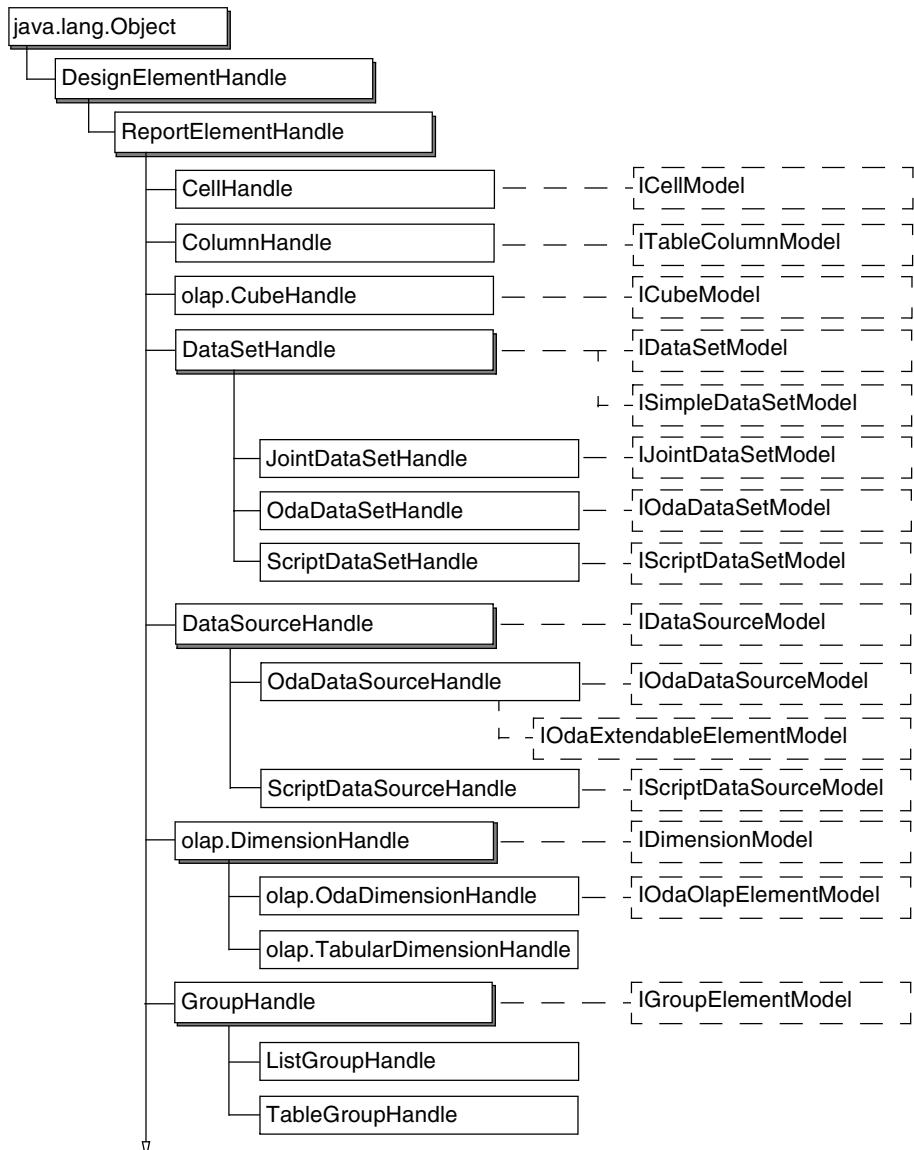


Figure 13-5 ReportElementHandle class hierarchy (*continues*)

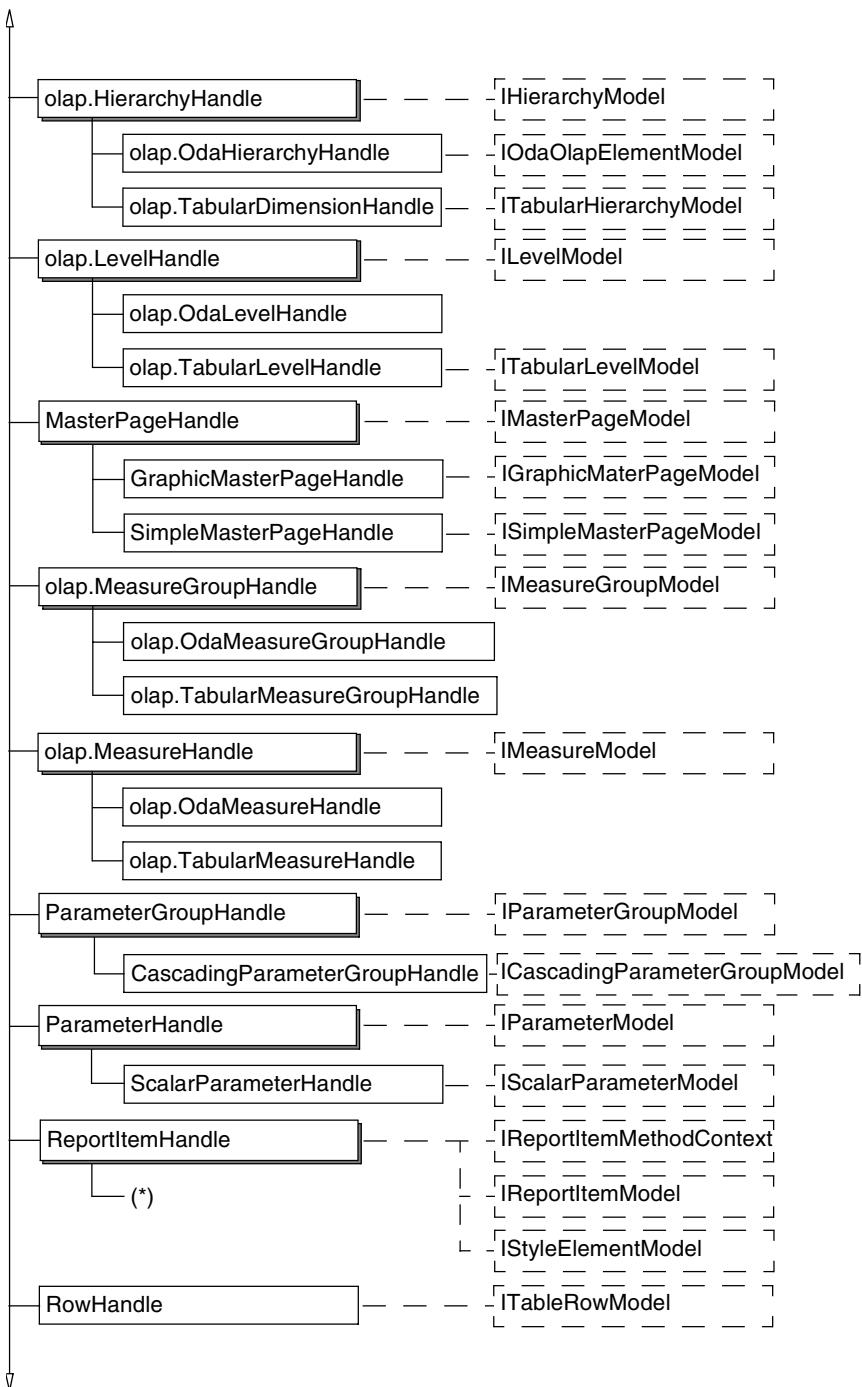


Figure 13-5 ReportElementHandle class hierarchy

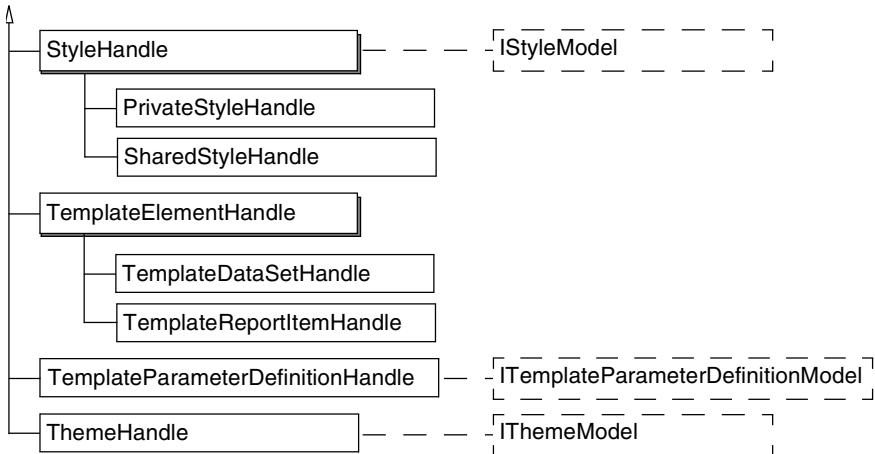


Figure 13-5 ReportElementHandle class hierarchy (*continued*)

ReportItemHandle hierarchy

Figure 13-6 contains the class hierarchy for ReportItemHandle in the org.eclipse.birt.report.model.api package and the classes that derive from it. The interfaces that the classes implement are all in the org.eclipse.birt.report.model.elements.interfaces and org.eclipse.birt.report.model.elements packages.

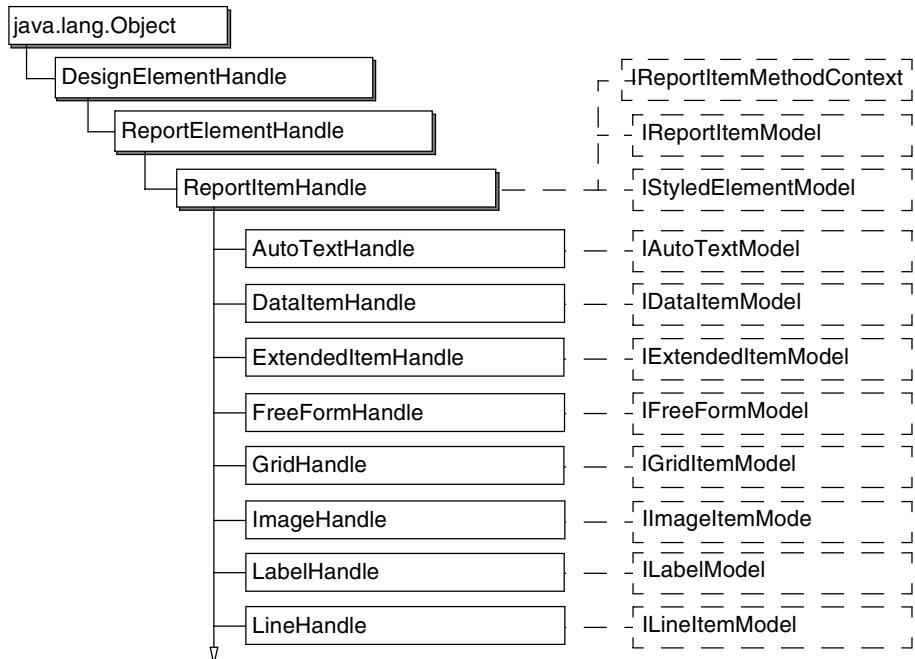


Figure 13-6 ReportItemHandle class hierarchy (*continues*)

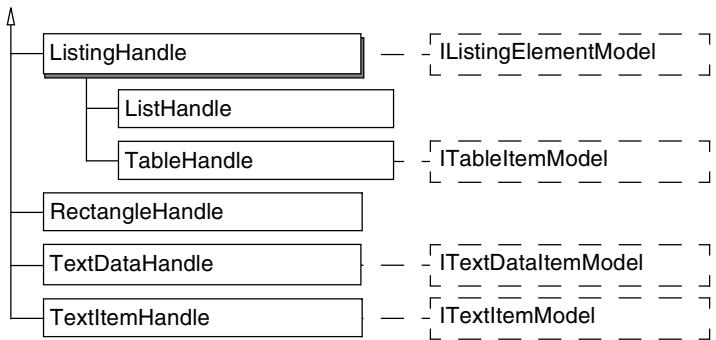


Figure 13-6 ReportItemHandle class hierarchy (*continued*)

ElementDetailHandle hierarchy

Figure 13-7 contains the class hierarchy for ElementDetailHandle in the org.eclipse.birt.report.model.api package and the classes that derive from it.

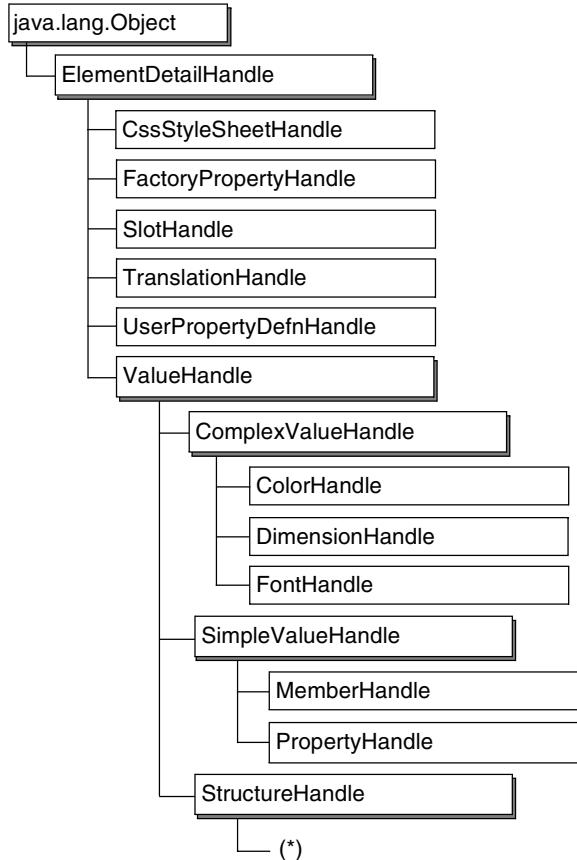


Figure 13-7 ElementDetailHandle class hierarchy

StructureHandle hierarchy

Figure 13-8 contains the class hierarchy for StructureHandle in the org.eclipse.birt.report.model.api package and the classes that derive from it.

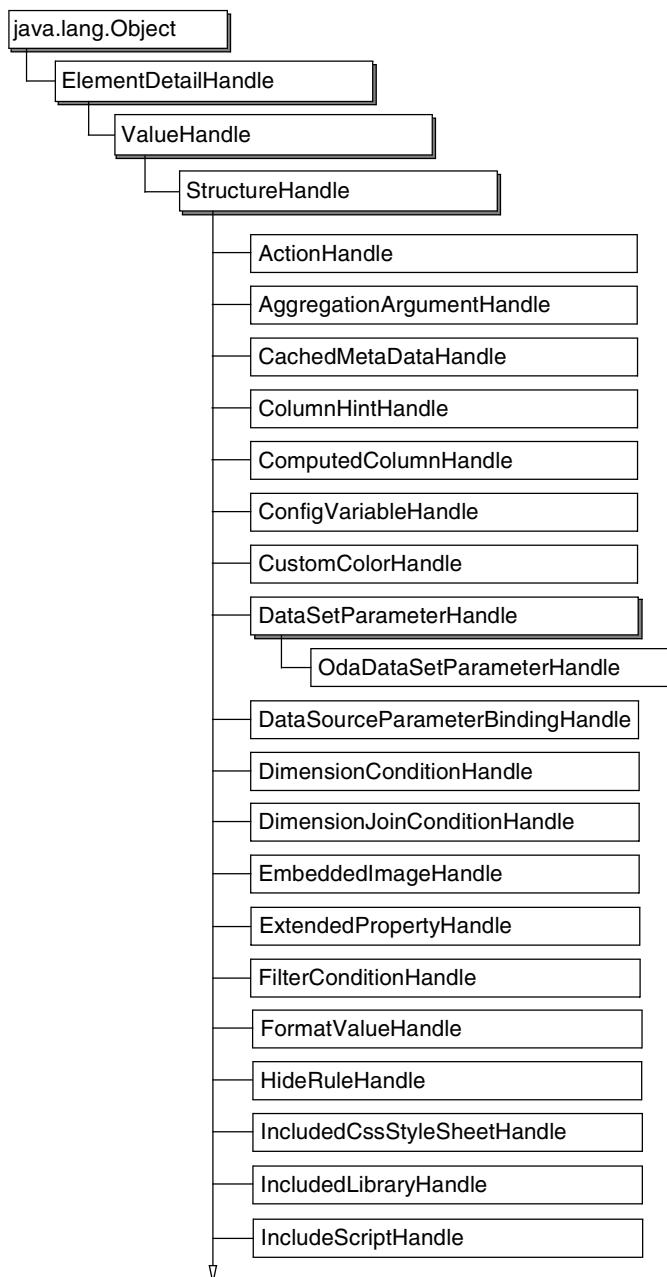


Figure 13-8 StructureHandle class hierarchy (*continues*)

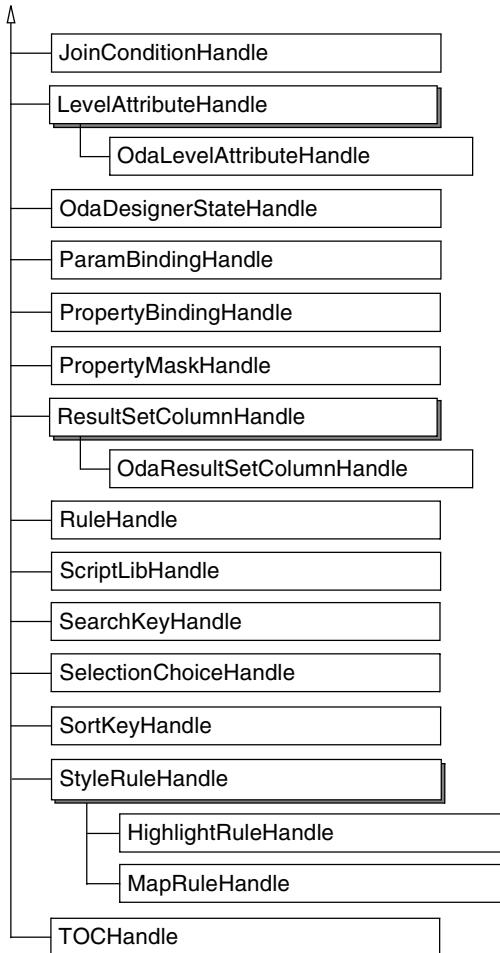


Figure 13-8 StructureHandle class hierarchy (continued)

Design engine interface hierarchy

Figure 13-9 contains the interface hierarchy for the Design Engine API. All interfaces in this diagram are in the org.eclipse.birt.report.model.api package.

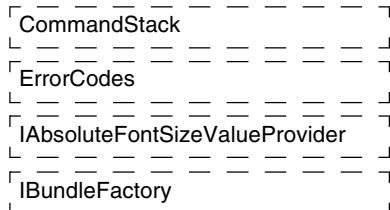


Figure 13-9 Interface hierarchy for the design engine package

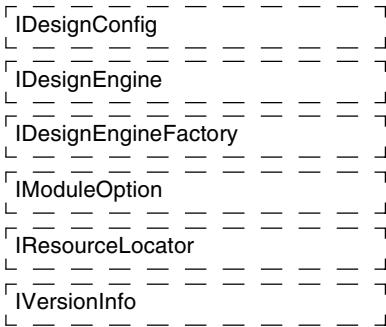


Figure 13-9 Interface hierarchy for the design engine package (continued)

About the BIRT Chart Engine API

The chart engine API is based upon the Eclipse Modeling Framework (EMF) as a structured data model. Use the classes and interfaces in the chart engine API to modify chart objects within a BIRT reporting application or in a stand-alone charting application.

The Chart Engine API includes many packages in the org.eclipse.birt.chart hierarchy. The model.* packages contain the core chart model interfaces and enumeration classes generated using EMF. The model.*.impl packages contain the core chart model implementation classes generated using EMF. All other packages are dependencies from and indirect references to the core model.

There is a one-to-one correspondence between the classes in the impl packages and the interfaces in corresponding model packages. The classes in the impl packages implement the methods in the interfaces of the corresponding model classes. The impl classes also contain factory methods that you use to create an instance of a class.

Using the BIRT Chart Engine API

Although there are over 500 classes and interfaces in the BIRT Chart Engine API, most of functionality for creating or modifying a chart is concentrated in a small subset of classes and interfaces. The primary interface in the BIRT Chart Engine API is the Chart interface. An object of the Chart type is called the chart instance object. The Chart interface has two subinterfaces, ChartWithAxes and ChartWithoutAxes. DialChart is a third interface that inherits from ChartWithoutAxes. ChartWithoutAxes defines a pie chart and DialChart defines a meter chart. ChartWithAxes defines all other chart types.

You create a chart instance object with the create() method of either ChartWithAxesImpl, ChartWithoutAxesImpl, or DialChartImpl, as in the following statement:

```
ChartWithAxes myChart = ChartWithAxesImpl.create();
```

You set the basic properties of a chart, such as its orientation and dimensionality with setter methods of the Chart interface, such as:

```
myChart.setOrientation( Orientation.VERTICAL_LITERAL );
myChart.setDimension( ChartDimension.THREE_DIMENSIONAL );
```

You set the more complex properties of a chart, like the characteristics of the chart's axes and series by getting an instance of the object you want to modify and then setting its properties. For example, to set the caption of a chart's x-axis, you can use the following code:

```
Axis xAxis = myChart.getPrimaryBaseAxes( )[0];
xAxis.getTitle( ).setCaption( "Months" );
```

Although charts are often identified by type, such as a pie chart or a line chart, a chart with multiple series of differing types cannot be classified as one type. A series, on the other hand, has a specific type. With the BIRT Chart Engine API, you can create a specific type of series by using the create() method of one of the SeriesImpl subclasses. For example, the following code creates a bar series:

```
BarSeries barSeries1 = ( BarSeries ) BarSeriesImpl.create( );
```

Chart engine class hierarchy

The diagrams that follow contain hierarchies for the following chart engine packages:

- org.eclipse.birt.chart.aggregate
- org.eclipse.birt.chart.datafeed
- org.eclipse.birt.chart.device
- org.eclipse.birt.chart.event
- org.eclipse.birt.chart.exception
- org.eclipse.birt.chart.factory
- org.eclipse.birt.chart.log
- org.eclipse.birt.chart.model
- org.eclipse.birt.chart.model.attribute
- org.eclipse.birt.chart.model.component
- org.eclipse.birt.chart.model.data
- org.eclipse.birt.chart.model.layout
- org.eclipse.birt.chart.model.type
- org.eclipse.birt.chart.render
- org.eclipse.birt.chart.script
- org.eclipse.birt.chart.util

The hierarchy diagrams for the org.eclipse.birt.chart.model.*.impl packages are not included because they are simply implementations of the interfaces in the corresponding org.eclipse.birt.chart.model.* packages. The model packages, with two exceptions, contain only interfaces.

The first exception is org.eclipse.birt.chart.model, which has one class, ScriptHandler. The second exception is org.eclipse.birt.chart.model.attribute, which has a set of enumeration classes, one for each attribute. Each of the enumeration classes contains only a list of valid values for its attribute.

chart.aggregate class and interface hierarchy

The chart.aggregate package contains the class and interface that support the aggregate functions that produce the values that a chart shows. Figure 13-10 contains the class and interface hierarchy for org.eclipse.birt.chart.aggregate.

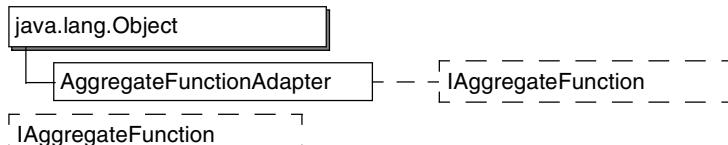


Figure 13-10 Classes and interfaces in org.eclipse.birt.chart.aggregate

chart.datafeed class and interface hierarchy

The chart.datafeed package contains the class and interfaces that support defining a custom data set for a chart. Figure 13-11 contains the class and interface hierarchy for org.eclipse.birt.chart.datafeed.

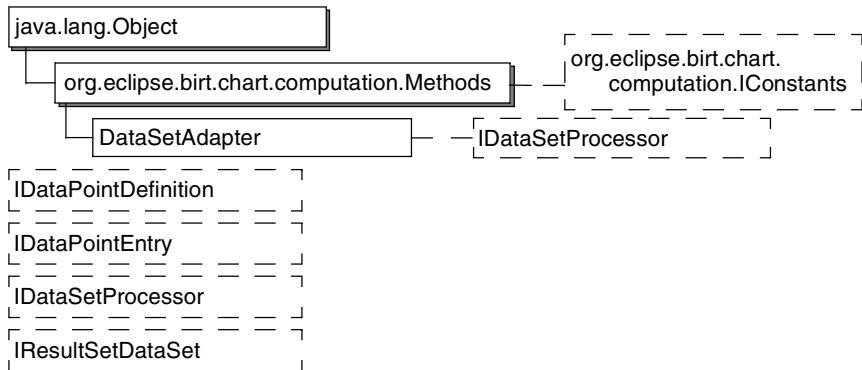


Figure 13-11 Classes and interfaces in org.eclipse.birt.chart.datafeed

chart.device class and interface hierarchy

The chart.device package contains the classes and interfaces that support rendering a chart to a specific output type. Device adapter classes provide the rendered output. These classes process the events defined in the chart.event package. Display adapter classes provide metrics for the output device. Figure 13-12 contains the class hierarchy for org.eclipse.birt.chart.device.

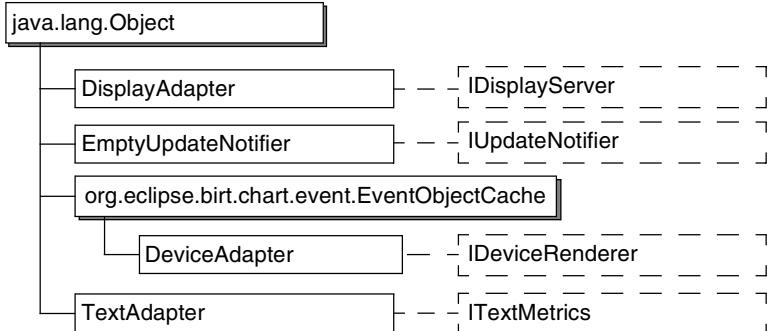


Figure 13-12 Classes in `org.eclipse.birt.chart.device`

Figure 13-13 contains the interface hierarchy for `org.eclipse.birt.chart.device`.

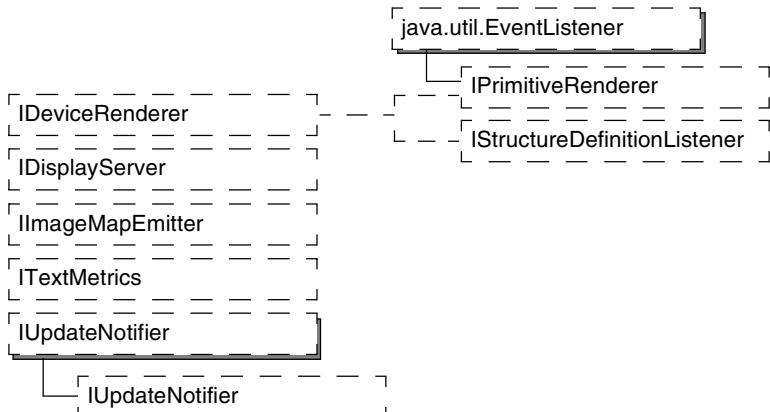


Figure 13-13 Interfaces in `org.eclipse.birt.chart.device`

chart.event class and interface hierarchy

The `chart.event` package contains the set of events that a rendering device that extends the `chart.device.DeviceAdapter` class can support. It also provides structural and caching classes for events. Figure 13-14 contains the interface hierarchy for `org.eclipse.birt.chart.event`.

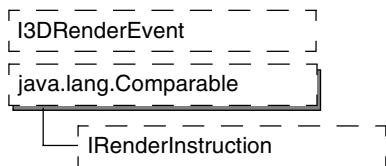


Figure 13-14 Interfaces in `org.eclipse.birt.chart.event`

Figure 13-15 contains the class hierarchy for `org.eclipse.birt.chart.event`.

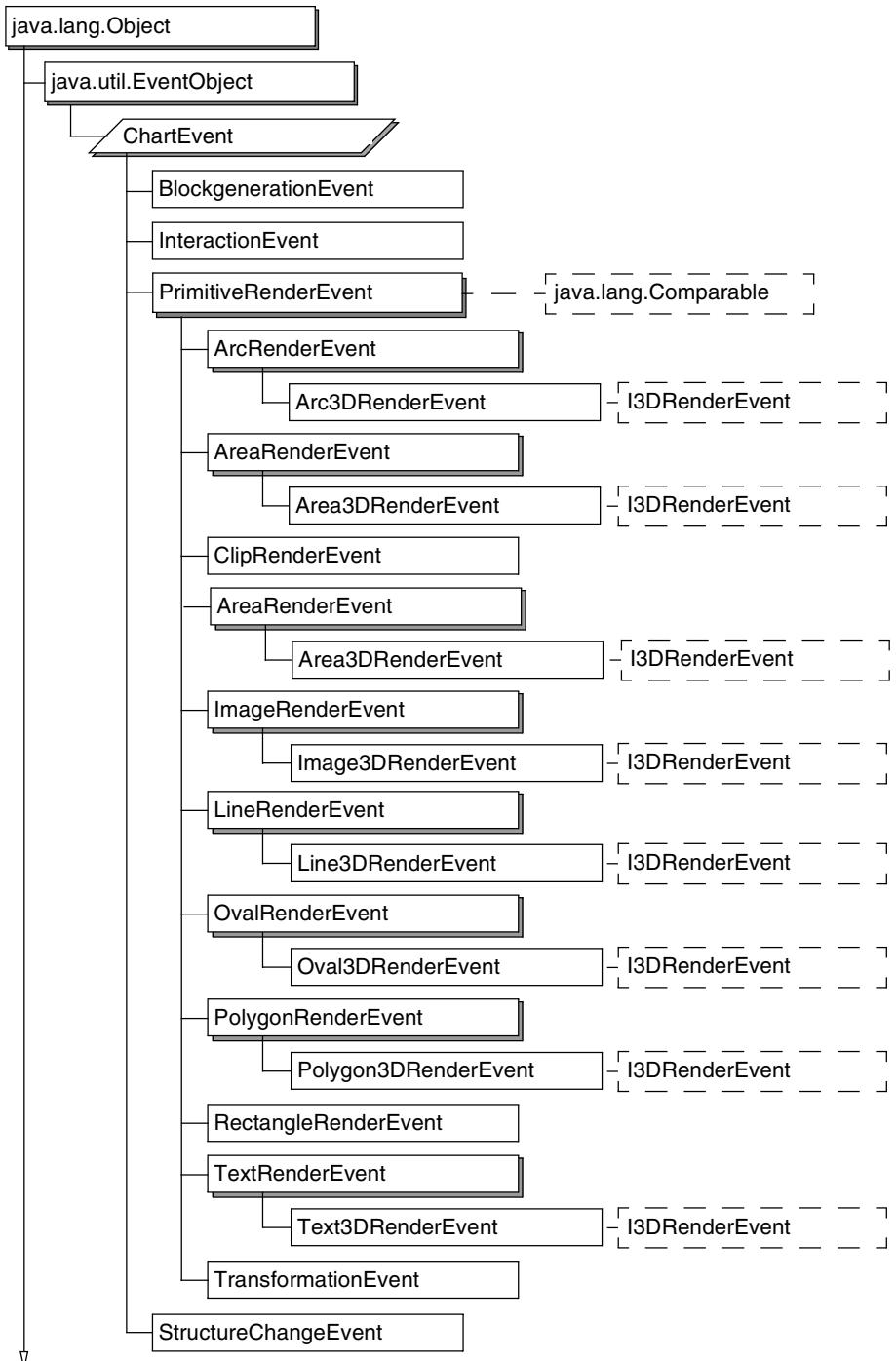


Figure 13-15 Classes in `org.eclipse.birt.chart.event` package (*continues*)

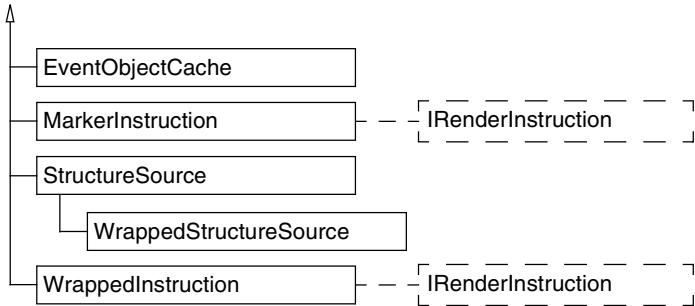


Figure 13-15 Classes in `org.eclipse.birt.chart.event` package (*continued*)

chart.exception class hierarchy

The `chart.exception` package contains the single exception class the BIRT Chart Engine defines. Figure 13-16 contains the class hierarchy for `org.eclipse.birt.chart.exception`.

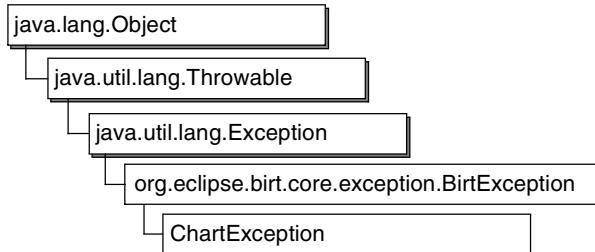


Figure 13-16 Class in `org.eclipse.birt.chart.exception`

chart.factory class and interface hierarchy

The `chart.factory` package contains classes and interfaces that build and generate a chart. It also contains a context class that provides information about the environment in which the factory runs, such as the locale and associated localization information. Figure 13-17 contains the class hierarchy for `org.eclipse.birt.chart.factory`.

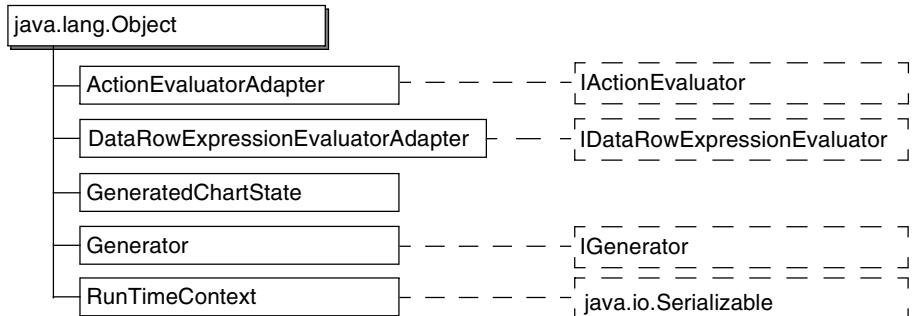


Figure 13-17 Classes in `org.eclipse.birt.chart.factory`

Figure 13-18 contains the class hierarchy for org.eclipse.birt.chart.factory.

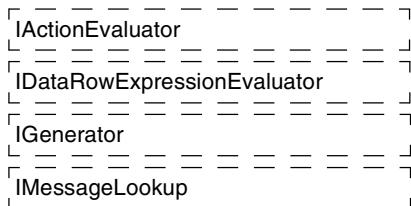


Figure 13-18 Interfaces in org.eclipse.birt.chart.factory

chart.log class and interface hierarchy

The chart.log package contains a single class and an interface to manage logging of the tasks that the BIRT Chart Engine performs. Figure 13-19 contains the class hierarchy for org.eclipse.birt.chart.log.

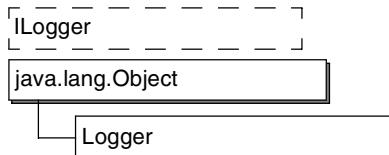


Figure 13-19 Class and interface in org.eclipse.birt.chart.log

chart.model interface hierarchy

The chart.model package contains the interfaces that describe the available chart types and the factory for generating charts. Figure 13-20 contains the interface hierarchy for org.eclipse.birt.chart.model.

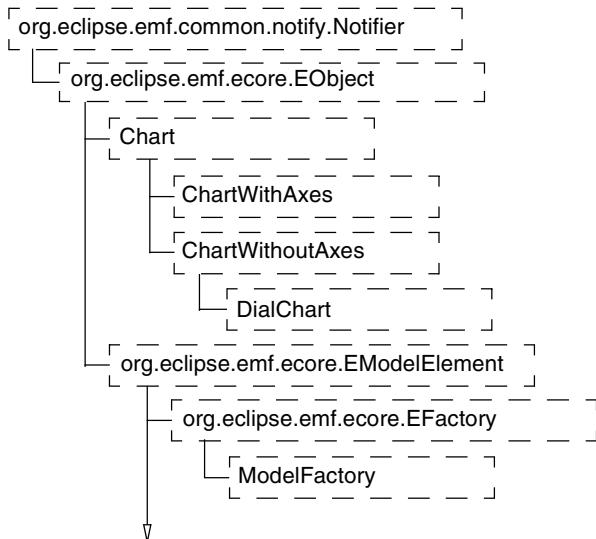


Figure 13-20 Interfaces in org.eclipse.birt.chart.model (*continues*)

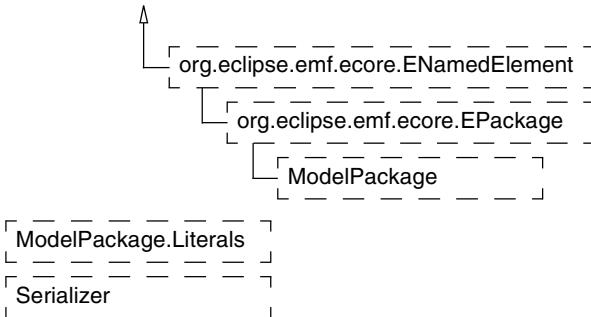


Figure 13-20 Interfaces in `org.eclipse.birt.chart.model` (*continued*)

chart.model.attribute class and interface hierarchy

The `chart.model.attribute` package contains classes that define the permitted values for attributes as static fields. Figure 13-21 contains the class hierarchy for `org.eclipse.birt.chart.model.attribute`.

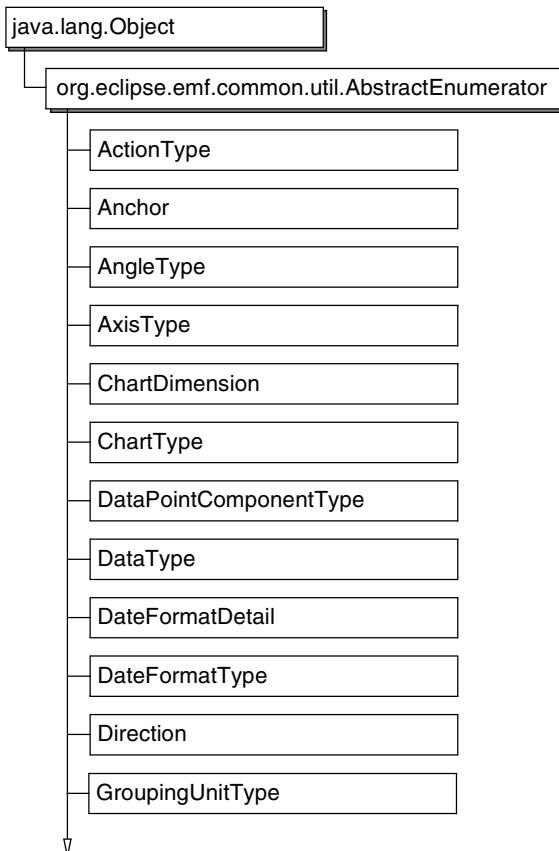


Figure 13-21 Classes in `org.eclipse.birt.chart.model.attribute`

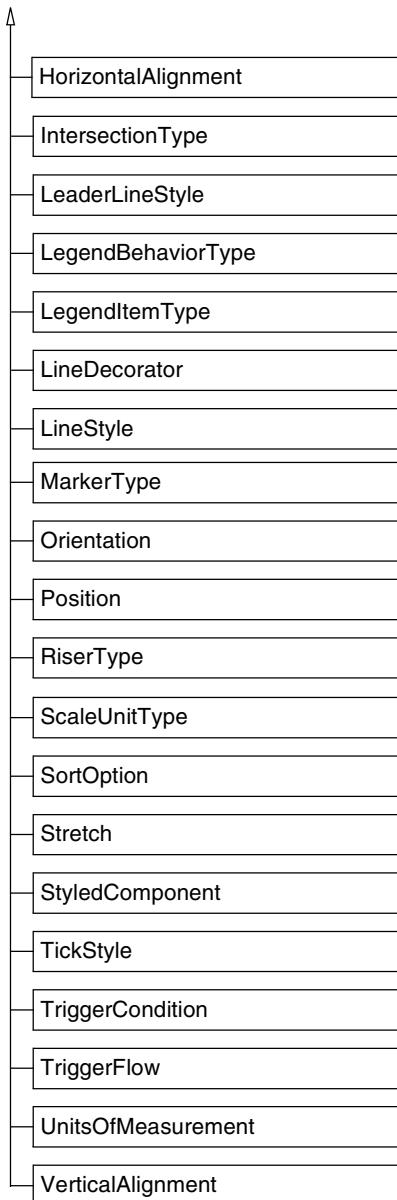


Figure 13-21 Classes in `org.eclipse.birt.chart.model.attribute` (*continued*)

The interfaces in the `chart.model.attribute` provide the getter and setter methods that chart engine objects use to test and set the values of the attributes. Figure 13-22 contains the interface hierarchy for `org.eclipse.birt.chart.model.attribute`.

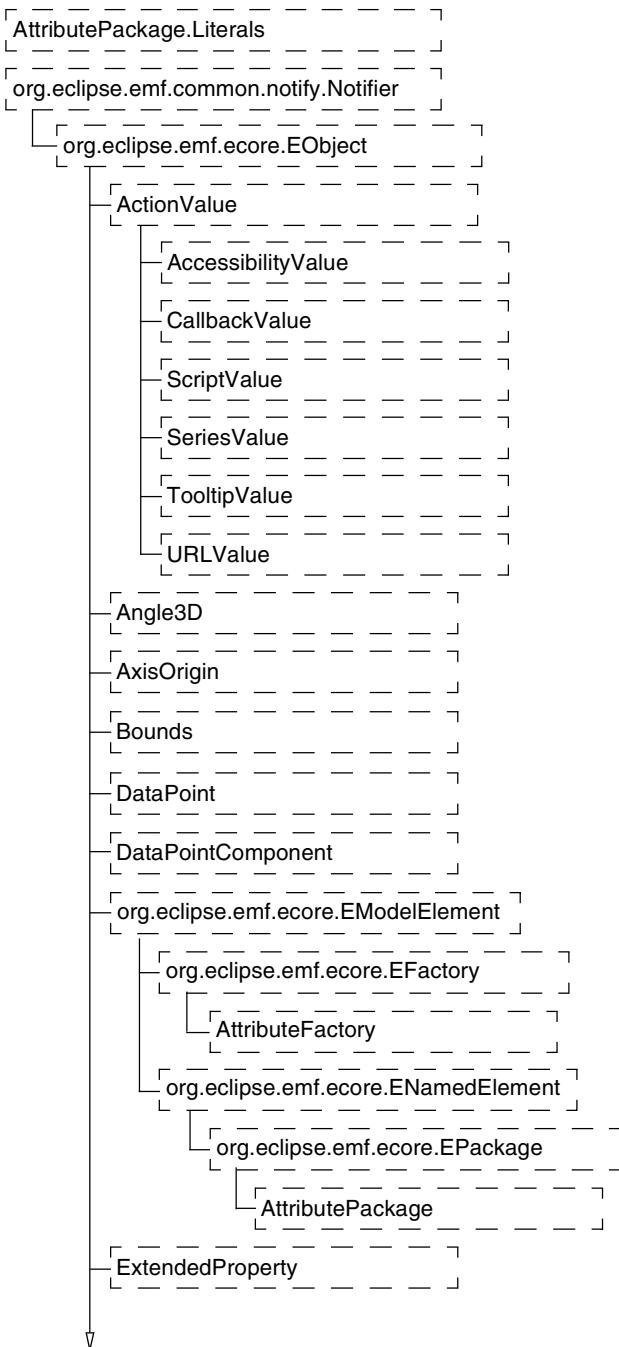


Figure 13-22 Interfaces in `org.eclipse.birt.chart.model.attribute`

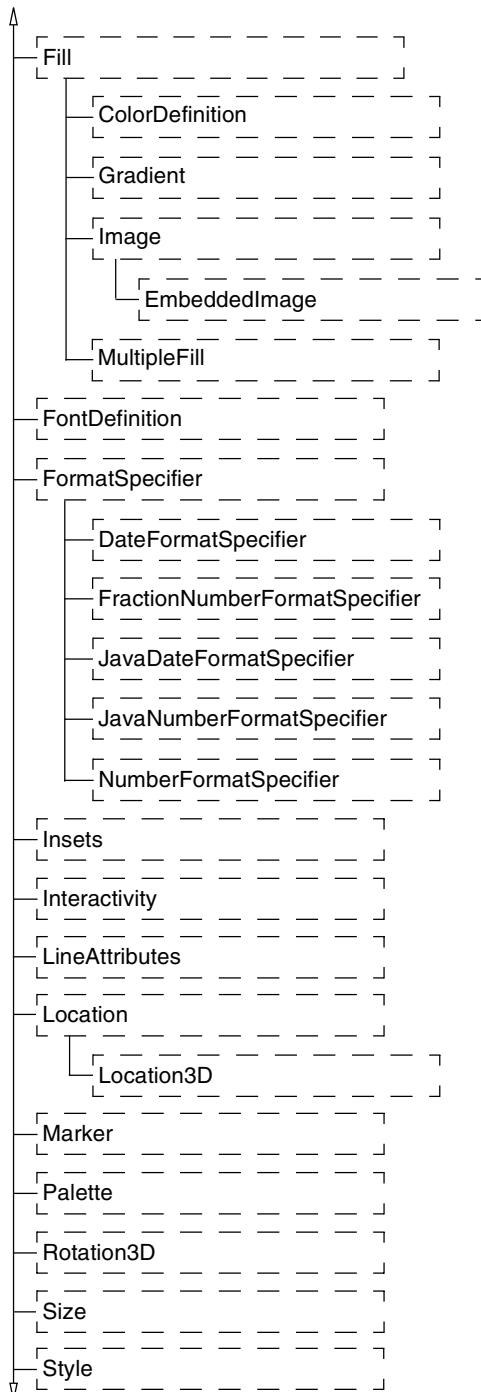


Figure 13-22 Interfaces in `org.eclipse.birt.chart.model.attribute` (*continues*)

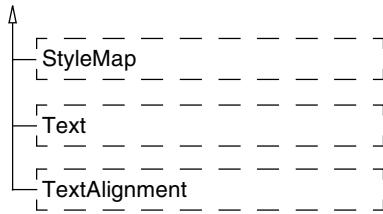


Figure 13-22 Interfaces in `org.eclipse.birt.chart.model.attribute` (*continued*)

chart.model.component interface hierarchy

The chart.model.component package contains the interfaces that define the behavior of all the components that make up a chart. Figure 13-23 contains the interface hierarchy for `org.eclipse.birt.chart.model.component`.

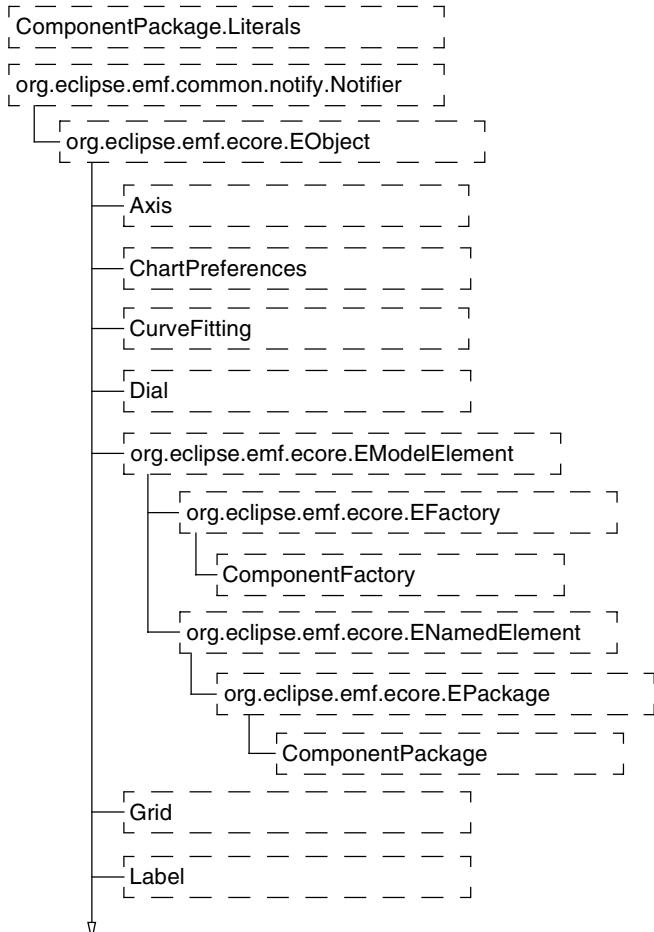


Figure 13-23 Interfaces in `org.eclipse.birt.chart.model.component`

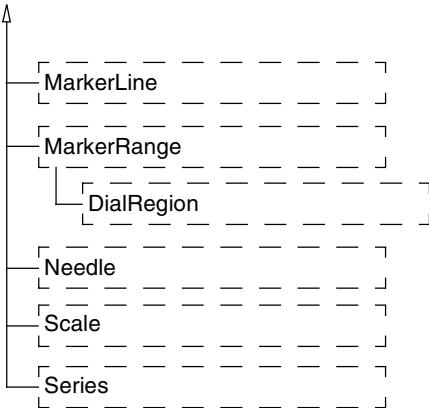


Figure 13-23 Interfaces in `org.eclipse.birt.chart.model.component` (*continued*)

chart.model.data interface hierarchy

The chart.model.data package contains interfaces that define a data source, a query structure, and the data components for a chart. These interfaces define getter and setter methods for data set properties and the permitted values for these properties as static fields. Figure 13-24 contains the interface hierarchy for `org.eclipse.birt.chart.model.data`.

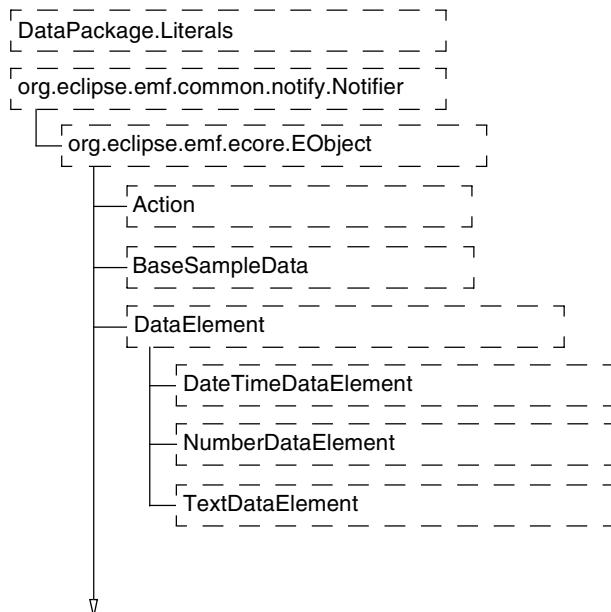


Figure 13-24 Interfaces in `org.eclipse.birt.chart.model.data` (*continues*)

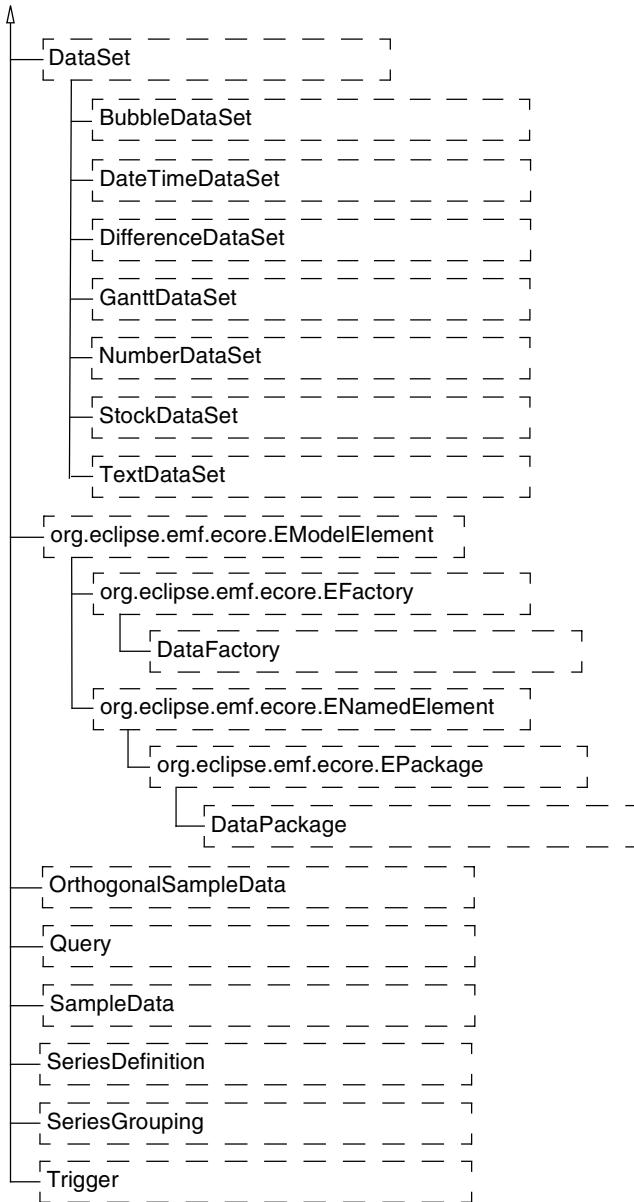


Figure 13-24 Interfaces in `org.eclipse.birt.chart.model.data` (*continued*)

chart.model.layout interface hierarchy

The `chart.model.layout` package contains the interfaces that define and arrange the blocks that make up the main components of a chart, such as the plot, title, and legend areas. Figure 13-25 contains the interface hierarchy for `org.eclipse.birt.chart.model.layout`.

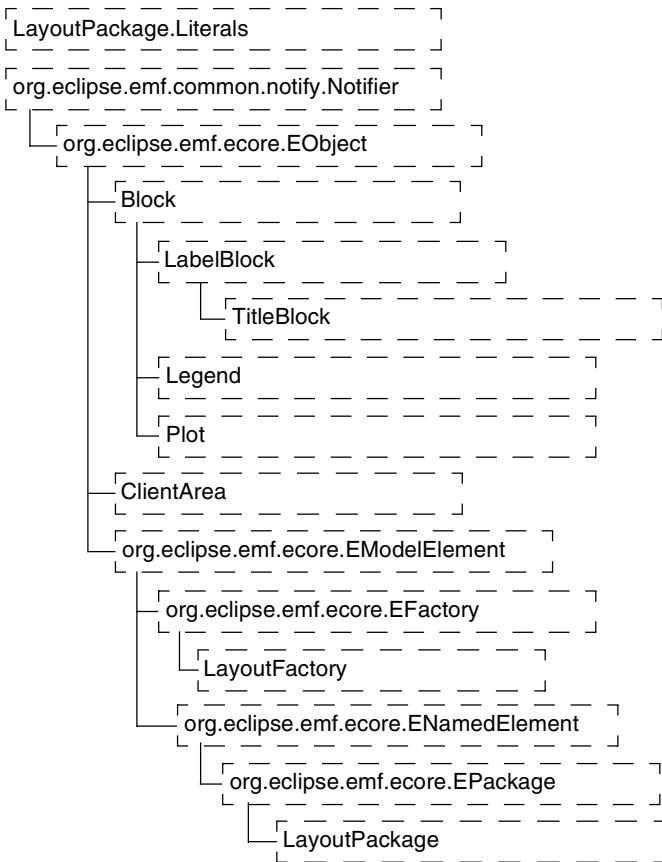


Figure 13-25 Interfaces in `org.eclipse.birt.chart.model.layout`

chart.model.type interface hierarchy

The chart.model.type package contains the interfaces that define the series for all the available chart types. Figure 13-26 contains the interface hierarchy for `org.eclipse.birt.chart.model.type`.

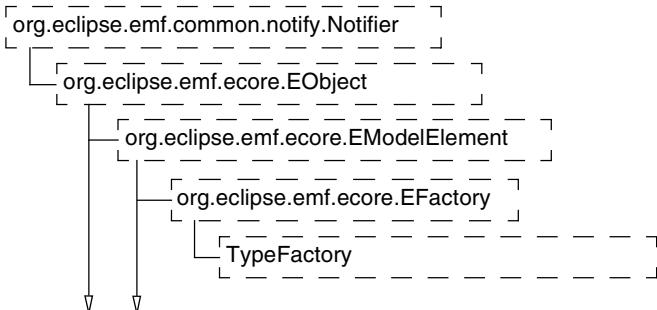


Figure 13-26 Interfaces in `org.eclipse.birt.chart.model.type` (*continues*)

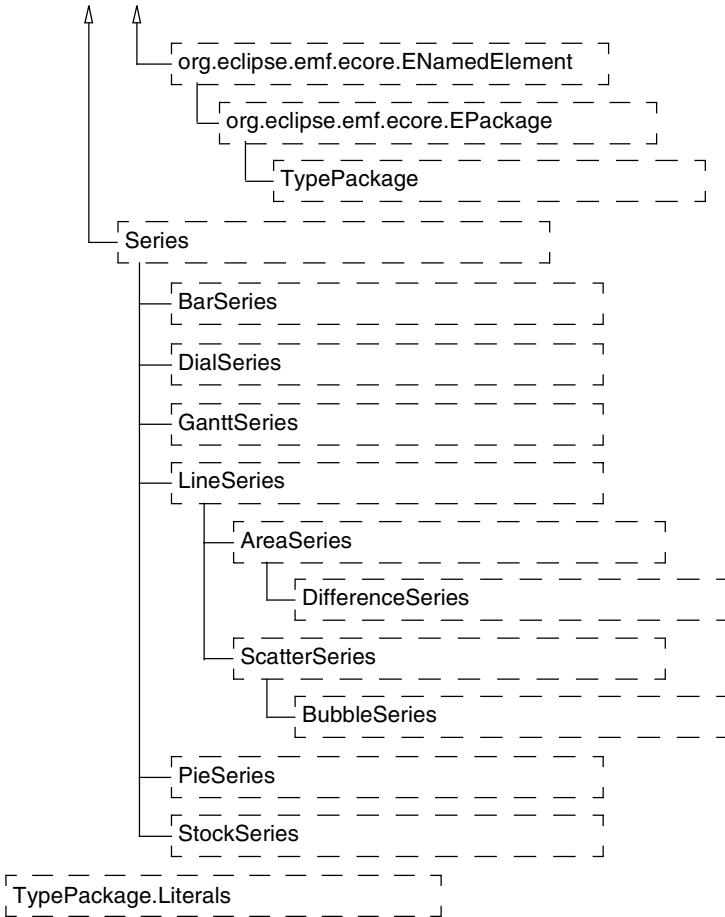


Figure 13-26 Interfaces in `org.eclipse.birt.chart.model.type` (*continued*)

chart.render class and interface hierarchy

The chart.render package contains classes and interfaces that render a chart. Figure 13-27 contains the interface hierarchy for `org.eclipse.birt.chart.render`.

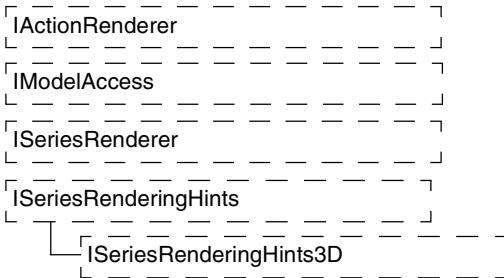


Figure 13-27 Interfaces in `org.eclipse.birt.chart.render`

Figure 13-28 contains the class hierarchy for org.eclipse.birt.chart.render.

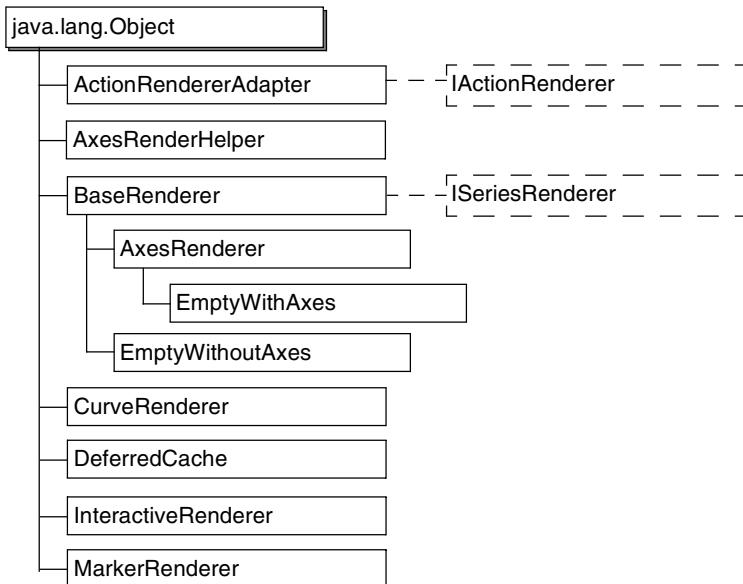


Figure 13-28 Classes in org.eclipse.birt.chart.render

chart.script class and interface hierarchy

The chart.script package contains the classes and interfaces that support script handling for chart developers to write custom code within the chart class itself. Figure 13-29 contains the class and interface hierarchy for org.eclipse.birt.chart.script.

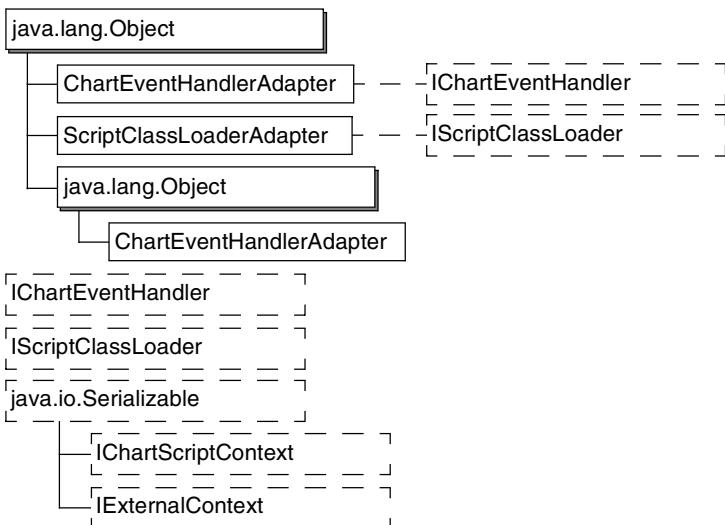


Figure 13-29 Classes and interfaces in org.eclipse.birt.chart.script package

chart.util class hierarchy

The chart.util package contains classes that support data types and looking up attribute and property values in the literal classes. Figure 13-30 contains the class hierarchy for org.eclipse.birt.chart.util.

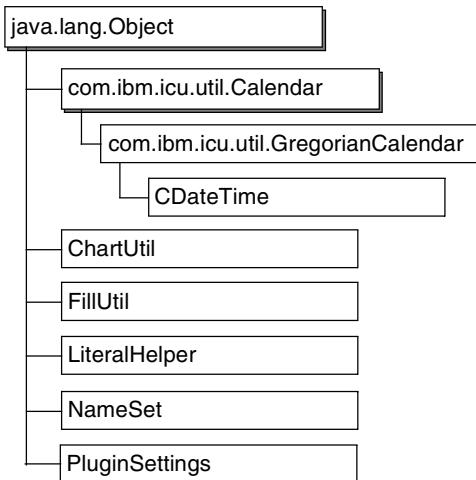


Figure 13-30 Classes in org.eclipse.birt.chart.util

14

Programming using the BIRT Reporting APIs

A reporting application uses the BIRT report engine API to generate reports from report design (.rptdesign) files. Typically, the application produces the report as a formatted file or stream, in HTML or PDF format. Alternatively, the application can create a report document (.rptdocument) file that contains the report content in binary form, then render the report to one of the supported output formats later.

This chapter describes the fundamental requirements of a reporting application and describes the BIRT API classes and interfaces that you use in the application. This chapter also provides detailed information about the tasks to perform.

The BIRT APIs in the `org.eclipse.birt.report.engine.api` package support the process of generating a report from a report design. This package provides the `ReportEngine` class and supporting interfaces and classes.

Optionally, the reporting application can use the BIRT design engine API to access the structure of report designs, templates, and libraries. With this API, the application can create and modify report designs before generating a report. This API supports creating and modifying the report items and other elements within designs. The `org.eclipse.birt.report.model.api` package and its subpackages provide access to all the items that comprise a report design.

For complete information about all the methods and fields of the classes and interfaces in these packages, see the online Javadoc. To view the Javadoc, open BIRT Report Designer and choose `Help`→`Help Contents`→`BIRT Programmer Reference`→`Reference`→`API Reference`. Choose `Report Engine API Reference` for the report engine API and `Report Object Model API Reference` for the design engine API. The Javadoc also shows supporting packages in the public API.

Building a reporting application

An application that generates a report must carry out at least the required tasks described in the following sections. Further tasks, such as supplying user-entered values for parameters, are optional.

Creating and configuring a report engine

A single report engine object can generate multiple reports from multiple report designs. Use the `Platform.createFactoryObject()` method to create a `ReportEngine` object. Set the BIRT home directory, which defines the location of required plug-ins and libraries.

Opening a report design or report document

Use one of the `openReportDesign()` methods of the `ReportEngine` class to open a report design from a String file name or an input stream. These methods return an `IReportRunnable` object.

Use the `openReportDocument()` method of the `ReportEngine` class to open a report document (.rptdocument) file from a String file name. This method returns an `IReportDocument` object.

Ensuring access to the data source

Ensure that the report engine can locate the classes that connect to the data source and supply data to the data set. The report engine can either create a connection to the data source or use a `Connection` object that the application provides.

Preparing to create a report in the supported output formats

Use an `IRenderOption` object to set the output format, the output file name or stream, the locale, and format-specific settings. The `HTMLRenderOption` class supports the HTML output format. For PDF output, use `RenderOption`.

Generating a report in one of the supported output formats

Use an `IRunAndRenderTask` object to create the report from the `IReportRunnable` object. Use an `IRenderTask` object to create the report from an `IReportDocument` object.

Alternatively, use a URL to access the report viewer servlet, such as when deploying a BIRT report to an application server, as described earlier in this book. The report viewer can generate a report from either a design file or a document file.

Destroying the engine

Destroy the report engine if the application does not need to generate more reports.

Optional tasks

The tasks in the following list are optional for a reporting application. Typically, the application performs one or more of these tasks.

- Gather values for parameters.

If the application uses a report design that has parameters, use the default values for the parameters or set different values.
- Create a report document file.

A report document file contains the report in a binary storage form. If the application uses a report design, use an `IRunTask` object to create the report document as an `IReportDocument` object.
- Export data from a report document.

Use an `IDataExtractionTask` object to extract data values from any item or set of items in the report document. Export the data values to a file or another application, or perform further processing on the data.

About the environment for a reporting application

You must ensure that the deployed application can access all the classes required for BIRT, your external data sources, and any other classes you need. The key requirement for BIRT is the path to the BIRT home directory. The BIRT home directory contains the BIRT plug-ins and libraries needed to generate a report from a report design. The BIRT Report Engine package provides a complete BIRT home in its `ReportEngine` subdirectory.

If you use or customize the BIRT source code, you must ensure that the version that your application uses matches the version of the plug-ins and libraries in the BIRT home directory. If these versions are not the same, your reporting application is likely to fail.

About plug-ins used by the report engine

The BIRT home directory has a subdirectory, `plug-ins`, that contains the `org.eclipse.birt` and other plug-ins that a reporting application can use. Depending on the requirements of your reporting application and the items in your report designs, you can omit plug-ins that provide functionality that the application and designs do not use.

About libraries used by the report engine

The `lib` subdirectory of the BIRT home directory contains the JAR files described in Table 14-1. This location within the report engine ensures that the class loader of the application server or JVM in which you deploy the report engine can locate the libraries. Depending on the requirements of your reporting application and the items in your report designs, you can omit libraries that provide functionality that the application and designs do not use.

Table 14-1 Libraries in the BIRT home lib directory

Library	Description
chartengineapi.jar	Contains chart model and factory classes. Supports generation of charts in a report.
com.ibm.icu_<version>.jar	Provides International Components for Unicode to support text in multiple locales.
commons-cli-<version>.jar	Used by the ReportRunner application in the report.engine plug-in. Provides command-line parsing. Reporting applications in a web environment do not require this library.
coreapi.jar	Contains framework and utility classes.
dataadapterapi.jar	Contains data adapter classes.
dteapi.jar	Provides access to data sources. Transforms data that the data set provides.
engineapi.jar	Required for generating a report from a report design.
flute.jar	Used by the report.model plug-in. Provides access to CSS functionality.
js.jar	Provides scripting functionality.
modelapi.jar	Describes the report design elements.
modelodaapi.jar	Used for conversions between ROM elements and required Eclipse Data Tools Platform (DTP) Open Data Access (ODA) elements.
odadesignapi.jar	Used when retrieving data.
org.apache.commons.codec-<version>.jar	Used by the report.engine plug-in. Provides encoding and decoding functionality.
org.eclipse.emf.common_<version>.jar	Required for charts.
org.eclipse.emf.ecore.xmi_<version>.jar	Required for charts.
org.eclipse.emf.ecore_<version>.jar	Required for charts.
org.w3c.css.sac_<version>.jar	Used by the report.model plug-in. Required for CSS functionality.
scriptapi.jar	Required for Java-based scripting.

About required JDBC drivers

The BIRT home plugins/org.eclipse.birt.report.data.oda.jdbc_<version> folder contains a subdirectory, drivers. Place the driver classes or Java archive (.jar) files that you require to access JDBC data sources in this location.

Modifying a report design with the API

A reporting application can also modify a report design before generating the report. For example, the application can add or remove items from the report design based on the user who generates the report. To provide further customization of a report, during the generation of the output, a report design can use Java script classes or embedded JavaScript code to handle events. The classes in the Design Engine API support making changes to a report design and including or changing scripts. The key Design Engine classes are in the org.eclipse.birt.model.api package.

Generating reports from an application

The key tasks that a reporting application must perform are to access to a BIRT home; set any parameter values; set up the task objects to generate the report; and run the report. The reporting application does not require the BIRT Report Designer user interface to generate a report.

The org.eclipse.birt.report.engine.api package contains the classes and interfaces that an application uses to generate reports. The main classes and interfaces are ReportEngine, EngineConfig, IReportRunnable, IRenderOption and its descendants, and IEngineTask and its descendants.

Setting up the report engine

A report engine is an instantiation of the ReportEngine class. This object is the key component in any reporting application. It provides access to runnable report designs, parameters, the structure of a report design, and the task for generating a report from a report design. You prepare the report engine's properties with an EngineConfig object. After setting all the required properties, you use the BIRT engine factory to create the report engine. The following sections describe the various configuration options and creating the engine.

Configuring the BIRT home

The BIRT home, which is the location of the BIRT plug-ins and libraries, is the key property that the report engine requires. The report engine cannot parse the report design nor run the report without a defined BIRT home. For a stand-alone application, the BIRT home is an absolute path to a file system location. For an application running from a web archive (.war) file on an application server, the BIRT home is a relative path in the WAR file. To set the BIRT home location, you use one of the following techniques:

- For a stand-alone application, call EngineConfig.setBIRTHome() with an argument that is the path to your BIRT home directory, for example:

```
config.setBIRTHome  
  ( "C:/birt-runtime-<version>/ReportEngine" );
```

- In your application's environment, set up the BIRT_HOME and CLASSPATH environment variables to access the required libraries. For example, in a Windows batch file that launches a stand-alone application, include commands similar to the following ones before running your application:

```
set BIRT_HOME="C:\birt-runtime-<version>\ReportEngine"
SET CLASSPATH=%BIRT_HOME%\<required library 1>;
%BIRT_HOME%\<required library 2 and so on>;
%CLASSPATH%
```

- For a web application that has a location in the file system, use the servlet context to find the real path of the BIRT home, for example:

```
config.setBIRTHome
( servletContext.getRealPath( "/WEB-INF" ) );
```

- For a web application that runs from a WAR file, use a relative path from the WAR file root, as shown in the following example. This configuration uses PlatformServletContext.

```
config.setBIRTHome( "" );
```

- In Eclipse, set BIRT_HOME in the VM arguments in the Run dialog. For example, in VM arguments, type text similar to the following line:

```
-DBIRT_HOME="C:\birt-runtime-<version>\ReportEngine"
```

- If you use BIRT in an Eclipse-based application, such as an RCP application, and the BIRT plug-ins are located in your application's plugins directory, you do not need to set BIRT_HOME.

Configuring the report engine

Optionally, you can also set other configuration properties using methods on an EngineConfig object. Table 14-2 describes these properties and how to set them with EngineConfig methods. The EngineConfig class also provides getter methods to access these properties.

Table 14-2 EngineConfig properties

Property type	Setting the property
Logging	To set the logging file location and level, call setLogConfig().
OSGi (Open Services Gateway Initiative) configuration	To set specific OSGi startup parameters, call setOSGiArguments() or setOSGiConfig().
Platform context	To indicate whether the application and BIRT home are in a stand-alone environment or packaged as a web archive (.war) file, create an implementation of the IPlatformContext interface. Then call setEngineContext().

(continues)

Table 14-2 EngineConfig properties (*continued*)

Property type	Setting the property
Resource files	To set the location where the reporting application can access resource files such as libraries and properties files that contain localized strings, call <code>setResourcePath()</code> . To change the algorithm for locating these resources, call <code>setResourceLocator()</code> .
Scripting configuration	To provide external values to scripting methods, call <code>setConfigurationVariable()</code> . To provide additional Java resources to scripting methods, call <code>getAppContext()</code> and add the object to the application context.
Status handling	To provide a custom status handler, create an implementation of the <code>IStatusHandler</code> interface. Then call <code>setStatusHandler()</code> .
Temporary file location	To set up a custom location for temporary files, call <code>setTempDir()</code> .

Setting up a stand-alone or WAR file environment

Two engine configuration properties depend on whether the environment in which the application runs is stand-alone or in a web archive (.war) file on an application server. These properties are the platform context and the HTML emitter configuration. The platform context provides the report engine with the mechanism to access plug-ins. The default platform context provides direct file system access to these plug-ins, as used by a stand-alone application. The HTML emitter configuration provides the functionality to process images and handle hyperlinking and bookmark actions.

Setting up the platform context

Because BIRT is an Eclipse-based application, it uses the OSGi platform to start up the plug-ins that make up the report and design engines. BIRT requires that the plug-ins are in a known folder, the BIRT home. The BIRT application locates the plug-ins using the platform context, which is an implementation of the `org.eclipse.birt.core.framework.IPlatformContext` interface. This interface defines the method, `getPlatform()` that returns the location of the plugins directory.

Use the `IPlatformContext` object as the argument to the `EngineConfig` object's `setEngineContext()` method. For example, the code in Listing 14-2 sets up a platform context for a reporting application that runs from a WAR file.

The BIRT framework provides two implementations of the `IPlatformContext` interface. These implementations provide all the required functionality for the platform context. The default implementation, `PlatformFileContext`, accesses the plug-ins in the BIRT home folder on the file system. Use this

implementation for a stand-alone application or for a web application that uses file system deployment on the application server.

For a web deployment, such as an application running from a WAR file on an application server, use the PlatformServletContext implementation. This class uses the resource-based access provided by the J2EE ServletContext class to locate the required plug-ins. The constructor for this class takes one argument, a ServletContext object. By default, PlatformServletContext finds plug-ins in the /WEB-INF/platform/ directory by using the ServletContext.getRealPath() method. Some application servers return null from this method. In this case, the PlatformServletContext object creates a platform directory in the location defined by the system property, javax.servlet.context.tempdir. The PlatformServletContext object copies the plug-ins and configuration folders to this location.

If neither of these IPlatformContext implementations meets your needs, implement your own version of IPlatformContext. In the same way as for the built-in platform contexts, call EngineConfig.setPlatformContext() to configure the engine to use the new implementation.

Setting up the HTML emitter

When you generate a report in HTML format, the report engine uses an org.eclipse.birt.report.engine.api.HTMLRenderOption object to determine how to handle the resources that the HTML emitter uses. These resources include new image files for image elements and chart elements, and the locations of jumps from bookmark and drill-through actions. BIRT uses different image handlers for file system-based applications and applications deployed on the web. To set up the HTML emitter, instantiate an HTMLRenderOption object. Use this object as the argument to ReportEngine.setEmitterConfiguration(), as shown in Listing 14-1. You do not have to set up the HTML emitter when you create the report engine. If you prefer, you can set it up later when you render a report in HTML.

Call the HTMLRenderOption.setImageHandler() method to configure the image handler. This method determines how to save the image files by using an org.eclipse.birt.report.engine.api.IHTMLImageHandler object. Images that are defined in a report as a URL are not saved locally.

BIRT provides two implementations of the IHTMLImageHandler interface. Both implementations create unique image file names for temporary images. The default image handler implementation, HTMLCompleteImageHandler, saves images to the file system. Use this implementation for a stand-alone application or for a web application that uses file system deployment on the application server. This image handler finds the location for images by searching first for the image directory set in the HTMLRenderOption object, next the temporary files directory as set by the EngineConfig.setTempDir() method, and finally by the location set by the system setting, java.io.tmpdir. All images in the generated HTML use file references to the created images.

For a web deployment, such as an application running from a WAR file on an application server, use the HTMLServerImageHandler implementation. This

implementation saves the images to the image directory set in the `HTMLRenderOption` object. The `src` attribute of any images in the generated HTML appends the image name to the the base image URL, which is also set in the `HTMLRenderOption` object. In this way, the report engine produces the images locally and shares the images using a URL. To use this implementation, you must set the image directory and the base image URL, as shown in Listing 14-3. The example BIRT Web Viewer in the `org.eclipse.birt.report.viewer` plug-in uses this implementation.

If neither of these `IHTMLImageHandler` implementations meets your needs, implement your own version of `IHTMLImageHandler` interface or extend an existing image handler. Typically, the functionality for file system access that `HTMLCompleteImageHandler` provides is sufficient, so your application does not need to extend it. For the application server environment, you need to extend from `HTMLServerImageHandler`. In the same way as for the built-in image handlers, call `EngineConfig.setEmitterConfiguration()` to configure the engine to use the new implementation.

Starting the platform

After setting up the `PlatformContext`, your application can start the platform by using the `org.eclipse.birt.core.framework.Platform` class. This class acts as a wrapper around the Eclipse OSGILauncher class and provides the synchronized static method, `startup()`, to start the platform. This operation is expensive, so your application should do it only done once. When your application is finished with the platform, call `Platform.shutdown()`. If you deploy the report engine as a web application, call `Platform.startup()` in the servlet's `init()` method or in the first request that uses the platform. The application can achieve this behavior by wrapping the platform startup code in a singleton, as shown in Listing 14-2. If you use the example Web Viewer or deploy the report engine in an Eclipse-based project such as an Rich Client Platform (RCP) application, you do not need to start up or shut down the platform, because these applications start OSGi themselves.

Creating the report engine

BIRT provides a factory service for creating the `ReportEngine` object. Create the factory, which is an implementation of the interface, `org.eclipse.birt.report.engine.api.IReportEngineFactory` using the `Platform.createFactoryObject()` method. This method uses a `PlatformConfig` object. Because `EngineConfig` extends `PlatformConfig`, you can use the `EngineConfig` object to create the factory. Finally, create the report engine using the `IReportEngineFactory.createReportEngine()` method and the same `EngineConfig` object.

How to set up a report engine as a stand-alone application

Listing 14-1 shows an example of setting up a report engine as a stand-alone application on a Windows system. The application uses the BIRT home located in the BIRT run-time directory. The report output format is HTML.

Listing 14-1 Setting up the report engine for a stand-alone application

```
// Create an EngineConfig object.  
EngineConfig config = new EngineConfig( );  
// Set up the path to your BIRT home directory.  
config.setBIRTHome  
    ( "C:/Program Files/birt-runtime-2_2_1/ReportEngine" );  
// Explicitly set up the stand-alone application  
IPlatformContext context = new PlatformFileContext( );  
config.setEngineContext( context );  
// Start the platform for a non-RCP application.  
Platform.startup( config );  
IReportEngineFactory factory =  
    ( IReportEngineFactory ) Platform.createFactoryObject  
        ( IReportEngineFactory.EXTENSION_REPORT_ENGINE_FACTORY );  
// Set up writing images or charts embedded in HTML output.  
HTMLRenderOption ho = new HTMLRenderOption( );  
ho.setImageHandler( new HTMLCompleteImageHandler( ) );  
config.setEmitterConfiguration  
    ( RenderOptionBase.OUTPUT_FORMAT_HTML, ho );  
// Create the engine.  
IReportEngine engine = factory.createReportEngine( config );
```

How to set up a report engine as a web application

- 1 Set up the platform context, as shown in Listing 14-2.

Listing 14-2 Setting up the platform context for WAR file deployment

```
// Example class to create the report engine  
public class BirtEngine {  
    private static IReportEngine birtEngine = null;  
  
    public static synchronized IReportEngine  
    getBirtEngine( ServletContext sc ) {  
        if (birtEngine == null) {  
            EngineConfig config = new EngineConfig( );  
            config.setBIRTHome( "" );  
            IPlatformContext context =  
                new PlatformServletContext( sc );  
            config.setPlatformContext( context );  
            try {  
                Platform.startup( config );  
                IReportEngineFactory factory =  
                    ( IReportEngineFactory )  
                    Platform.createFactoryObject  
                    ( IReportEngineFactory.EXTENSION_REPORT_ENGINE_FACTORY );  
                birtEngine =  
                    factory.createReportEngine( config );  
            }  
            catch ( Exception e ) { e.printStackTrace( ); }  
        }  
    }
```

```

        }
        return birtEngine;
    }
}

```

- 2** Set up further configuration options on the engine after you instantiate the class, as shown in Listing 14-3.

Listing 14-3 Setting up HTML options for WAR file deployment

```

// In a different class, get the report engine
reportEngine = BirtEngine.getBirtEngine
    ( request.getSession( ).getServletContext( ) );
// Set up the engine
EngineConfig config = reportEngine.getConfig( );
HTMLRenderOption ho = new HTMLRenderOption( );
ho.setImageHandler( new HTMLServerImageHandler( ) );
ho.setImageDirectory("output/image");
ho.setBaseImageURL("http://myhost/prependme?image=");
config.setEmitterConfiguration
    ( RenderOptionBase.OUTPUT_FORMAT_HTML, ho );

```

Using the logging environment to debug an application

BIRT Report Engine uses the `java.util.logging` classes, `Logger` and `Level`, to log information about the processing that the engine performs. When you run an application in the Eclipse workbench, by default, the messages appear in the console. When you run an application external to Eclipse, the default location of the log messages depends on your environment. The default logging threshold is `Level.INFO`. Typically, you change this level in your application to reduce the number of internal logging messages.

To set up the logging environment to write the engine's log messages to a file on disk, use the `EngineConfig.setLogConfig()` method. This method takes two arguments. The first argument is the directory in which to create the log file. BIRT Report Engine creates a log file with a name whose format is `ReportEngine_YYYY_MM_DD_hh_mm_ss.log`. The second argument is the lowest level at which to log information. Set the logging level to a high threshold so that the engine logs fewer messages. Typically, you want to see information at `INFO` level when you first develop a block of code. To modify the amount of information that the engine logs, use the `ReportEngine.changeLogLevel()` method. This method takes a single argument, which is a `Level` constant. When the code is stable, you no longer need to see all the engine's `INFO` messages. At that point, you can delete or comment out the call to `changeLogLevel()`.

How to use BIRT logging

The following example shows how to use logging in an application. You set up the logging environment, then modify it later in your application. The variable, `config`, is the `EngineConfig` object that the code's active `ReportEngine` object, `engine`, is using.

- 1 Configure logging on the report engine object.

```
// Set up the location and level of the logging output.
config.setLogConfig( "C:/Temp", Level.ERROR );
```
- 2 In any newly written code, increase the amount of logging.

```
engine.changeLogLevel( Level.INFO );
```

Opening a source for report generation

BIRT Report Engine can generate a report from either a report design or a report document. The engine can also generate a report document from a report design.

To open a report design, you call one of the `openReportDesign()` methods on `ReportEngine`. These methods instantiate an `IReportRunnable` object, using a String that specifies the path to a report design or an input stream.

To open a report document, you call `ReportEngine.openReportDocument()`. This method instantiates an `IReportDocument` object, using a String that specifies the path to a report document. You must handle the `EngineException` that these methods throw.

Understanding an `IReportRunnable` object

The `IReportRunnable` object provides direct access to basic properties of the report design. The names of report design properties are static String fields, such as `IReportRunnable.AUTHOR`. To access a report design property, use the `getProperty()` method with a String argument that contains one of these fields.

To access and set the values of parameters generically, you use methods on a parameter definition task object, as described later in this chapter. If the application developer knows the parameter names in the report design, you use methods on the run task instead of using the parameter definition task. To generate a report from a design, open the report design as shown in Listing 14-4, then perform the tasks shown later in this chapter.

How to access a report design

Listing 14-4 shows how to open a report design and find a property value. If the engine cannot open the specified report design, the code destroys the engine. The variable, `engine`, is a `ReportEngine` object.

Listing 14-4 Accessing a report design

```
String designName = "./SimpleReport.rptdesign";
IReportRunnable runnable = null;
try {
    runnable = engine.openReportDesign( designName );
}
catch ( EngineException e ) {
```

```

        System.err.println(
            ( "Design " + designName + " not found!" );
        engine.destroy( );
        System.exit( -1 );
    }
    // Get the value of a simple property.
    String author = ( String ) runnable.getProperty(
        IReportRunnable.AUTHOR );

```

Understanding an IReportDocument object

The IReportDocument object provides access to the data in a report and the report's structure. IReportDocument provides methods to retrieve table of contents entries, bookmarks, and page information.

To access table of contents entries, use the findTOC() method. This method takes a TOCNode argument and returns a TOCNode object. To find the root table of contents entry, use an argument of null. To access the table of contents entries for a particular locale, call the getTOCTree() method, which returns an ITOCTree object. To find the subentries of a table of contents entry, use the TOCNode.getChildren() method. This method returns a List of TOCNode objects. From a TOCNode object, you can retrieve the display value of the entry and a Bookmark object. In turn, you can use the Bookmark object as an argument to the getPageNumber() method, which returns the number of the page to which the bookmark links. With this information, you can specify particular pages to render to a formatted report.

How to access a report document

Listing 14-5 shows how to open a report document and navigate its table of contents to find a page. If the engine cannot open the specified report design, the code destroys the engine. The variable, engine, is a ReportEngine object.

Accessing a report parameter programmatically

A report parameter is a report element that provides input to a report design before the application generates the report. A report document does not use report parameters for generating a report. If your application already has the parameter names and the values you wish to set, or the default values for all report parameters for a report design are always valid, or your report source is a report document, you do not need to perform the tasks in this section.

Report parameters have attributes that a reporting application can access. The most commonly used attributes are name and value. The report engine uses the report design logic and the report parameter values to perform tasks such as filtering a data set or displaying an external value in the report.

After the reporting application sets the values for the report parameters, it must pass these values to the task that generates the report, as shown later in this chapter. To access report parameters and their default values and to set user-supplied values to a parameter, the reporting application uses the BIRT report engine API classes and interfaces shown in Table 14-3.

Listing 14-5 Accessing a report document

```
String dName = "./SimpleReport.rptdocument";
IReportDocument doc = null;
try {
    doc = engine.openReportDocument( dName );
} catch ( EngineException e ) {
    System.err.println( "Document " + dName + " not found!" );
    engine.destroy();
    System.exit( -1 );
}
// Get the root of the table of contents.
TOCNode td = doc.findTOC( null );
java.util.List children = td.getChildren();
long pNumber;
// Loop through the top level table of contents entries.
if ( children != null && children.size( ) > 0 ) {
    for ( int i = 0; i < children.size( ); i++ ) {
        // Find the required table of contents entry.
        TOCNode child = ( TOCNode ) children.get( i );
        if ( child.getDisplayString( ).equals( "103" ) ) {
            // Get the number of the page that contains the data.
            pNumber = doc.getPageNumber( child.getBookmark( ) );
            System.out.println( "Page to print is " + pNumber );
        }
    }
}
```

You can access report parameters by name or with generic code. You use generic code if the application must be able to run any report design, for example, if you access report designs from a list that depends on user input. If the application runs only a fixed set of known report designs, you can access the report parameters by name.

Table 14-3 Classes that support report parameters

Class or interface	Use
ReportEngine	The class that instantiates the task that accesses report parameters. Call the createGetParameterDefinitionTask() method to create the task object. This method returns an IGetParameterDefinitionTask object.
IGetParameterDefinitionTask	The interface that accesses a single report parameter or a collection of all the report parameters in a report design. This interface also provides access to valid values for parameters that use restricted sets of values, such as cascading parameters.

Table 14-3 Classes that support report parameters (*continued*)

Class or interface	Use
IParameterDefnBase	The base interface for report parameter elements. Scalar parameters implement the derived interface, IScalarParameterDefn. Parameter groups implement the derived interface IParameterGroupDefn.
IParameterGroupDefn	The base interface for report parameter groups. Cascading parameter groups implement the derived interface ICascadingParameterGroup.
IParameterSelectionChoice	The interface for valid values for a report parameter that uses a restricted set of values, such as a cascading parameter.
ReportParameter Converter	The class that converts a String value provided by a user interface into a locale-independent format.

Creating a parameter definition task object for the report design

A single IGetParameterDefinitionTask object provides access to all parameters in a report design. Create only one of these objects for each report design, by calling ReportEngine.createGetParameterDefinitionTask(). When you have finished using the parameter definition task object, close the task by calling its close() method.

Testing whether a report design has report parameters

To test if a report design has report parameters, call the getParameterDefns() method on IGetParameterDefinitionTask. This method returns a Collection. To test whether the Collection has elements call the Collection.isEmpty() method. If the application runs only known report designs, you do not need to perform this task.

Getting the report parameters in a report design

To access a single report parameter with a known name, use the IGetParameterDefinitionTask.getParameterDefn() method. This method returns an object of type IParameterDefnBase. Alternatively, use the IGetParameterDefinitionTask.getParameterDefns() method to return a Collection of IParameterDefnBase objects. The application can then use an Iterator to access each report parameter from this Collection in turn.

The getParameterDefns() method takes a Boolean argument. When the argument is false, the method returns an ungrouped set of report parameters. When the argument is true, the method returns parameter groups, as defined in the report design. To create a user interface that replicates the parameter group structure, use a value of true.

To check whether a report parameter is a group, the application must call IParameterDefnBase.getParameterType(). This method returns IParameterDefnBase.PARAMETER_GROUP when the parameter is a group or IParameterDefnBase.CASCADING_PARAMETER_GROUP when the parameter is a cascading parameter group. To access the group's report parameters, use the IParameterGroupDefn.getContents() method. This method returns an ArrayList object of objects of type IScalarParameterDefn.

Getting the default value of each report parameter

This task is optional. To get the default value of a single known report parameter, use IGetParameterDefinitionTask.getDefaultValue(). This method returns an Object. To determine the effective class of the Object, use IScalarParameterDefn.getDataType(). This method returns an int value, which is one of the static fields in IScalarParameterDefn. Call IGetParameterDefinitionTask.getDefaultValues() to get the default value of all parameters in the report design. This method returns a HashMap object, which maps the report parameter names and their default values.

Getting valid values for parameters using a restricted set of values

Some report parameters accept only values from a restricted list. In some cases, this list is a static list of values, such as RED, BLUE, or GREEN. In other cases, the list is dynamic and a query to a database provides the valid values. For example, a query can return the set of sales regions in a sales tracking database. To determine the list of valid values, call the IGetParameterDefinitionTask.getSelectionList() method. This method returns a Collection of IParameterSelectionChoice objects. IParameterSelectionChoice has two methods. getLabel() returns the display text and getValue() returns the value. If the Collection is null, the report parameter can take any value.

Getting the attributes of each report parameter

This task is optional. To get the attributes of a report parameter, use the IScalarParameterDefn methods. The application can use the attributes to generate a customized user interface. For example, to get the data type of a report parameter, use the getDataType() method.

Collecting an updated value for each report parameter

To provide new values for the report parameters, provide application logic such as a user interface or code to retrieve values from a database. Call IGetParameterDefinitionTask.setParameterValue() to set the value of the parameter.

If you provide a user interface that returns String values to your application for date and number parameters, you must convert the String into a locale-independent format before setting the value. To perform this task, first call

`ReportParameterConverter.parse()` to set the value to a locale-independent format. Next call `IGetParameterDefinitionTask.setParameterValue()`.

After setting the report parameter values, call the `IGetParameterDefinitionTask.getParameterValues()` method. This method returns a `HashMap` object that contains values that calls to `IGetParameterDefinitionTask.setParameterValue()` set. You can later use this `HashMap` object to set the report parameter values for report generation, as described later in this chapter.

How to set the value of a known report parameter

The code sample in Listing 14-6 shows how to set the value of a report parameter that has a known name. The sample creates a `HashMap` object that contains the parameter values to use later to run the report. The variable, `engine`, is a `ReportEngine` object. The variable, `Runnable`, is an object of type `IReportRunnable`. This sample does not show details of code for retrieving the parameter value from a user interface or a database. The code to perform these tasks depends on your application's requirements.

Listing 14-6 Setting the value of a single parameter

```
// Create a parameter definition task.  
IGetParameterDefinitionTask task =  
    engine.createGetParameterDefinitionTask( runnable );  
// Instantiate a scalar parameter.  
IScalarParameterDefn param = (IScalarParameterDefn)  
    task.getParameterDefn( "customerID" );  
// Get the default value of the parameter. In this case,  
// the data type of the parameter, customerID, is Double.  
int customerID =  
    ((Double) task.getDefaultValue( param )).intValue( );  
// Get a value for the parameter. This example assumes that  
// this step creates a correctly typed object, inputValue.  
// Set the value of the parameter.  
task.setParameterValue( "customerID", inputValue );  
// Get the values set by the application for all parameters.  
HashMap parameterValues = task.getParameterValues( );  
// Close the parameter definition task.  
task.close( );
```

How to use the Collection of report parameters

The code sample in Listing 14-7 shows how to use the Collection of report parameters. The sample uses the `ReportParameterConverter` class to convert the `String` values that the user interface supplies into the correct format for the parameter. The sample creates a `HashMap` object that contains the parameter values to use later to run the report. The variable, `engine`, is a `ReportEngine` object. The variable, `Runnable`, is an object of type `IReportRunnable`. This sample does not show details of code for retrieving the parameter values from a user interface or a database. The code to perform these tasks depends on your application's requirements.

Listing 14-7 Setting the values of multiple parameters without grouping

```
// Create a parameter definition task.  
IGetParameterDefinitionTask task =  
    engine.createGetParameterDefinitionTask( runnable );  
  
// Create a collection of the parameters in the report design.  
Collection params = task.getParameterDefns( false );  
// Get the default values of the parameters.  
HashMap parameterValues = task.getDefaultValues( );  
  
// Get values for the parameters. Later code in this example  
// assumes that this step creates a HashMap object,  
// inputValues. The keys in the HashMap are the parameter  
// names and the values are those that the user provided.  
  
// Iterate through the report parameters, setting the values  
// in standard locale-independent format.  
Iterator iterOuter = params.iterator();  
ReportParameterConverter cfgConverter = new  
    ReportParameterConverter( "", Locale.getDefault() );  
while ( iterOuter.hasNext() ) {  
    IParameterDefnBase param =  
        (IParameterDefnBase) iterOuter.next();  
    String value = (String) inputValues.get( param.getName() );  
    if ( value != null ) {  
        parameterValues.put( param.getName(),  
            cfgConverter.parse( value, param.getDataType() ) );  
    }  
}  
// Close the parameter definition task.  
task.close();
```

Getting the values for cascading parameters

A cascading parameter group contains an ordered set of parameters that provide lists of acceptable values for the end user to select. The value chosen for the first parameter affects the available values for the second one, and so on. The parameters use one or more queries to retrieve the values to display to the user from a data set. The parameter definition task uses the data rows from the queries to filter the values for each parameter in the group, based on the values of preceding parameters in the group. For example, consider a cascading parameter group that uses the following query:

```
SELECT  
    PRODUCTS.PRODUCTLINE,  
    PRODUCTS.PRODUCTNAME,  
    PRODUCTS.PRODUCTCODE  
FROM CLASSICMODELS.PRODUCTS
```

The group contains two parameters, ProductLine on PRODUCTS.PRODUCTLINE and ProductCode on PRODUCTS.PRODUCTCODE. The

display text for ProductCode is PRODUCTS.PRODUCTNAME. Figure 14-1 shows the appearance of the requester that prompts for values for these parameters when you preview the report in BIRT Report Designer.

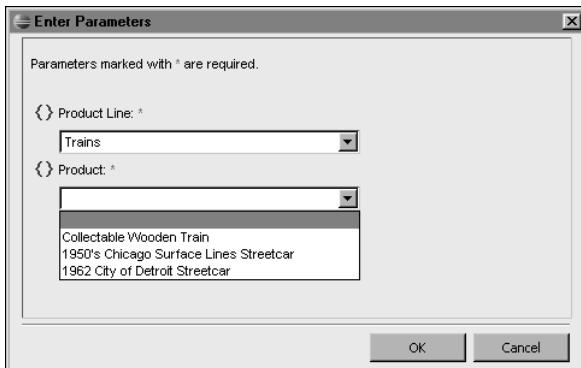


Figure 14-1 Cascading report parameters

To use the report engine API to get the values for cascading parameters, perform the tasks in the following list.

- To prepare the data values for the cascading parameters, call the method, IGetParameterDefinitionTask.evaluateQuery(). This method takes the String name of the parameter group as a parameter.
- To populate the list of values for the first report parameter in the group, call IGetParameterDefinitionTask.getSelectionListForCascadingGroup(). This method takes two parameters, the String name of the parameter group and an array of Object. For the first parameter, this array is empty. The method returns a Collection of IParameterSelectionChoice objects.
- To populate the list of values for further report parameter in the group, call getSelectionListForCascadingGroup() again. In this case, the Object[] array contains the values for the preceding report parameters in the group. In the example shown in Figure 14-1, the Object[] array is:

```
new Object[ ] {"Trains" }
```

How to use cascading parameters

The code sample in Listing 14-8 first shows how to run the query for cascading parameters. Next, the sample accesses the set of valid values for each parameter in the cascading parameter group in turn. The variable, task, is an object of type IGetParameterDefinitionTask.

Preparing to generate the report

BIRT provides output emitters for HTML, Adobe PDF, Adobe PostScript (PS), Microsoft Excel (XLS), Microsoft PowerPoint (PPT), and Microsoft Word (DOC) formats. You can also provide custom output formats by creating a new renderer from the rendering extension points, as discussed in Chapter 19, "Developing a Report Rendering Extension."

Listing 14-8 Getting the valid values for cascading parameters

```
// Create a grouped collection of the design's parameters.
Collection params = task.getParameterDefns( true );

// Iterate through the parameters to find the cascading group.
Iterator iter = params.iterator();
while ( iter.hasNext() ) {
    IParameterDefnBase param = (IParameterDefnBase) iter.next();
    if ( param.getParameterType() ==
        IParameterDefnBase.CASCADING_PARAMETER_GROUP ) {
        ICascadingParameterGroup group =
            (ICascadingParameterGroup) param;
        Iterator i2 = group.getContents().iterator();

        // Run the query for the cascading parameters.
        task.evaluateQuery( group.getName() );
        Object[ ] userValues =
            new Object[group.getContents().size()];

        // Get the report parameters in the cascading group.
        int i = 0;
        while ( i2.hasNext() ) {
            IScalarParameterDefn member =
                (IScalarParameterDefn) i2.next();

            // Get the values for the parameter.
            Object[ ] setValues = new Object[i];
            if ( i > 0 )
                System.arraycopy( userValues, 0, setValues, 0, i );
            Collection c = task.getSelectionListForCascadingGroup
                ( group.getName(), setValues );
            // Iterate through the values for the parameter.
            Iterator i3 = c.iterator();
            while ( i3.hasNext() ) {
                IParameterSelectionChoice s =
                    ( IParameterSelectionChoice ) i3.next();
                String choiceValue = s.getValue();
                String choiceLabel = s.getLabel();
            }
            // Get the value for the parameter from the list of
            // choices. This example does not provide the code for
            // this task.
            userValues[i] = inputChoiceValue;
            i++;
        }
    }
}
```

Three task classes support generating a report from a source. Sections earlier in this chapter described how to open the two types of source, a report design and a report document. The tasks that you use to generate a report from the source are:

- IRunAndRenderTask. An object of this type creates a report in one of the supported formats by running a report design directly. To instantiate this object, call the ReportEngine method, createRunAndRenderTask().
- IRunTask. An object of this type creates a report document (.rptdocument) file from a report design. To instantiate this object, call the ReportEngine method, createRunTask(). After creating the report document, you create the report output with an IRenderTask object.
- IRenderTask. An object of this type creates a complete report or a set of pages from a report by formatting the contents of a report document. To instantiate this object, call the ReportEngine method, createRenderTask().

Each type of task object can act on multiple sources. When the application no longer needs the task object, call the task's close() method.

Setting the parameter values for running a report design

To set the values for parameters for generating a report, use methods on an IRunAndRenderTask or an IRunTask object. These tasks run a report design to generate output. IRenderTask does not support changing the parameters for a report because its source is a report document. The IRunTask object that created the report document already specified the parameter values.

Call setParameterValues() to set the values for all the parameters. This method takes a HashMap as an argument. To create a suitable HashMap, use the techniques shown in Listing 14-6 or Listing 14-7, earlier in this chapter. To set the value for a single parameter when generating a report, call the setParameterValue() method. When the task generates the report or the report document, it uses the default values for any parameters that were not set by either of these methods.

Adding to the report engine's class path

Some report designs require access to external Java classes. The BIRT Report Engine uses class path information from various settings in its environment to locate the external classes. You can define some of these locations by setting properties programmatically on the application context or on the EngineConfig object, or with the Java System class. To set the properties, use constants from the org.eclipse.birt.report.engine.api.EngineConstants class. To set a property on the application context, use the EngineTask object or the EngineConfig object, as shown in the following lines of code, where MyClass is the class that starts the report engine:

```
configOrTask.getApplicationContext().put
  ( EngineConstants.APPCONTEXT_CLASSLOADER_KEY,
    MyClass.class.getClassLoader() );
```

To set a CLASSPATH property on the EngineConfig object, use code similar to the following lines. The property value must be a valid CLASSPATH.

```
config.setProperty( EngineConstants.WEBAPP_CLASSPATH_KEY,
  "c:/myjars/jar1.jar;c:/myclasses" );
```

To set a CLASSPATH property with the Java System class, use code similar to the following lines. The property value must be a valid CLASSPATH.

```
System.setProperty( EngineConstants.WEBAPP_CLASSPATH_KEY,  
    "c:/myjars/jar1.jar;c:/myclasses" );
```

BIRT searches locations for external classes in the order shown in the following list:

- The CLASSPATH for the report engine plug-in.
- The CLASSPATH of the parent class loader that is set as the EngineConstants.APPCONTEXT_CLASSLOADER_KEY. Define this property on the application context.
- The CLASSPATH set as EngineConstants.WEBAPP_CLASSPATH_KEY. Define this property with the Java System class or on the EngineConfig object.
- The CLASSPATH set as EngineConstants.PROJECT_CLASSPATH_KEY. Define this property with the Java System class or on the EngineConfig object.
- The CLASSPATH in EngineConstants.WORKSPACE_CLASSPATH_KEY. Define this property with the Java System class or on the EngineConfig object.
- JAR files that are included in the report design.

Providing an external object to a report design

BIRT supports an application passing previously instantiated objects into the report engine. In this way, the engine does not have to re-create the object. The calling application can manipulate the object in memory immediately before calling the report engine. Typically, you use external objects in the BIRT scripting environment. After you pass the object to the report engine, script expressions can reference the object by name at the appropriate stage in the report generation process. To supply an object to the report engine, you use the application context, which you get from either the EngineConfig object or the task object, as shown in the code in Listing 14-9.

Listing 14-9 Setting up an external Connection object

```
MyObject mo = new MyObject();  
config = new EngineConfig();  
// Get the application context from the config or the task  
HashMap hm = config.getAppContext();  
HashMap hm = task.getAppContext();  
hm.put( "MyCreatedObject", mo );  
config.setAppContext( hm );  
  
// To refer to this object in a BIRT script  
// or expression, use MyCreatedObject.myMethod()
```

Setting up the rendering options

Before generating a report to one of the supported output formats, the application must set options that determine features of the output. The options must specify either an output file name or a stream. Other configuration settings, such as whether to create embeddable HTML, are optional. BIRT supports two types of HTML output, HTML and embeddable HTML. Embeddable HTML is suitable for including inside another web page. This format contains no header information nor the <body> or <html> tags.

The application uses a rendering options object to set the output options on an IRunAndRenderTask or an IRenderTask object. The format-specific rendering options classes implement IRenderOption and extend RenderOption. The rendering options class supporting the HTML format is HTMLRenderOption and the PDF format uses the PDFFRenderOption class. All other output formats use the RenderOption class.

Use the setRenderOption() method on an IRenderTask or an IRunAndRenderTask object. This method performs no function on an IRunTask object because this task does not render to an output format. To support the rendering options that are specific to each output format, BIRT provides extensions of the org.eclipse.birt.report.engine.api.RenderOption class, which implement the IRenderOption interface.

After creating the rendering option object, call the task's setRenderOption() method. This method takes a IRenderOption object as an argument. Listing 14-10 includes code that sets the rendering option for HTML.

To set up the options for HTML rendering, you use an HTMLRenderOption object. To set up the options for PDF rendering, you use a PDFFRenderOption object. To apply a rendering setting, use setter methods on the rendering option object. These classes also have getter methods that retrieve render settings. Common option settings are a base URL for hyperlinks, an action handler, setting an image handler, output format, and supported image output formats. They also support sending the output to a stream or a file.

The supported image formats setting is used for extended report items such as charts or custom extended items. The final rendering environment for the report, such as the browser for a report in HTML format, affects this option value. To set the supported formats, use setSupportedImageFormats() with a String that contains a list of the supported image formats as its argument. The image formats are standard types, such as BMP, GIF, JPG, and SVG. Semicolons (;) separate the items in the list. The method getSupportedImageFormats() returns a String of the same format.

Setting up HTML rendering options

Before generating an HTML report that uses images on disk or creates images or charts in a report, the application must provide additional settings. The HTMLRenderOption class provides many ways to customize the emitted

HTML. The following list describes some of the commonly used options and how the options interact with each other:

- **Image directory.** Many reports include images, as either static images or dynamically created images, such as charts. HTML reports place all these images in a defined location. To set this location, call the `HTMLRenderOption.setImageDirectory()` method.
- **Base image URL.** If you use web deployment, such as a WAR file for your reporting application, the images in a report that are created in the image directory must be available to a user's web browser. You provide the engine with the URL to prefix to the path to the images. Use the `HTMLRenderOption.setBaseImageURL()` method to set the path, as shown in Listing 14-3.
- **Image handler.** Use the `HTMLRenderOption.setImageHandler()` method to set up an image handler, as described earlier in this chapter.
- **Embeddable HTML.** Call the `HTMLRenderOption.setEmbeddable()` method with the argument, true, to set this option. The embeddable produced HTML does not have `<html>` and `<body>` tags.
- **Right to left generation.** The HTML emitter can generate content right to left. To set this option, call the `HTMLRenderOption.setHtmlRtLFlag()` method with the argument, true.
- **Title tag.** Use the `HTMLRenderOption.setHtmlTitle()` to write a `<title>` tag to the produced HTML. This title appears in the title bar of the browser and in the tab on a multi-tabbed browser window.
- **Master page content.** When building BIRT reports the developer can use the Master Page to set up page sizes, header and footers. These settings affect the produced HTML. To suppress the master page content, call the `HTMLRenderOption.setMasterPageContent()` with the argument, false. The master page content setting affects the results of setting paginated HTML and fixed layout.
- **Floating footer.** By default, the master page footer appears directly below the content of each HTML page. Pass the argument, false, to the `HTMLRenderOption.setPageFooterFloatFlag()` method to force the footer to the bottom of a fixed-size master page. This setting adds a `div` statement with a `height` attribute to the produced HTML.
- **Master page margins.** Master page margins affect the appearance of the HTML output if the report uses fixed layout or if you pass the value, true, to the `HTMLRenderOption.setOutputMasterPageMargins()` method.
- **Paginated HTML.** When rendering the report in HTML, the engine supports paginating the results. The report design's master page defines the size of the page and the content of its page headers and footers. To display the entire or partial report as a single HTML page, call the `HTMLRenderOption.setHtmlPagination()` method with the argument, false. In this case, setting a render task page range of 5-7 renders pages 5

to 7 as one HTML page. Header and footer information appear several times within this HTML page if you choose to display master page content. For your application to support pagination, you need to set up pagination controls similar to the example Web Viewer.

- Fixed or automatic layout. By default, BIRT generates tables with columns that are not fixed in size. When the user changes the width of the browser window, the column widths adjust accordingly. BIRT also supports fixed layout, in which column widths do not change. Fixed layout produces reports with a consistent layout, but this choice can increase rendering time. The `table-layout:fixed` attribute is applied to all tables. Appropriate `div` statements set column widths and padding. The `div` settings are also applied if master page content is used. To change this setting, use the `HTMLRenderOption.setLayoutPreference()` method. Pass the value, `IHTMLRenderOption.LAYOUT_PREFERENCE_AUTO` or `IHTMLRenderOption.LAYOUT_PREFERENCE_FIXED`, as its argument.
- Style calculation. The `HTMLRenderOption.setEnableAgentStyleEngine()` method provides the developer control over how the styles are calculated. Passing a value of true to this method emits the styles directly to the produced HTML. In this case, the browser performs the style calculations. Passing a value of false to the method causes the emitter to use the BIRT style engine to do the calculations. Like the fixed or automatic layout setting, this setting affects the consistency of the report's appearance.
- Base URL. Most BIRT report items have a configurable hyperlink property. When building this hyperlink in the report designer, the report developer can use a URL, an internal bookmark, or a drill-through operation. The report design stores the results of the design session. The `HTMLRenderOption.setBaseUrl()` method defines the base URL to prefix to the hyperlinks built within the designer. Use this setting for applications deployed in a web environment.
- Action handler. The emitter handles hyperlink, bookmark, and drill-through actions. When rendering the report as HTML, BIRT uses the `HTMLActionHandler` class to determine how to build the hyperlink for these actions. The `HTMLActionHandler` class implements the `IHTMLActionHandler` interface. If the default `HTMLActionHandler` does not meet your needs, you can use a custom implementation of the `IHTMLInterface`. To set up the new action handler, use the `HTMLRenderOption.setActionHandler()` method.

Setting up the PDF rendering options

To generate a PDF report that needs to embed fonts in a report or uses fonts from non-standard locations, the application must provide PDF rendering settings. The `PDFRenderOption` class provides the following settings:

- Font directory. If your deployment platform uses fonts from a custom location, set the path to their location with the `setFontDirectory()` method. This method takes a `String` argument, which is a list of directories separated by semicolon characters.

- Embedding fonts. To embed custom fonts in the PDF document, call `PDFRenderOption.setEmbeddedFont()` with an argument of true.

How to configure properties for a report in HTML format

The code sample in Listing 14-10 shows the use of rendering options on an `IRunAndRenderTask` object to set report parameter values, the output format of the report, and the output file name. The variable, `engine`, is a `ReportEngine` object. The variable, `runnable`, is an object of type `IReportRunnable`. The variable, `name`, is the name of the report design.

Listing 14-10 Configuring properties on an `IRunAndRenderTask` object

```
// Create a run and render task object.  
IRunAndRenderTask task =  
    engine.createRunAndRenderTask( runnable );  
  
// Set values for all parameters in a HashMap, parameterValues  
task.setParameterValues( parameterValues );  
  
// Validate parameter values.  
boolean parametersAreGood = task.validateParameters( );  
  
// Set the name of an output file.  
HTMLRenderOption options = new HTMLRenderOption( );  
String output = name.replaceFirst( ".rptdesign", ".html" );  
options.setOutputFileName( output );  
options.setImageDirectory( "image" );  
options.setHtmlRtLFlag( true );  
options.setEmbeddable( false );  
  
// Apply the rendering options to the task.  
task.setRenderOption( options );  
  
// Run and close the task  
task.run();  
task.close();
```

Generating the formatted output programmatically

To generate a report, the application must call the `run()` method on an `IRunAndRenderTask` or an `IRunTask` object. The application must handle the `EngineException` that `run()` can throw. After generating the report, the application can reuse the report engine to generate further reports. If your application only generates a single report, destroy the engine after performing the report generation.

How to generate a report

The code sample in Listing 14-11 generates a report, then destroys the report engine. The variable, `engine`, is a `ReportEngine` object. The variable, `task`, is an `IRunAndRenderTask` or an `IRunTask` object. The variable, `name`, is the name of the report design. The variable, `output`, is the name of the output file.

Listing 14-11 Generating a report from a report design or a report document

```
try {
    task.run( );
    System.out.println( "Created Report " + output + "." );
}
catch ( EngineException e1 ) {
    System.err.println( "Report " + name + " run failed." );
    System.err.println( e1.toString( ) );
}
engine.destroy( );
```

Accessing the formatted report

When you generate a report document as a file on disk, you can access the report in the same way as any other file. For example, you open HTML documents in a web browser and PDF documents using Adobe Reader. If you send the report to a stream, the stream must be able to process the information.

Cancelling a running report task

The BIRT Report Engine supports checking the status of any engine task and cancelling that task. Typically, you use a separate thread to perform these actions. To check the status of a running report, use the `IEngineTask.getStatus()` method, which returns one of the following values:

- `IEngineTask.STATUS_NOT_STARTED`. The task has not started.
- `IEngineTask.STATUS_RUNNING`. The task is currently running.
- `IEngineTask.STATUS_SUCCEEDED`. The task completed with no errors.
- `IEngineTask.STATUS_FAILED`. The task failed to execute.
- `IEngineTask.STATUS_CANCELLED`. The task was cancelled.

To cancel a running task call the `IEngineTask.cancel()` method. All the tasks that run and render reports implement and extend this interface. If you cancel a task that is running a report, the task stops generating the report content. If the task has started a query on a database, the database continues to run the query to completion.

How to cancel a running report task

- 1 Define a thread class as an inner class to cancel the report. In Listing 14-12, the name of this class is `CancelReport`.
- 2 Within the code that creates the task, create an instance of the `CancelReport` class, as shown in the following line:

```
CancelReport cancelThread = new
    CancelReport( "cancelReport", task);
```

Listing 14-12 Defining a class to cancel an engine task

```
private class CancelReport extends Thread
{
    private IEngineTask eTask;
    public CancelReport( String myName, IEngineTask task ) {
        super( myName );
        eTask = task;
    }
    public void cancel( ) {
        try {
            Thread.currentThread( ).sleep( 100 );
            eTask.cancel( );
            System.out.println( "#### Report cancelled #####" );
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }
}
```

- 3 Test for a condition, then use the CancelReport object to cancel the report, as shown in the following line:

```
cancelThread.cancel( );
```

Programming with a report design

A reporting application typically generates a report from a report design. In this type of reporting application, you typically develop a report design and include the design along with your application at deployment time. Any changes to the generated report depend on the values of report parameters and the data from the data set. To access the report design, the application uses an IReportRunnable object.

Sometimes business logic requires changes to the report design before generating the report. You can make some changes through using parameters and scripting. Other changes can only occur through modification of the report design itself.

A reporting application can make changes to the report design and the ROM elements that make up the design. To access the structure of the report design, the application obtains a ReportDesignHandle object from the design engine. To access the design engine, an application must first instantiate a report engine, as in any other reporting application.

The ReportDesignHandle object provides access to all properties of the report design and to the elements that the report design contains. The model API provides handle classes to access all ROM elements. For example, a

GridHandle object provides access to a grid element in the report design. All ROM element handles, including the report design handle, inherit from DesignElementHandle. Report items inherit from ReportElementHandle and ReportItemHandle.

After making changes to a report design or its elements, the application can write the result to a stream or a file. The report engine can then open an IReportRunnable object on the resulting design and generate a report.

An application typically accesses the items in a report design to perform one of the following tasks:

- Modify an existing report design programmatically to change the contents and appearance of the report output. An application can modify page characteristics, grids, tables, and other report items in the design, the data source, and the data set that extracts data from a data source.
- Build a report design and generate report output entirely in an application without using BIRT Report Designer.

A reporting application can access and modify the structures in a template or a library file in the same way as the structures in a report design. The techniques described in the rest of this chapter are applicable to these files as well as to report designs. A template has identical functionality to a report design. For this reason, the ReportDesignHandle class provides access to a template. The LibraryHandle class provides access to a library. Both these classes derive from the ModuleHandle class, which provides the fields and methods for the common functionality, such as accessing elements in the file.

The package that contains the classes and interfaces to work with the items in a report design, library, or template is org.eclipse.birt.report.model.api.

About BIRT model API capabilities

A report developer can write an application that creates and modifies a report design programmatically. The BIRT model API has the same capabilities as BIRT Report Designer. For example, the following list shows some of the ways in which you can use the BIRT model API to manipulate a report design programmatically:

- Adding a report item to a report design:
 - Add a simple report item such as a data item, label, or image. Set the value to display in the new report item, such as the expression of a data item or the text in a label item.
 - Create a complex item such as a grid, table, or list. Add other items into the grid, table, or list.
- Changing the properties of a report item in a report design:
 - Change the data set bound to a table or list.
 - Change the expression or other property of a report item.

- Format a report item, changing the font, font color, fill color, format, alignment, or size.
- Changing the structure of a report design:
 - Add a report parameter.
 - Add or delete a group or column in a table or list.
- Modifying non-visual elements in a report design:
 - Set a design property such as a report title, author, wallpaper, or comment.
 - Set a page property, such as height, width, or margins.
 - Specify a data source for a data set.

Opening a report design programmatically for editing

To access a report design and its contents, the application must instantiate a report engine, then use a ReportDesignHandle object. You instantiate a ReportDesignHandle by calling a method on another class, such as the model class, SessionHandle, or the report engine interface, IReportRunnable.

The SessionHandle object manages the state of all open report designs. Use a SessionHandle to open, close, create report designs, and to set global properties, such as the locale and the units of measure for report elements. The SessionHandle can open a report design from a file or a stream. Create the session handle only once. BIRT supports only a single SessionHandle for a user of a reporting application.

Configuring the design engine to access a design handle

The DesignEngine class provides access to all the functionality of the Report Object Model (ROM) in the same way that the ReportEngine class provides access to report generation functionality. To create a DesignEngine object, you first create a DesignConfig object to contain configuration settings for the design engine such as the BIRT home location. The BIRT home is the same location that the report engine uses. The DesignConfig object sets up custom access to resources and custom configuration variables for scripting.

You use a factory service for creating a DesignEngine in the same way as for a ReportEngine. After setting all the required configuration properties, create the design engine with the Platform.createFactoryObject() and the IDesignEngineFactory.createDesignEngine() methods. The DesignConfig object is used in this process. If your application has already started the platform in order to set up a report engine, you must use this instance to create the design engine. If your application runs in an RCP environment, do not start the platform. See “Configuring the report engine” earlier in this chapter for more information about setting BIRT home and starting the platform.

Create the SessionHandle object by calling the method, newSessionHandle() on the DesignEngine object. To open the report design, call the method, openDesign(), on the SessionHandle object. This method takes the name of the report design as an argument and instantiates a ReportDesignHandle.

How to open a report design for editing

The code sample in Listing 14-13 creates a DesignEngine object, which it uses to create a SessionHandle object. The code then uses the SessionHandle object to open a report design.

Listing 14-13 Opening a report design for editing

```
// Create a design engine configuration object.  
DesignConfig dConfig = new DesignConfig();  
dConfig.setBIRTHome  
    ( "C:/Program Files/birt-runtime-2_2_1/ReportEngine" );  
// Start the platform for a non-RCP application  
Platform.startup( dConfig );  
IDesignEngineFactory factory =  
    ( IDesignEngineFactory ) Platform.createFactoryObject  
    ( IDesignEngineFactory.EXTENSION DESIGN ENGINE FACTORY );  
IDesignEngine dEngine = factory.createDesignEngine( dConfig );  
// Create a session handle, using the system locale.  
SessionHandle session = dEngine.newSessionHandle( null );  
// Create a handle for an existing report design.  
String name = "./SimpleReport.rptdesign";  
ReportDesignHandle design = null;  
try {  
    design = session.openDesign( name );  
} catch (Exception e) {  
    System.err.println( "Report " + name +  
        " not opened!\nReason is " + e.toString() );  
    return null;  
}
```

Using an IReportRunnable object to access a design handle

You can also open a report design from an IReportRunnable object by using the getDesignHandle() method. The ReportDesignHandle object provides access to the design opened by the report engine. You use a design engine to access the elements in this report design in the same way as for any other design handle.

Using a report item in a report design

A report item is a visual element in the report design. Typically, a report developer adds a report item to the design in BIRT Report Designer by dragging an item from the palette to the layout editor. Sometimes you need to change the properties of certain report items in the design before running the report. An application uses methods on the ReportDesignHandle class to

access a report item either by name or from a list of items in a slot in a container report item.

A slot is a logical component of a report item. For example, a table element has four slots: Header, Detail, Footer, and Groups. In turn, each of these slots can have further slots. Each slot has zero or more members of the appropriate report item type. For example, the Header, Detail, and Footer slots all contain elements of type RowHandle. RowHandle has a Cell slot that contains all the cells in the row. For a visual representation of the slots in an individual report item, see the Outline view in BIRT Report Designer.

Accessing a report item by name

To make a report item accessible by name, the item must have a name. A report developer can set the name in BIRT Report Designer or programmatically by using the item's `setName()` method. To find a report item by name, use the `findElement()` method. This method returns a `DesignElementHandle` object. All report items derive from this class.

Accessing a report item by iterating through a slot

To access a report item through the report design's structure, the application first gets the slot handle of the report body by calling the `getBody()` method. This slot handle holds the top-level report items in the report design. For example, consider a simple report structure that has three top-level items: a grid of header information, a table containing grouped data, and a label that displays a report footer. Figure 14-2 shows its outline view in BIRT Report Designer.

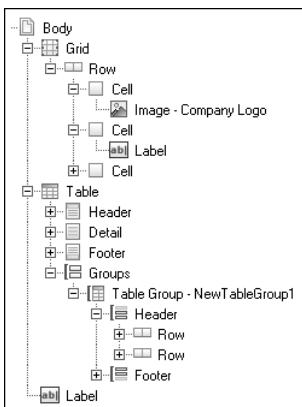


Figure 14-2 Slots in a report design

To access the top-level items in this report design, you iterate over the contents of the body slot handle. These contents all derive from `DesignElementHandle`. To access the iterator for a slot handle, call `SlotHandle.iterator()`. Each call to `Iterator.getNext()` returns a report item. Alternatively, to access a report item at a known slot index, call `SlotHandle.get()`. The slot index number is zero-based. `ReportDesignHandle`

also provides finder methods, which can access an item or other report element by name.

Examining a report item programmatically

To examine a report item, check the class of the report item, cast the object to its actual class, then call methods appropriate to that class. For example, the class of a label element handle is LabelHandle. To get the text that the label displays, call LabelHandle.getText().

Some report items, such as a label or a text element, are simple items. Other items, such as a grid or a table element, are structured items. You can access properties for the whole of a structured item in the same way as for a simple item.

You can also iterate over the contents of the structured item. For example, use this technique to determine the contents of a cell in a table. To access the contents of a structured item, you call a method to retrieve the slot handle for rows or columns. For example, to access the RowHandle objects that make up a table element's footer, call TableHandle.getFooter(). Table and list elements also have a slot for groups. Like the body slot handle, the slot handles for the contents of structured report items can contain zero, one, or multiple elements.

Accessing the properties of a report item

To provide information about report items, each class has getter methods specific to the report item type. For example, an image element handle, ImageHandle, has the getURI() method. This method returns the URI of an image referenced by URL or file path. The DesignElementHandle class and other ancestor classes in the hierarchy also provide generic getter methods, such as getName().

Some properties of a report item are simple properties, with types that are Java types or type wrapper classes. An example of this type of property is the name property, which is a String object. Some of these properties, like name, have arbitrary values. Other simple properties have restricted values from a set of BIRT String constants. The interface, DesignChoiceConstants in the org.eclipse.birt.report.model.api.elements package, defines these constants. For example, the image source property of an image element can have only one of the values, IMAGE_REF_TYPE_EMBED, IMAGE_REF_TYPE_EXPR, IMAGE_REF_TYPE_FILE, IMAGE_REF_TYPE_NONE, or IMAGE_REF_TYPE_URL.

Other properties are complex properties and the getter method returns a handle object. For example, the DesignElementHandle.getStyle() method returns a SharedStyleHandle object and ReportItemHandle.getWidth() returns a DimensionHandle object. The handle classes provide access to complex properties of a report item, as described later in this chapter. These classes provide getter methods for related properties. For example, StyleHandle classes provide access to font and background color.

How to access a report item by name

The code sample in Listing 14-14 finds an image item by name, checks its type, then examines its URI. The variable, `design`, is a `ReportDesignHandle` object.

Listing 14-14 Finding a report item with a given name

```
DesignElementHandle logoImage =
    design.findElement( "Company Logo" );
// Check for the existence of the report item.
if ( logoImage == null) {
    return null;
}
// Check that the report item has the expected class.
if ( !( logoImage instanceof ImageHandle ) ) {
    return null;
}
// Retrieve the URI of the image.
String imageURI = ( (ImageHandle ) logoImage ).getURI( );
return imageURI;
```

How to use the report structure to access a report item

The code sample in Listing 14-15 finds an image item in a grid, checks its type, then examines its URI. Use this technique for generic code to navigate a report design structure or if you need to find an item that does not have a name. The variable, `design`, is a `ReportDesignHandle` object.

Listing 14-15 Navigating the report structure to access a report item

```
// Instantiate a slot handle and iterator for the body slot.
SlotHandle shBody = design.getBody( );
Iterator slotIterator = shBody.iterator( );

// To retrieve top-level report items, iterate over the body.
while (slotIterator.hasNext( )) {
    Object shContents = slotIterator.next( );

    // To get the contents of the top-level report items,
    // instantiate slot handles.
    if ( shContents instanceof GridHandle ) {
        GridHandle grid = ( GridHandle ) shContents;
        SlotHandle grRows = grid.getRows( );
        Iterator rowIterator = grRows.iterator( );

        while ( rowIterator.hasNext( )) {
            // Get RowHandle objects.
            Object rowSlotContents = rowIterator.next( );
            // To find the image element, iterate over the grid.
            SlotHandle cellSlot =
                ( ( RowHandle ) rowSlotContents ).getCells( );
            Iterator cellIterator = cellSlot.iterator( );
```

Modifying a report item in a report design programmatically

To set the simple properties of report items, each class has setter methods specific to the report item type. For example, an image element handle, ImageHandle, has the setURI() method. This method sets the URI of an image referenced by URL or file path. The DesignElementHandle class and other ancestor classes in the hierarchy also provide generic setter methods, such as setName(). Setter methods throw exceptions, such as NameException, SemanticException, and StyleException.

To set attributes of a complex property, such as a style, you must call methods on a handle object, as described later in this chapter. These classes provide setter methods for related properties. For example, `StyleHandle` classes provide access to style properties, such as font and background color.

How to change a simple property of a report item

The code sample in Listing 14-16 uses a method on LabelHandle to change the text in a label. The variable, `design`, is a `ReportDesignHandle` object. This sample accesses the label by name. You can also access a report item by navigating the report structure.

Listing 14-16 Changing the text property of a label report item

```
// Access the label by name.  
LabelHandle headerLabel =  
    ( LabelHandle ) design.findElement( "Header Label" );  
try {  
    headerLabel.setText( "Updated " + headerLabel.getText( ) );  
} catch ( Exception e ) {  
    // Handle the exception  
}
```

Accessing and setting complex properties

Complex properties use BIRT handle objects to access data structures. For example, a DimensionHandle object provides access to size and position properties, such as the absolute value and the units of the width of a report item. Some String properties on a handle object, such as font style and text alignment on a style handle, have restricted values defined by constants in the org.eclipse.birt.report.model.api.elements.DesignChoiceConstants interface. For example, the font style property can only have values of FONT_STYLE_ITALIC, FONT_STYLE_NORMAL, or FONT_STYLE_OBLIQUE.

Using a property handle

To access complex properties, you use getter methods on the report item. For example, to access the width of a report item, call the method ReportItemHandle.getWidth(). This method returns a DimensionHandle object. To work with complex properties, you use getter and setter methods on the handle object. For example, to get and set the size of a dimension, you use DimensionHandle.getMeasure() and DimensionHandle.setAbsolute(), respectively. When you set a value on a complex property, the change to the handle object affects the report item straight away. You do not call another setter method on the report item itself.

Using styles on a report item

The StyleHandle class provides access to many fundamental properties of a report item, such as margin size, text alignment, background color, borders, font, and so on. StyleHandle provides a full set of getter methods for each style property. For simple properties, StyleHandle provides setter methods. To modify complex properties, you use setter methods on the property handle object, not on the style handle itself. A report item can use two styles; a private style and a shared style. The handle classes for these styles are PrivateStyleHandle and SharedStyleHandle, respectively. Both classes derive from StyleHandle.

A private style contains the settings that the report developer chose in the property editor when designing the report. Shared styles appear in the Outline view in BIRT Report Designer. You use shared styles to apply the same appearance to multiple items in a report design. Changes to a shared

style affect all report items that use the style. Style settings in a private style override the settings in a shared style.

How to change a complex property of a report item

The code sample in Listing 14-17 shows how to use PrivateStyleHandle and ColorHandle objects to change the background color of a label. The variable, design, is a ReportDesignHandle object. This sample accesses the label by name. You can also access a report item by navigating the report structure.

Listing 14-17 Changing a complex property of a report item

```
// Access the label by name.  
LabelHandle headerLabel =  
    ( LabelHandle ) design.findElement( "Header Label" );  
try {  
    // To prepare to change a style property, get a StyleHandle.  
    StyleHandle labelStyle = headerLabel.getPrivateStyle();  
    // Update the background color.  
    ColorHandle bgColor = labelStyle.getBackgroundColor();  
    bgColor.setRGB( 0xFF8888 );  
} catch ( Exception e ) {  
    // Handle the exception  
}
```

Understanding property structure objects

Complex property structures represent many optional report element features within BIRT. For example, computed columns, highlight rules, sort keys, and table of contents entries are all optional complex properties of report elements. The classes for these properties all derive directly or indirectly from org.eclipse.birt.report.model.core.Structure. You can access existing structure property objects on a report element or create new ones.

Using an existing property structure object

You use the property handle, which is an object of class PropertyHandle, to apply a new structure object to a report element's property or to modify an existing property structure. You access the property handle with the DesignElementHandle.getPropertyHandle(), which is available on all report elements. This method takes a String argument that defines the property to access. Constants with names of the form XXX_PROP define the available properties. The property handle object provides access to one or more property structure objects. Use the PropertyHandle getter methods to access existing property structures.

Typically, the structure class provides a few setter methods for key properties. You set other properties by calling the object's setProperty() method. This method takes two arguments: a String that represents the property and an object that is the property's value. Constants defined on the structure class provide the values for the property String.

How to set values on an existing property structure object

The code sample in Listing 14-18 shows how to change the sort direction value on a table item's sort key.

Listing 14-18 Changing the sort direction on a table

```
void modSortKey( TableHandle th ) {
    try {
        SortKeyHandle sk;
        PropertyHandle ph =
            th.getPropertyHandle( TableHandle.SORT_PROP );
        sk = ( SortKeyHandle ) ph.get( 0 );
        sk.setDirection
            ( DesignChoiceConstants.SORT_DIRECTION_DESC );
    } catch (Exception e) { e.printStackTrace( ); }
}
```

Creating a property structure object

The design engine class, StructureFactory, provides static methods for creating structure property objects. Most StructureFactory methods take the form createXXX, where XXX is the structure to create. After you create the new structure object, set its simple or complex properties using its setter methods. Report element handles provide methods to add key structure properties to their property handles. For example, you add a filter condition to a data set object by using the addFilter() method. To add other properties to a report element, use the method, PropertyHandle.addItem().

How to create a complex property object

The code sample in Listing 14-19 shows how to create a highlight rule object and apply that rule to a row handle. The variable, rh, is a RowHandle object. Because RowHandle does not have a specific method to add a highlight rule property, the code uses the PropertyHandle.addItem() method.

Listing 14-19 Adding a highlight rule to a table or grid row

```
try {
    HighlightRule hr = StructureFactory.createHighlightRule( );
    hr.setOperator( DesignChoiceConstants.MAP_OPERATOR_GT );
    hr.setTestExpression( "row[\"CustomerCreditLimit\"]" );
    hr.setValue1( "100000" );
    hr.setProperty
        ( HighlightRule.BACKGROUND_COLOR_MEMBER, "blue" );

    PropertyHandle ph =
        rh.getPropertyHandle( StyleHandle.HIGHLIGHT_RULES_PROP );
    ph.addItem( hr );
} catch ( Exception e ){ e.printStackTrace(); }
```

The code samples in Listing 14-20 provide further examples of using the structure factory to create complex properties.

Listing 14-20 Various structure factory examples

```
// Use the structure factory to add a sort key to a table.  
void addSortKey( TableHandle th ) {  
    try {  
        SortKey sk = StructureFactory.createSortKey( );  
        sk.setKey( "row[\"CustomerName\"]" );  
        sk.setDirection  
            ( DesignChoiceConstants.SORT_DIRECTION_ASC );  
  
        PropertyHandle ph =  
            th.getPropertyHandle( TableHandle.SORT_PROP );  
        ph.addItem( sk );  
    } catch ( Exception e ) { e.printStackTrace( ); }  
}  
  
// Add a column binding to a table.  
void addColumnBinding( TableHandle th ) {  
    try {  
        ComputedColumn cs1;  
        cs1 = StructureFactory.createComputedColumn( );  
        cs1.setName( "CustomerName" );  
        cs1.setExpression( "dataSetRow[\"CUSTOMERNAME\"]" );  
        th.addColumnBinding( cs1, false );  
    } catch ( Exception e ) { e.printStackTrace( ); }  
}  
  
// Add a filter condition to a table.  
void addFilterCondition( TableHandle th ) {  
    try {  
        FilterCondition fc = StructureFactory.createFilterCond();  
        fc.setExpr( "row[\"CustomerCountry\"]" );  
        fc.setOperator(DesignChoiceConstants.MAP_OPERATOR_EQ);  
        fc.setValue1("USA");  
        PropertyHandle ph =  
            th.getPropertyHandle( TableHandle.FILTER_PROP );  
        ph.addItem( fc );  
    } catch ( Exception e ) { e.printStackTrace( ); }  
}  
  
// Add a filter condition to a data set.  
void addFilterCondition( OdaDataSetHandle dh ) {  
    try {  
        FilterCondition fc = StructureFactory.createFilterCond();  
        fc.setExpr( "row[\"COUNTRY\"]" );  
        fc.setOperator( DesignChoiceConstants.MAP_OPERATOR_EQ );  
        fc.setValue1( "USA" );  
        // Add the filter to the data set.  
        dh.addFilter( fc );  
    } catch ( Exception e ) { e.printStackTrace( ); }  
}
```

```

// Add a hyperlink to a label.
void addHyperlink( LabelHandle lh ) {
    try {
        Action ac = StructureFactory.createAction( );
        ActionHandle actionHandle = lh.setAction( ac );
        actionHandle.setURI( "http://www.google.com" );
        actionHandle.setLinkType
            ( DesignChoiceConstants.ACTION_LINK_TYPE_HYPERLINK );
    } catch ( Exception e ) { e.printStackTrace( ); }
}

// Add a table of contents entry to a data item.
void addToc( DataItemHandle dh ) {
    try {
        TOC myToc = StructureFactory.createTOC
            ( "row[\"CustomerName\"]" );
        dh.addTOC( myToc );
    } catch ( Exception e ) { e.printStackTrace( ); }
}

// Add an embedded image to the report design.
void addImage( ) {
    try {
        EmbeddedImage image =
            StructureFactory.createEmbeddedImage( );
        image.setType
            ( DesignChoiceConstants.IMAGE_TYPE_IMAGE_JPEG );
        image.setData( load( "logo3.jpg" ) );
        image.setName( "mylogo" );
        designHandle.addImage( image );
    } catch ( Exception e ) { e.printStackTrace( ); }
}

// Load the embedded image from a file on disk.
public byte[ ] load( String fileName ) throws IOException
{
    InputStream is = null;
    is = new BufferedInputStream
        ( this.getClass( ).getResourceAsStream( fileName ) );
    byte data[ ] = null;
    if ( is != null ) {
        try {
            data = new byte[is.available( )];
            is.read( data );
        }
        catch ( IOException e1 ) {
            throw e1;
        }
    }
    return data;
}

```

Adding a report item to a report design programmatically

A reporting application can use a simple report design or a template to create more complex designs. The application can add extra report items to the design's structure based on external conditions. For example, based on the user name of the user requesting generation of a report, you can add extra information to the report for that category of user. If you create a design entirely with the API, you use the same techniques to add content to it.

The class that creates new elements, such as report items, in a report design is ElementFactory. This class provides methods of the form, newXXX(), where XXX is the report item or element to create. The method newElement() is a generic method that creates an element of any type. To access the element factory, call the ReportDesign.getElementFactory() method.

You can place new report items at the top level of the report design, directly in the body slot, within containers such as a cell in a table or grid, or on the master page. You can add a simple item, such as a label, or complex items, such as a table with contents in its cells. Wherever you add the new report item, the location is a slot, such as the body slot of the report design or a cell slot in a row in a table. To add a report item to a slot, you use one of the SlotHandle.add() methods. The method has two signatures that support adding the report item to the end of a slot, or to a particular position in a slot.

Table and list elements are container items that iterate over the rows that a data set provides. For these report items to access the data rows, you must bind them to a data set. The table or list element provides data rows to the report items that it contains. For this reason, you usually bind only the container item to a data set, as described later in this chapter.

How to add a grid item and label item to a report design

The code sample in Listing 14-21 creates a grid item, then adds a label item to one of the cells in the grid. An application can create any other report item in a similar manner. The variable, design, is a ReportDesignHandle object.

Accessing a data source and data set with the API

This section shows how to use ROM elements that are not report items. To use other ROM elements, such as the libraries that the report design uses, you employ similar techniques.

You access the report design's data sources and data sets from methods on the ReportDesignHandle instance, in a similar way to other report elements. The model classes that define a data source and data set are DataSourceHandle and DataSetHandle, respectively. A data set provides a report item such as a table with data from a data source. For a report item to access the data set, use the setDataSet() method.

You can use a finder method on the report design handle to access a data source or data set by name. The finder methods are findDataSource() and findDataSet(), respectively. Alternatively, to access all the data sources or

data sets, you can use a getter method that returns a slot handle. The getter methods are `getDataSources()` and `getDataSets()`, respectively. To access the individual data sources or data sets in a slot handle, you iterate over the contents of the slot handle in the same way as for any other slot handle.

Listing 14-21 Adding a container item to the body slot

```
// Instantiate an element factory.  
ElementFactory factory = design.getElementFactory();  
try {  
    // Create a grid element with 2 columns and 1 row.  
    GridHandle grid = factory.newGridItem( "New grid", 2, 1 );  
    // Set a simple property on the grid, the width.  
    grid.setWidth( "50%" );  
    // Create a new label and set its properties.  
    LabelHandle label = factory.newLabel( "Hello Label" );  
    label.setText( "Hello, world!" );  
  
    // Get the first row of the grid.  
    RowHandle row = ( RowHandle ) grid.getRows( ).get( 0 );  
    // Add the label to the second cell in the row.  
    CellHandle cell = ( CellHandle ) row.getCells( ).get( 1 );  
    cell.getContent( ).add( label );  
  
    // Get the body slot. Add the grid to the end of the slot.  
    design.getBody( ).add( grid );  
} catch ( Exception e ) {  
    // Handle any exception  
}
```

About data source classes

`DataSourceHandle` is a subclass of `ReportElementHandle`. You get and set report item properties for a data source in the same way as for any other report element. `DataSourceHandle` also provides methods to access the scripting methods of the data source.

The two subclasses of `DataSourceHandle`, `OdaDataSourceHandle` and `ScriptDataSourceHandle`, provide the functionality for the two families of BIRT data sources. For more information about ODA data sources, see the Javadoc for the ODA API, in Open Data Access (ODA) 3.0.0 API Reference. The scripting methods for a scripted data source fully define the data source, as described earlier in this book.

About data set classes

`DataSetHandle` is a subclass of `ReportElementHandle`. You get and set properties for a data set in the same way as for any other report element. `DataSetHandle` also provides methods to access properties specific to a data set, such as the data source, the data set fields, and the scripting methods of the data set.

The two subclasses of `DataSetHandle`, `OdaDataSetHandle` and `ScriptDataSetHandle`, provide the functionality for the two families of BIRT data sets. For more information about ODA data sets, see the Javadoc for the ODA API.

Using a data set programmatically

Typically, a reporting application uses data sets and data sources already defined in the report design. You can use the data set's `setDataSource()` method to change the data source of a data set. For example, based on the name of the user of the reporting application, you can report on the sales database for a particular geographical region, such as Europe or North America.

Changing the properties of a data set

Changing the properties of a data set requires consideration of the impact on the report design. If you change the data source of a data set, the type of the data source must be appropriate for the type of the data set. You must also be certain that the new data source can provide the same fields as the original data source.

How to change the data source for a data set

The code sample in Listing 14-22 shows how to check for a particular data source and data set in a report design, then changes the data source for the data set. The code finds the data source and data set by name. Alternatively, use the `getDataSets()` and `getDataSources()` methods. Then use the technique for iterating over the contents of a slot handle. The variable, `design`, is a `ReportDesignHandle` object.

Listing 14-22 Modifying a data set

```
// Find the data set by name.  
DataSetHandle ds = design.findDataSet( "Customers" );  
// Find the data source by name.  
DataSourceHandle dso = design.findDataSource( "EuropeSales" );  
  
// Check for the existence of the data set and data source.  
if (dso == null) || ( ds == null ) {  
    System.err.println( "EuropeSales or Customers not found" );  
    return;  
}  
// Change the data source of the data set.  
try {  
    ds.setDataSource( dso );  
} catch ( SemanticException e1 ) { e1.printStackTrace( ); }
```

Changing the data set binding of a report item

You can also use the report item's `setDataSet()` method to set or change the data set used by a report item. If you change the data set used by a report

item, you must ensure that the contents of the report item access only data bindings that are supplied by the new data set. If necessary, you must change the references to data bindings in data elements, text elements, and scripting methods. If the data bindings in the old data set do not match the names or data types of the fields that the new data set provides, you must correct the data bindings before you generate a report from the modified report design.

Use the ReportItemHandle method, columnBindingsIterator(), to iterate over the column bindings that the report item uses. The items in the list are of type ComputedColumnHandle. This class provides methods to access the name, expression, and data type of the column binding.

To access the data set column and expression that a data item uses, call the methods, getResultSetColumn() and getResultSetExpression(). You can compare the data type and name with the result set columns that the data set returns.

How to bind a data set to a table

The code sample in Listing 14-23 shows how to check for a particular data set in a report design, then changes the data set for a table. The code finds the table and data set by name. Alternatively, use slot handles to navigate the design structure. The variable, design, is a ReportDesignHandle object.

Listing 14-23 Binding a data set to a report item

```
// Find the table by name.  
TableHandle table =  
    ( TableHandle ) design.findElement( "Report Data" );  
// Find the data set by name.  
DataSetHandle ds = design.findDataSet( "EuropeanCustomers" );  
// Check for the existence of the table and the data set.  
if (table == null) || ( ds == null ) {  
    System.err.println( "Incorrect report structure" );  
    return;  
}  
// Change the data set for the table.  
try {  
    table.setDataSet( ds );  
} catch (Exception e) {  
    System.err.println( "Could not set data set for table" );  
}
```

Saving a report design programmatically

After making changes to an existing report design or creating a new report design, you can choose to save the design for archival purposes, or for future use. To overwrite an existing report design to which the application has made changes, use the ReportDesignHandle.save() method. To save a new report design or to keep the original report design after making changes, use the ReportDesignHandle.saveAs() method. Alternatively, if you do not need to

save the changes to the report design, call ReportDesignHandle.serialize() method. This method returns an output stream. The report engine can generate a report by opening a stream as an input stream.

If you do not need to make any further changes to the report design, call the method, ReportDesignHandle.close(), to close the report design, as shown in the following code. The variable, design, is a ReportDesignHandle object.

```
design.saveAs( "sample.rptdesign" );
design.close();
```

Creating a report design programmatically

You can build a report design and generate the report output in an application without using BIRT Report Designer. You use the createDesign() method on the session handle class, SessionHandle, to create a report design. You use the other model classes to create its contents.

How to create a new report design

The following code creates a report design:

```
SessionHandle session = DesignEngine.newSession( null );
ReportDesignHandle design = session.createDesign();
```

This page intentionally left blank

15

Programming using the BIRT Charting API

This chapter describes the basic requirements of a charting application and illustrates the use of BIRT charting API classes and interfaces for modifying an existing chart definition and for creating a new chart. This API supports:

- Writing Java applications with many chart types, such as bar charts, pie charts, line charts, scatter charts, area charts, dial charts, and stock charts.
- Customizing many properties of a chart to fit the users' requirements.
- Modifying an existing chart item in a BIRT report design or adding a chart to an existing report design.

This chapter shows how to customize an existing chart and create a new chart within a BIRT application, but describing how to use the charting API in a stand-alone application is beyond the scope of this book. Using a chart in a stand-alone application is not significantly different from using a chart in a BIRT application. For examples of completed charting applications, download the BIRT samples from the BIRT web site.

This chapter discusses only the most important of the more than 400 classes and interfaces in the BIRT charting API. For information about the complete set of charting API classes and interfaces, see the online Javadoc. To view the Javadoc, open BIRT Report Designer and choose Help->Help Contents->BIRT Charting Programmer Reference->Reference->API Reference.

About the chart engine contents

Downloading and extracting the chart engine archive file from the Eclipse BIRT download page creates the following three folders:

- ChartRuntime, which contains the Eclipse plug-ins required to run, render, and edit charts
- ChartSDK, which contains everything you need to create charting applications, including the following components:
 - All the chart run-time plugins and an example plug-in with sample charting applications including a chart viewer
 - An Eclipse Web Tools Platform (WTP) extension for building a web application that uses charts
 - Documentation in the form of online help, context-sensitive help for the user interface components, and Javadoc for the charting API
 - Source code for all the BIRT Chart Engine packages
- DeploymentRuntime, which contains the JAR files that you need to run a charting application outside of Eclipse

About the environment for charting application

The minimum requirements for creating a basic charting application are:

- The BIRT charting run-time engine
- The BIRT run-time engine, for an application that runs within the BIRT report context
- Java JDK 1.5.0 or later
- Numerous Java archive (.jar) files

When developing or running a Java application that incorporates a BIRT chart, you need to include certain JAR files in your Java CLASSPATH. To be certain that your Java CLASSPATH includes all the JAR files your application needs, include the following files in the Java CLASSPATH:

- All the JAR files in chartRuntime/DeploymentRuntime/ChartEngine
- Any custom extension plug-in JAR files that you use

Typically, a charting application does not need the functionality from every JAR files in the ChartEngine folder, but having extra JAR files in your CLASSPATH does not have a negative effect on the performance or size of an application, so the easiest solution is to include them all.

Configuring the chart engine run-time environment

An application using the BIRT Chart Engine can run in the context of the BIRT Report Engine or as a stand-alone application. If you want to deploy your charts in a BIRT report, you use the BIRT Report Engine. If you want to produce and consume charts within a single application, or pass the chart

structure and data to another application, your application is a stand-alone one and does not use the BIRT Report Engine. To configure the environment, set one and only one of the following two system environment variables:

- STANDALONE

Create this variable if your application has no need for any extension plug-ins. The variable does not require a value.

- BIRT_HOME

Create this variable if your application requires the BIRT chart extension plug-ins. Set the value to the directory containing the engine plug-ins.

You can set whichever of these two environment variables you need on the command that runs the application. For example, to set the STANDALONE variable, use a command similar to the following one:

```
java -DSTANDALONE ChartingAp
```

If you get an error about an inability to load OSGi when you run this command, your application is not stand-alone. Instead it requires the BIRT_HOME variable to be set to the location of the run-time engine folder. Use a command line argument similar to the following one to set the variable:

```
-DBIRT_HOME="C:/birt-runtime-2_2_1/ReportEngine"
```

To test your application in the Eclipse workbench, set these arguments in Run→Open Run Dialog→Java Application→Arguments→VM Arguments.

Verifying the environment for charting applications

Listing 15-1 illustrates the most basic charting application it is possible to write. The output of this program is file called myChart.chart, which contains several hundred lines of XML code. This code is a definition of a basic chart with no data, no labels, no series, and no titles. Although this output does not represent a useful chart, by compiling and running the program, you verify that your environment is configured correctly for a charting application.

Listing 15-1 Basic charting application

```
import java.io.*;
import org.eclipse.birt.chart.model.*;
import org.eclipse.birt.chart.model.impl.*;
public class MyFirstChartProg {
    public static void main( String[ ] args ) {
        Chart myChart = ChartWithAxesImpl.create( );
        Serializer si = SerializerImpl.instance( );
        try {
            si.write( myChart, new FileOutputStream( new File
                ( "C:\\\\myChart.chart" ) ) );
        } catch ( IOException e ) { e.printStackTrace( ); }
    }
}
```

Using the charting API to modify an existing chart

A Java program can open an existing BIRT report design file and alter the content of the report before displaying or saving the report. The chapter on programming BIRT describes how to open a report design file using the BIRT engine and model APIs. This section describes how to use the BIRT charting API to modify an existing chart element within the report design. The following sections contain code examples for each step in the process.

The chart element in a report design extends the basic BIRT report item through the `org.eclipse.birt.report.model.api.ExtendedItemHandle` class, so supports the same standard properties and functionality as all other report items. To access the chart itself from the design engine's API, call the `ExtendedItemHandle.getProperty()` method, with an argument of "chart.instance". This call returns an `org.eclipse.birt.chart.model.Chart` object. All chart types implement this interface through one of the classes, `ChartWithAxesImpl`, `ChartWithoutAxesImpl`, and `DialChartImpl` from the `org.eclipse.birt.chart.model.impl` package. You can cast the `Chart` object to the appropriate class when you need to change its properties.

`ChartWithAxesImpl` is the most commonly used chart class. This class supports area, bar, bubble, cone, difference, Gantt, line, pyramid, scatter, stock, and tube charts. `ChartWithoutAxesImpl` supports pie charts.

`DialChartImpl` supports meter charts. Many of these chart types provide subtypes. For example, a bar chart has stacked, percent stacked, and side-by-side subtypes that affect the appearance of a multi-series bar chart.

The key visual components of a chart are moveable areas known as blocks that are defined by the interface `org.eclipse.birt.chart.model.layout.Block`. Blocks include the plot, title, and legend areas. You can change the location of the title and legend areas with respect to the plot area.

The key data component of a chart is the series. The series controls the set of values to plot and how to group those values. Charts display the values from series as data points in a chart with axes and as slices or pointers in a chart without axes. You can define series values statically as a data set or dynamically as a query.

Getting a Chart object from the report design

To get a chart report item from a report design, you first perform the following steps using the BIRT core and model APIs, as described earlier in this book:

- On a `DesignConfig` object, set the BIRT home property to the directory that contains the report engine.
- Start the platform using the configuration object, a design engine factory, and a design engine.
- Use the design engine to create a session handle object.

- Create a design handle object for a report design from the session handle.
- Use the design handle object to access the chart element in the design.

You retrieve a Chart object from the chart item. With this Chart object, you start to access the BIRT Chart Engine classes and use BIRT's charting API.

How to get a Chart object from a report item in a report design

Listing 15-2 illustrates the process of getting a Chart report item. This code assumes that the chart that you want to modify is the first report item in a list and that the list is the first report item in the report.

Listing 15-2 Getting a ReportDesignHandle object and a Chart object

```
DesignConfig dConfig = new DesignConfig();
dConfig.setBIRTHome( "C:/birt-runtime-2_2_1/ReportEngine" );

IDesignEngine dEngine = null;
ReportDesignHandle dHandle = null;

try {
    Platform.startup( dConfig );
    IDesignEngineFactory dFactory = ( IDesignEngineFactory )
        Platform.createFactoryObject( IDesignEngineFactory.
            EXTENSION DESIGN_ENGINE_FACTORY );
    dEngine = dFactory.createDesignEngine( dConfig );
    SessionHandle sessionHandle =
        dEngine.newSessionHandle( ULocale.ENGLISH );
    dHandle = sessionHandle.openDesign( reportName );
} catch ( BirtException e ) { e.printStackTrace(); }

ListHandle li = ( ListHandle )
    dHandle.getBody().getContents().get( 0 );
ExtendedItemHandle eihChart1 = ( ExtendedItemHandle )
    li.getSlot( 0 ).getContents().get( 0 );
Chart chart =
    ( Chart ) eihChart1.getProperty( "chart.instance" );
```

Modifying chart properties

Each of the available chart implementations provides a set of properties. You use the BIRT Chart Engine API to change these properties. All charts support the properties that the Chart interface provides, such as plot, which is the area that contains the chart itself, and dimension, which sets the appearance of the chart as two-dimensional, two-dimensional with depth, or three-dimensional.

The ChartWithAxesImpl class implements the ChartWithAxes interface, which provides properties related to x-, y-, and for some chart types, z-axes. The ChartWithoutAxesImpl class implements the ChartWithoutAxes interface, which provides properties for pie slices. The DialChartImpl class implements the ChartWithoutAxes and DialChart interfaces. DialChart

supports the superimposition property that meter charts use. Each interface provides getter and setter methods for its properties, and other methods if necessary. For example, for the dimension property, the Chart interface provides getDimension(), setDimension(), isSetDimension(), and unsetDimension() methods.

Some properties, like dimension, are simple properties that take a single value. In some cases, these properties accept a restricted set of values that the class defines as static fields. You use the Javadoc to find these values. Other properties, such as a chart's plot or title are complex. A getter method for a complex property returns an object. For example, the Chart.getTitle() method returns an org.eclipse.birt.chart.model.layout.TitleBlock object. The chart interfaces do not provide setter methods for complex properties. When you change the properties of a complex property, the changes take effect on the chart immediately. The following code example shows how to set both simple and complex properties:

```
// Set a chart's appearance to two-dimensional  
chart.setDimension( ChartDimension.TWO_DIMENSIONAL_LITERAL );  
// Set the chart's title  
chart.getTitle( ).getLabel( ).getCaption( ).setValue  
    ( "Orders by Month" );  
// Rotate the text in the chart's title  
chart.getTitle( ).getLabel( ).getCaption( ).getFont( )  
    .setRotation( 5 );
```

Some properties, such as the horizontal spacing of elements within a plot, use values based on the current units of measurement. Call the Chart.setUnits() method to set the units that you prefer, as shown in the following code:

```
chart.setUnits( UnitsOfMeasurement.getLiteral  
    ( UnitsOfMeasurement.PIXELS_LITERAL ));
```

Modifying plot properties

All charts have a plot property, which is the area in a Chart object that contains the chart itself. The plot is an object that implements the org.eclipse.birt.chart.model.layout.Plot interface. Plot defines horizontal and vertical spacing and the client area, which contains the rendering of the chart. The client area is itself a complex property that defines properties such as the background color and outline of the rendered chart. Plot also provides all the properties defined by the Block interface in the same package, such as the background and outline. When you decide to set a property that multiple components provide, you need to determine the best class on which to set the property based on the characteristics of all the classes. The following code illustrates how to get the chart plot and modify its properties:

```
Plot plot = chart.getPlot( );  
plot.getClientArea( ).setBackground  
    ( ColorDefinitionImpl.CREAM( ));  
plot.setHorizontalSpacing( plot.getHorizontalSpacing * 1.2 );  
plot.getOutline( ).setVisible( true );
```

Modifying the legend properties

All charts have a legend property, which is the area in a Chart object that contains the chart legend. For a chart without axes, the legend identifies the slices on a pie chart or pointers on a meter chart. For a chart with axes, the legend identifies the series that display values on the x-axis. Typically, if there is only one x-axis group, you do not make the legend visible. The legend is an object that implements the org.eclipse.birt.chart.model.layout.Legend interface, which extends the Block interface. Legend properties include all the Block properties, plus properties specific to Legend, such as position, title, text, and values. The default position of the legend is to the right of the plot area. Within this position, you can change the location of the legend by setting its anchor property. The following code illustrates how to get the chart legend and modify its properties:

```
Legend legend = chart.getLegend();
legend.getText().getFont().setSize( 16 );
legend.getInsets().set( 10, 5, 0, 0 );
legend.setAnchor( Anchor.NORTH_LITERAL );
```

Modifying the series properties

All charts use series to define the data values to represent. Series definition objects contain the series objects for the chart and standard properties such as the palette of colors that the chart uses to display pie sectors, bars, bubbles, and other markers. You instantiate a SeriesDefinition object by calling SeriesDefinitionImpl.create().

Series objects contain a list of values to plot on the chart. You instantiate a Series object by calling SeriesImpl.create(). You define the set of values for a series either with a query that accesses an external data source or with a data set that is a list of values. The series for an x-axis or for the values that control the number of sectors in a pie can contain non-numeric data values, such as dates or text values. For example, these series can contain the quarters in a range of years, or product codes or countries. The series for a y-axis or for the values that control the size of the sectors in a pie chart must hold numeric values. These series are implementations of specific subinterfaces of the Series interface. Each chart type uses its own series type, to control the chart represents the values. For example, a BarSeries object has riser and riser outline properties to control the appearance of the bars. You can define grouping of these series by setting a query on the series definition object. You can add multiple series to a series definition. The series do not all have to be the same type. For example, you can display a line chart on the same axes as a bar chart. Typically, you add extra series to a y-axis, not to an x-axis.

The following code illustrates how to get a series from an axis and how to change the properties of the series.

```
SeriesDefinition seriesDefX = SeriesDefinitionImpl.create();
seriesDefX = ( SeriesDefinition )
    xAxisPrimary.getSeriesDefinitions().get( 0 );
seriesDefX.getSeriesPalette().shift( 1 );
```

Modifying axes properties

To modify the properties of one or more axes of a chart, you must first cast the Chart object to a type of ChartWithAxes, as shown in the following statement:

```
chart = ( ChartWithAxes ) chart;
```

Listing 15-3 illustrates the technique for getting the axes of a chart and setting their properties.

Listing 15-3 Getting an axis and setting its properties

```
Axis xAxisPrimary = chart.getPrimaryBaseAxes( )[0];
xAxisPrimary.getLabel( ).getCaption( ).getFont( ).setRotation
    ( 45 );
xAxisPrimary.getTitle( ).getCaption( ).setValue( "Months" );
Axis yAxisPrimary =
    chart.getPrimaryOrthogonalAxis( xAxisPrimary );
yAxisPrimary.getMajorGrid( ).setTickStyle
    ( TickStyle.LEFT_LITERAL );
yAxisPrimary.setType( AxisType.LINEAR_LITERAL );
yAxisPrimary.getTitle( ).getCaption( )
    .setValue( "Sales Growth" );
yAxisPrimary.setFormatSpecifier
    ( JavaNumberFormatSpecifierImpl.create( "$" ) );
```

Adding a series to a chart

You can add a new series to a chart. Listing 15-4 illustrates how to create a second series, set some of its properties, assign data to the series, and add the series to an axis.

Listing 15-4 Creating a series, setting its properties, and adding the series to an axis

```
SeriesDefinition seriesDefY = SeriesDefinitionImpl.create( );
seriesDefY.getSeriesPalette( ).update( ColorDefinitionImpl
    .YELLOW( ) );
BarSeries barSeries2 = ( BarSeries ) BarSeriesImpl.create( );
barSeries2.setSeriesIdentifier( "Q2" );
barSeries2.setRiserOutline( null );
barSeries2.getLabel( ).setVisible( true );
barSeries2.setLabelPosition( Position.INSIDE_LITERAL );

// Assign data to the series
Query query2 = QueryImpl.create( "row[\"Value2\"]" );
barSeries2.getDataDefinition( ).add( query2 );
seriesDefY.getSeries( ).add( barSeries2 );
seriesDefY.getQuery( ).setDefinition( "\"Q2\"" );

// Add the new series to the y-axis
yAxisPrimary.getSeriesDefinitions( ).add( seriesDefY );
```

Adding a chart event handler to a charting application

There are two kinds of chart event handlers that you can add to a charting application: a Java event handler or a JavaScript event handler.

Adding a Java chart event handler to a charting application

To add a Java event handler, you must first create a separate Java class file containing your new event handler method or methods. The process for creating a Java event handler class is identical to the process for creating a Java event handler class for any other report item, as described in the chapter on scripting with Java.

To register a Java class in the charting application code, use the setScript() method of the chart instance object, as shown in the following statement.

```
chart.setScript  
  ( "com.MyCompany.eventHandlers.ChartEventHandlers" );
```

In the preceding statement, the string passed to the setScript() method is the fully qualified name of the Java class. Notice that the .class extension is not included in the class name.

Adding a JavaScript chart event handler to a charting application

To add a JavaScript event handler, you must code the script as one long string and pass that string to the setScript() method of the chart instance object. You must include a function for every event handler method of the chart. For example, the Java statement in Listing 15-5 passes a string to chart.setScript() containing event handler scripts for the beforeDrawDataPointLabel event handler method.

Listing 15-5 Adding an event handler script to a bar chart

```
cwaBar.setScript  
  ( "function beforeDrawDataPointLabel"  
    + "(dataPoints, label, scriptContext)"  
    + "{val = dataPoints.getOrthogonalValue( );"  
    + "clr = label.getCaption( ).getColor( );"  
    + "if ( val < -10 ) clr.set( 32, 168, 255 );"  
    + "else if ( ( val >= -10 ) & ( val <=10 ) )"  
    + "clr.set( 168, 0, 208 );"  
    + "else if ( val > 10 ) clr.set( 0, 208, 32 );}"  
  )
```

Line breaks in the JavaScript code are indicated by backslash n (\n), and quotes within the script are indicated by a backslash quote (\"). The JavaScript code in Listing 15-5 consists of several strings concatenated together to form a single string. This technique helps make the script more readable.

Using the charting APIs to create a new chart

You can add a new chart to an existing report design or to a new report design. In either case, your program must perform a series of tasks. Creating a new chart and adding the chart to a report design requires performing the following tasks. If you create a chart with a stand-alone application, you only need to perform the first two tasks.

- **Creating the chart instance object**

The chart instance object contains the properties of the chart, such as its title, its axes, and its series.

- **Setting the properties of the chart instance object**

The properties of the chart include everything about the chart except the data set to which it is bound and the chart's ultimate display size.

- **Getting an ElementFactory object**

You use the ElementFactory object to create a new report element.

- **Setting the chart type and creating sample data**

The chart type and sample data provides a guide to the appearance of a chart element in a report design in BIRT Report Designer.

- **Getting an ExtendedItemHandle object**

The ExtendedItemHandle object is similar to a standard report item handle. The item handle is the object with which you access the report item instance. The handle is also the object that binds to a data set and the object that you add to the report design.

- **Setting the chart.instance property on the report item**

The chart.instance property of the report item identifies the chart instance object and links the report item to the chart instance object.

- **Getting a data set from the report design**

A chart must bind to data in order to have meaning. The report design provides access to one or more data sets that the report developer defined. The program also can create a data set and add it to the design.

- **Binding a chart to the data set**

To bind a chart to a data set, you specify the data set as a property of the extended item handle object.

- **Adding the new chart to the report design**

The last step is to add the chart to the report design by adding the extended item handle object to the report design.

- **Optionally saving the report design**

An application program that creates or modifies a BIRT report design can also save the new or modified report design.

The following sections describe these tasks in more detail and provide code examples for every step.

Creating the chart instance object

To create a chart instance object, you use a static method of one of the chart implementation classes. Depending on which chart implementation object you use, you can either create a chart with or without axes. The following line of code creates a chart with axes.

```
ChartWithAxes newChart = ChartWithAxesImpl.create( );
```

Setting the properties of the chart instance object

You define the characteristics of the chart by setting the properties of the chart instance object. For many properties, you first get a property object from the chart, then set properties on that object. The following sections describe setting various properties of the chart.

Setting the chart color and bounds

To set the chart's color and bounds, use setter methods of the chart instance object.

```
newChart.getBlock( ).setBackground  
    ( ColorDefinitionImpl.WHITE );  
newChart.getBlock( ).setBounds( BoundsImpl.create( 0, 0, 400,  
    250 ) );  
newChart.getTitle( ).getLabel( ).getCaption( ).setValue  
    ( "Europe" );
```

Setting plot properties

To set properties of the plot, first get a Plot object from the chart instance object, then use a setter method of a component of the Plot object.

```
Plot p = newChart.getPlot( );  
p.getClientArea( ).setBackground( ColorDefinitionImpl.create  
    ( 255, 255, 225 ) );
```

Setting legend properties

To set properties of the chart legend, first get a Legend object from the chart instance object, then use a setter method of a component of the Legend object.

```
Legend lg = newChart.getLegend( );  
lg.getText( ).getFont( ).setSize( 16 );  
lg.getInsets( ).set( 1, 1, 1, 1 );  
lg.getOutline( ).setVisible( false );  
lg.setAnchor( Anchor.NORTH_LITERAL );
```

Setting legend line properties

To set properties of the legend line, first get a LineAttribute object from the Legend object, as shown in the following code:

```
LineAttributes lia = lg.getOutline();
lia.setStyle(LineStyle.SOLID_LITERAL);
```

Setting axes properties

A chart with axes always has at least two axes, the primary base axis and the axis orthogonal to the base axis. There can be more than one primary base axis, but for every base axis there is one axis that is orthogonal to it.

To get a primary base axis, use the getPrimaryBaseAxes() method of the chart instance. This method returns an array. If there is only one primary base axis, get the zeroth element of the array, as shown in the following code:

```
Axis xAxisPrimary = newChart.getPrimaryBaseAxes()[0];
```

Listing 15-6 shows how to set the properties of the primary base axis.

Listing 15-6 Setting the properties of a primary base axis

```
xAxisPrimary.setType( AxisType.TEXT_LITERAL );
xAxisPrimary.getMajorGrid().setTickStyle
    ( TickStyle.BELOW_LITERAL );
xAxisPrimary.getOrigin().setType
    ( IntersectionType.VALUE_LITERAL );
xAxisPrimary.getTitle().setVisible( false );
```

To get the axis orthogonal to the primary base axis, use the chart instance object's getPrimaryOrthogonalAxis() method, as shown Listing 15-7.

Listing 15-7 Setting the properties of an orthogonal axis

```
Axis yAxisPrimary =
    newChart.getPrimaryOrthogonalAxis( xAxisPrimary );
yAxisPrimary.getMajorGrid().setTickStyle
    ( TickStyle.LEFT_LITERAL );
yAxisPrimary.setScale().setMax( NumberDataElementImpl.create
    ( 160 ) );
yAxisPrimary.setScale().setMin( NumberDataElementImpl.create
    ( -50 ) );
yAxisPrimary.getTitle().getCaption().setValue
    ( "Sales Growth" );
```

Creating a category series

On a chart with axes, the category series is the set of values that displays on the x-axis. On a chart without axes, the category series defines the number of sectors in a pie chart or the number of pointers on a meter chart. To create a category series, use the static create() method of the SeriesImpl class, as shown in Listing 15-8.

After you create a Series object, you must create either a query or a data set, then add that query or data set to the series data definition. To create a query, you use the static create() method of the QueryImpl class. If you create a data set, you must create the data set type that matches the values to display. You use the static create() method on the appropriate subclass of DataSetImpl. For example, date and time data values require a data set of class DateTimeDataSet. Listing 15-8 shows how to set both a query and a data set on a series.

Listing 15-8 Setting a query and a data set on category series

```
Series seriesCategory = SeriesImpl.create();
Query query = QueryImpl.create("row[\"Category\"]");
seriesCategory.getDataDefinition().add(query);

Series seBase = SeriesImpl.create();
DateTimeDataSet dsDateValues =
    DateTimeDataSetImpl.create(new Calendar[]{
        new CDateTime( 2004, 12, 21 ),
        new CDateTime( 2004, 12, 20 ),
        new CDateTime( 2004, 12, 17 ),
        new CDateTime( 2004, 12, 16 ),
    });
seBase.setDataSet(dsDateValues);
```

Creating an orthogonal series

On a chart with axes, the orthogonal series is the set of values that display on the y-axis. On a chart without axes, the orthogonal series determines the size of slices in a pie or the position of a pointer on a meter chart. To create an orthogonal series, use the static create() method of one of the following subclasses of the SeriesImpl class:

- AreaSeriesImpl
- BarSeriesImpl
- BubbleSeriesImpl
- DialSeriesImpl
- DifferenceSeriesImpl
- GanttSeriesImpl
- LineSeriesImpl
- PieSeriesImpl
- ScatterSeriesImpl
- StockSeriesImpl

The BarSeriesImpl class supports cone, pyramid, and tube chart types as well as bar charts. Because a chart can have multiple series of differing types, it is

not always possible to classify a chart as a single type. The following example creates a bar series and a line series.

```
BarSeries bs1 = ( BarSeries ) BarSeriesImpl.create( );
bs1.setSeriesIdentifier( "Q1" );
LineSeries ls1 = ( LineSeries ) LineSeriesImpl.create( );
ls1.setSeriesIdentifier( "Q2" );
```

Defining the orthogonal series data values

To set the data values for the orthogonal series, you can create Query objects and add the queries to the series objects. Alternatively, you can create data sets that hold numeric values. If you create a data set, you must create the data set type that matches the series. For example, a bar series requires a numeric data set, of class NumberDataSet. Some series require multiple sets of data. For example, a stock series requires four sets of data, high, low, open, and close. You can add four queries in that order to the series, or you can add a StockDataSet to the series. Listing 15-9 adds queries and data sets to line, bar, and stock series.

Listing 15-9 Setting queries and data sets on orthogonal series

```
Query query1 = QueryImpl.create( "row[\"Value1\"]" );
Query query2 = QueryImpl.create( "row[\"Value2\"]" );
bs1.getDataDefinition( ).add( query1 );
ls1.getDataDefinition( ).add( query2 );

NumberDataSet dsNumericValues1 = NumberDataSetImpl.create
    ( new double[ ]{ -10, 20, 80, 90 } );
seBase.setDataSet( dsNumericValues1 );

StockDataSet dsStockValues =
    StockDataSetImpl.create( new StockEntry[ ]{
        new StockEntry( 26.84, 27.15, 26.78, 26.97 ),
        new StockEntry( 27.00, 27.17, 26.94, 27.07 ),
        new StockEntry( 27.15, 27.28, 27.01, 27.16 ),
        new StockEntry( 27.22, 27.40, 27.07, 27.11 ),
    } );
StockSeries ss = ( StockSeries ) StockSeriesImpl.create( );
ss.setDataSet( dsStockValues );
```

Setting the orthogonal series properties

To set the properties of the y-series, use getter and setter methods of the appropriate series objects, as shown in Listing 15-10 and Listing 15-11 which set properties on BarSeries and LineSeries objects respectively.

Listing 15-10 Setting the properties of a bar series

```
bs1.getMarker( ).setType( MarkerType.TRIANGLE_LITERAL );
bs1.getLabel( ).setVisible( true );
bs1.getLabel( ).getCaption( ).setValue( "Q2" );
```

Listing 15-11 Setting the properties of two line series

```
ls1.getLineAttributes( ).setColor  
    ( ColorDefinitionImpl.RED( ) );  
ls1.getMarker( ).setType( MarkerType.TRIANGLE_LITERAL );  
ls1.getLabel( ).setVisible( true );  
ls1.getLabel( ).getCaption( ).setValue( "Q1" );  
ls2.getLineAttributes( ).setColor  
    ( ColorDefinitionImpl.YELLOW( ) );
```

Setting the series definition properties

To set the series definition properties of any series, you must first use the static `create()` method of the `SeriesDefinitionImpl` class to get a `SeriesDefinition` object. As well as providing the series to the chart object, this class also supports properties such as colors. To add a series to a series definition, you get the collection of series and add the series to that collection. To set the color of a series, get a `SeriesPalette` object and call its `shift()` method. Listing 15-12 shows how to set the colors and series on series definitions.

Listing 15-12 Setting the properties of a series definition

```
SeriesDefinition sd = SeriesDefinitionImpl.create( );  
sd.getSeriesPalette( ).shift( 0 );  
sd.getSeries( ).add( seCategory );  
SeriesDefinition sdCity = SeriesDefinitionImpl.create( );  
sdCity.getQuery( ).setDefinition( "Census.City" );  
sdCity.getSeries( ).add( sePie );
```

Adding a series definition to a chart

After setting the properties of a `SeriesDefinition` object, you must add the `SeriesDefinition` object to the chart. For a chart without axes, you add the series definitions directly to the chart object's collection of series definitions. The first series definition in the collection defines the category series and the second one defines the orthogonal series. For a chart with axes, you add the series definition to each `Axis` object's collection of series definitions as shown in Listing 15-13.

Listing 15-13 Adding series definitions to a pie chart and a chart with axes

```
ChartWithoutAxes cwoaPie = ChartWithoutAxesImpl.create( );  
cwoaPie.getSeriesDefinitions( ).add( sd );  
PieSeries sePie = ( PieSeries ) PieSeriesImpl.create( );  
sePie.setDataSet( seriesOneValues );  
sePie.setSeriesIdentifier( "Cities" )  
sd.getSeriesDefinitions( ).add( sdCity );  
  
xAxisPrimary.getSeriesDefinitions( ).add( sdX );  
yAxisPrimary.getSeriesDefinitions( ).add( sdY1 );  
yAxisPrimary.getSeriesDefinitions( ).add( sdY2 );
```

Adding series and queries to the series definitions

To add the series to the series definition, get the series collection from the SeriesDefinition object and add the series to it. To add a query to the series definition, get the query collection from the SeriesDefinition object and pass the query to the setDefinition() method. You add a category series to the SeriesDefinition object on an x-axis or to the first SeriesDefinition object on a chart without axes. You add an orthogonal series to the SeriesDefinition object on a y-axis or to the second SeriesDefinition object on a chart without axes. The following code shows how to add series and queries to series definitions:

```
sdY1.getSeries( ).add( ls1 );
sdY2.getSeries( ).add( bs1 );
sdY1.getQuery( ).setDefinition( "\"Q1\"");
sdY2.getQuery( ).setDefinition( "\"Q2\"");
sdX.getSeries( ).add( seriesCategory );
```

Using a chart item in a report design

The following tasks relate to the appearance and behavior of a chart inside a report design. If you are not deploying your chart as a report, you do not need to perform the tasks in this section.

Setting the chart type and subtype

A chart's type and subtype determine the appearance of the chart in BIRT Report Designer. In conjunction with sample data, these properties provide a realistic rendering of the chart item in the report design's layout window and in the chart dialogs. To ensure that the appearance of this rendered chart is as accurate as possible, you must set the chart's type so that it matches the series type that you set on the axis. Many types, such as bar, Gantt, line, and stock, are available for a chart with axes. For a chart without axes, the two classes each support only a single type, pie or dial. Some chart types have multiple subtypes. You set the chart type and subtype by using the Chart methods setType() and setSubType() respectively. These methods take a single String argument. Table 15-1 shows the available values for chart types and the valid subtypes for each chart type. Because cone, pyramid, and tube charts are merely different representations of a bar chart, they take the same subtypes as a bar chart.

Table 15-1 Chart type and subtype properties

Chart axes	Type	Subtype
With axes	Area Chart	Overlay
		Percent Stacked
		Stacked

Table 15-1 Chart type and subtype properties (*continued*)

Chart axes	Type	Subtype
	Bar Chart	Percent Stacked Side-by-side Stacked
	Bubble Chart	Standard Bubble Chart
	Cone Chart	Percent Stacked Side-by-side Stacked
	Difference Chart	Standard Difference Chart
	Gantt Chart	Standard Gantt Chart
	Line Chart	Overlay Percent Stacked Stacked
	Pyramid Chart	Percent Stacked Side-by-side Stacked
	Scatter Chart	Standard Scatter Chart
	Stock Chart	Standard Stock Chart Bar Stick Stock Chart
	Tube Chart	Percent Stacked Side-by-side Stacked
Without axes	Meter Chart	Standard Meter Chart Superimposed Meter Chart
	Pie Chart	Standard Pie Chart

The following code shows how to set a chart type and subtype:

```
ChartWithAxes cwaBar = ChartWithAxesImpl.create();
cwaBar.setType( "Bar Chart" );
cwaBar.setSubType( "Side-by-side" );
```

Creating sample data

This section describes an optional step in the creation of a chart. You do not need to perform this step if your application is stand-alone. Sample data provides visual information about its appearance in BIRT Report Designer. If you omit the code in Listing 15-14, the chart renders correctly when you

generate a report, but does not render sample values in the designer's layout window.

Listing 15-14 Adding sample data to a chart

```
SampleData sdt = DataFactory.eINSTANCE.createSampleData();
BaseSampleData sdBase =
    DataFactory.eINSTANCE.createBaseSampleData();
sdBase.setDataSetRepresentation( "A" );
sdt.getBaseSampleData( ).add( sdBase );

OrthogonalSampleData sdOrthogonal =
    DataFactory.eINSTANCE.createOrthogonalSampleData();
sdOrthogonal.setDataSetRepresentation( "1" );
sdOrthogonal.setSeriesDefinitionIndex( 0 );
sdt.getOrthogonalSampleData( ).add( sdOrthogonal );
newChart.setSampleData( sdt );
```

Getting a design engine element factory object

To create a chart item in a report design, you must have an ElementFactory object. To get an ElementFactory object, use the getElementFactory() method of the ReportDesignHandle object, as shown in the following line of code:

```
ElementFactory ef = dHandle.getElementFactory();
```

The chapter on programming with the BIRT APIs provides more information about using the Design Engine APIs and how to place a new item in a report design.

Getting an extended item handle object

A chart report item extends from the design engine's ReportItemHandle class by further extending the ExtendedItemHandle class. You use the ElementFactory object to create this object by using the newExtendedItem() method of the ElementFactory object, as shown in the following line of code:

```
ExtendedItemHandle chartHandle =
    ef.newExtendedItem( null, "Chart" );
```

Setting up the report item as a chart

You must set the chart.instance property of the report item object so that it contains the chart instance object. You get the report item from the extended item handle object, as shown in the following code:

```
try {
    chartHandle.getReportItem( ).setProperty
        ( "chart.instance", newChart );
} catch( ExtendedElementException e ) {
    e.printStackTrace();
}
```

Getting a data set from the report design

The new chart is still not bound to a data set. To bind a chart to a data set, you must get a data set from the report design, as shown in the following code that gets the first data set in the report design.

```
OdaDataSetHandle dataSet =
    ( OdaDataSetHandle ) dHandle.getDataSets( ).get( 0 );
```

Binding the chart to the data set

Use the extended item handle to bind the chart to the data set and to set the dimensions of the chart report item.

```
try {
    chartHandle.setDataSet( dataSet );
    chartHandle.setHeight( "250pt" );
    chartHandle.setWidth( "400pt" );
}
catch ( SemanticException e ) { e.printStackTrace( ); }
```

Adding the new chart to the report design

After the properties of the chart are set and the chart is bound to a data set, it is time to add the new chart to the report design. The following code adds the chart to the footer of a list item:

```
ListHandle li = ( ListHandle )
    dHandle.getBody( ).getContents( ).get( 0 );
try {
    li.getFooter( ).add( chartHandle );
}
catch ( ContentException e3 )
    { e3.printStackTrace( ); }
catch ( NameException e3 )
    { e3.printStackTrace( ); }
```

Saving the report design after adding the chart

Although the previous step is the final one in creating the chart and adding it to the report, the report design file does not yet contain the chart report item. Typically, you save the modified report design with a new name in order not to overwrite the original report design file.

```
try {
    dHandle.saveAs( "./Test_modified.rptdesign" );
}
catch ( IOException e ) {
    e.printStackTrace( );
}
dHandle.close( );
```

Putting it all together

The application in Listing 15-15 uses many of the techniques illustrated in this section and unites them in a coherent Java application that creates a chart report item.

Listing 15-15 Adding a chart to the report design

```
import java.io.IOException;
import java.util.HashMap;
import com.ibm.icu.util.ULocale;

import org.eclipse.birt.chart.model.*;
import org.eclipse.birt.chart.model.attribute.*;
import org.eclipse.birt.chart.model.attribute.impl.*;
import org.eclipse.birt.chart.model.component.*;
import org.eclipse.birt.chart.model.component.impl.*;
import org.eclipse.birt.chart.model.data.*;
import org.eclipse.birt.chart.model.data.impl.*;
import org.eclipse.birt.chart.model.layout.*;
import org.eclipse.birt.chart.model.type.*;
import org.eclipse.birt.chart.model.type.impl.*;
import org.eclipse.birt.chart.model.impl.*;

import org.eclipse.birt.report.engine.api.*;
import org.eclipse.birt.report.model.api.*;
import org.eclipse.birt.report.model.api.command.*;
import org.eclipse.birt.report.model.api.extension.*;
import org.eclipse.birt.report.model.api.activity.*;
import org.eclipse.birt.core.exception.BirtException;
import org.eclipse.birt.core.framework.Platform;

/****************************************************************************
 * Read a BIRT report design file, add a chart and write a
 * new report design file containing the added chart.
 * Run this application with the following command line:
 * java ChartReportApp origDesing modifiedDesign
****************************************************************************/
public class ChartReportApp
{
private EngineConfig config;
private static String birtHome =
    "C:/birt-runtime-2_2_1/ReportEngine";
private static String reportName =
    "./test.rptdesign";
private static String newReportDesign = "./test_new.rptdesign";

/****************************************************************************
 * Get the report design name and the name of the modified
 * report design from the command line if the command line has
 * any arguments.

```

```

* Create an instance of this class, create a new chart,
* write a new design file containing both the original and
* the new chart
*****/
public static void main( String[ ] args )
{
    String format = HTMLRenderOption.OUTPUT_FORMAT_HTML;
    //boolean showInfo = true;

    if(args.length > 0) {
        reportName = args[0];
    }
    if(args.length > 1) {
        newReportDesign = args[1];
    }

    ReportDesignHandle dHandle =
        createDesignHandle( reportName );

    // create an instance of this class
    ChartReportApp cra = new ChartReportApp( );

    // call the build method of this class
    cra.build( dHandle, newReportDesign );
}

/*****
 * The report design handle object is the entry point to
 * the report.

 * Create a report design handle object based on the
 * original design file by performing the following steps:

 * 1) Start the platform using the configuration object
 * 2) Create a design engine factory object from the
 *     platform
 * 3) Create a design engine using the factory object
 * 4) Create a session handle object from the design
 *     engine
 * 5) Create a design handle object from the session
 *     handle

 * The resulting design handle object is the entry point
 * to the report design and thus to the chart
*****/
private static ReportDesignHandle createDesignHandle
( String reportName )
{
    DesignConfig dConfig = new DesignConfig( );
    dConfig.setBIRTHome( birtHome );
    IDesignEngine dEngine = null;
    ReportDesignHandle dHandle = null;
}

```

```

try {
    Platform.startup( dConfig );
    IDesignEngineFactory dFactory =
        (IDesignEngineFactory) Platform.
        createFactoryObject(
            IDesignEngineFactory.EXTENSION DESIGN_ENGINE_FACTORY
        );
    dEngine = dFactory.createDesignEngine( dConfig );
    SessionHandle sessionHandle =
        dEngine.newSessionHandle(ULocale.ENGLISH );
    dHandle = sessionHandle.openDesign( reportName );
}
catch(BirtException e) {
    e.printStackTrace();
}
return dHandle;
}

/*************************************
* Build a chart
************************************/

public void build
    ( ReportDesignHandle dHandle, String newDesignName )
{
    // Create a new chart instance object
    ChartWithAxes newChart = ChartWithAxesImpl.create( );

    // Set the properties of the chart
    newChart.setType( "Line Chart" );
    newChart.setSubType( "Overlay" );

    newChart.getBlock().setBackground(
        ColorDefinitionImpl.WHITE() );
    newChart.getBlock().setBounds(
        BoundsImpl.create( 0, 0, 400, 250 ) );

    Plot p = newChart.getPlot();
    p.getClientArea().setBackground(
        ColorDefinitionImpl.create( 255, 255, 225 ) );
    newChart.getTitle().getLabel().getCaption().
        setValue( "Europe" );

    Legend lg = newChart.getLegend();
    LineAttributes lia = lg.getOutline();
    lg.getText().setFont().setSize( 16 );
    lia.setStyle( LineStyle.SOLID_LITERAL );
    lg.getInsets().set( 1, 1, 1, 1 );
    lg.getOutline().setVisible( false );
    lg.setAnchor( Anchor.NORTH_LITERAL );

    Axis xAxisPrimary = newChart.getPrimaryBaseAxes()[0];
    xAxisPrimary.setType( AxisType.TEXT_LITERAL );
}

```

```

xAxisPrimary.getMajorGrid().setTickStyle(
    TickStyle.BELOW_LITERAL );
xAxisPrimary.getOrigin().setType(
    IntersectionType.VALUE_LITERAL );
xAxisPrimary.getTitle().setVisible( false );

Axis yAxisPrimary = newChart.getPrimaryOrthogonalAxis(
    xAxisPrimary );
yAxisPrimary.getMajorGrid().setTickStyle(
    TickStyle.LEFT_LITERAL );
yAxisPrimary.getScale().setMax(
    NumberDataElementImpl.create( 160 ) );
yAxisPrimary.getScale().setMin(NumberDataElementImpl.
    create( -50 ) );
yAxisPrimary.getTitle().getCaption().
    setValue( "Sales Growth" );

// Create sample data.
SampleData sdt =
    DataFactory.eINSTANCE.createSampleData();
BaseSampleData sdBase =
    DataFactory.eINSTANCE.createBaseSampleData();
sdBase.setDataSetRepresentation("A");
sdt.getBaseSampleData().add( sdBase );

OrthogonalSampleData sdOrthogonal =
    DataFactory.eINSTANCE.createOrthogonalSampleData();
sdOrthogonal.setDataSetRepresentation( "1" );
sdOrthogonal.setSeriesDefinitionIndex(0);
sdt.getOrthogonalSampleData().add( sdOrthogonal );
newChart.setSampleData(sdt);
// Create the category series
Series seCategory = SeriesImpl.create();

// Set data value for X-Series
Query query = QueryImpl.create( "row[\"Category\"]" );
seCategory.getDataDefinition().add( query );

// Create the primary data set
LineSeries ls1 = ( LineSeries ) LineSeriesImpl.create();
ls1.setSeriesIdentifier("Q1");

// Set data value for Y-Series
Query query1 = QueryImpl.create( "row[\"Value1\"]" );
ls1.getDataDefinition().add( query1 );
ls1.getLineAttributes().setColor(
    ColorDefinitionImpl.RED( ) );
ls1.getMarker().setType( MarkerType.TRIANGLE_LITERAL );
ls1.getLabel().setVisible( true );
ls1.getLabel().getCaption().setValue( "Q1" );

LineSeries ls2 = (LineSeries) LineSeriesImpl.create( );
ls2.setSeriesIdentifier( "Q2" );

```

```

// Set data value for Y-Series
Query query2 = QueryImpl.create( "row[\"Value2\"]" );
ls2.getDataDefinition().add( query2 );

ls2.getLineAttributes().setColor(
    ColorDefinitionImpl.YELLOW() );
ls2.getMarker().setType( MarkerType.TRIANGLE_LITERAL );
ls2.getLabel().setVisible( true );
ls2.getLabel().getCaption().setValue( "Q2" );

SeriesDefinition sdX = SeriesDefinitionImpl.create();
sdX.getSeriesPalette().shift( 0 );
xAxisPrimary.getSeriesDefinitions().add( sdX );

SeriesDefinition sdY1 = SeriesDefinitionImpl.create();
sdY1.getSeriesPalette().shift( 0 );
sdY1.getSeries().add( ls1 );
sdY1.getQuery().setDefinition( "\"Q1\"" );
yAxisPrimary.getSeriesDefinitions().add( sdY1 );

SeriesDefinition sdY2 = SeriesDefinitionImpl.create();
sdY2.getSeriesPalette().shift( 0 );
sdY2.getSeries().add( ls2 );
sdY2.getQuery().setDefinition( "\"Q2\"" );
yAxisPrimary.getSeriesDefinitions().add( sdY2 );
sdX.getSeries().add( seCategory );

// Get a chart implementation object and set its
// chart.instance property

ElementFactory ef = dHandle.getElementFactory();
ExtendedItemHandle extendedItemHandle =
    ef.newExtendedItem( null, "Chart" );

try{
    IReportItem chartItem =
        extendedItemHandle.getReportItem();
    chartItem.setProperty( "chart.instance", newChart );
} catch( ExtendedElementException e ) {
    e.printStackTrace();
}

// Get an ODA data set and bind it to the chart
OdaDataSetHandle dataSet = ( OdaDataSetHandle )
    dHandle.getDataSets().get( 0 );
try {
    extendedItemHandle.setDataSet( dataSet );
    extendedItemHandle.setHeight( "250pt" );
    extendedItemHandle.setWidth( "400pt" );
} catch ( SemanticException e ) {
    e.printStackTrace();
}

```

```

// Add the chart to the report design
ListHandle li = (ListHandle) dHandle.getBody( ).getContents( ).get( 0 );
try {
    li.getFooter( ).add( extendedItemHandle );
} catch ( ContentException e3 ) {
    e3.printStackTrace( );
} catch ( NameException e3 ) {
    e3.printStackTrace( );
}
// Save the report design that now contains a chart
try {
    dHandle.saveAs( newDesignName );
} catch ( IOException e ) {
    e.printStackTrace( );
}
dHandle.close();
}
}

```

Using the BIRT charting API in a Java Swing application

The BIRT charting API does not rely on the BIRT design engine or the BIRT report engine. You can use the BIRT charting API to generate a chart in any Java application.

The program shown in Listing 15-16 uses the BIRT charting API to build a chart in a Java Swing application. The application does not use the BIRT design engine or the BIRT report engine, and it does not process a BIRT report design file.

Listing 15-16 Java Swing charting application

```

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridLayout;
import java.awt.Rectangle;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;

```

```

import java.awt.event.ActionListener;
import java.awt.event.ComponentEvent;
import java.awt.event.ComponentListener;
import java.util.HashMap;
import java.util.Map;
import javax.swing.JFrame;
import javax.swing.JPanel;

import org.eclipse.birt.chart.device.IDeviceRenderer;
import org.eclipse.birt.chart.device.IUpdateNotifier;
import org.eclipse.birt.chart.exception.ChartException;
import org.eclipse.birt.chart.factory.GeneratedChartState;
import org.eclipse.birt.chart.factory.Generator;
import org.eclipse.birt.chart.model.Chart;
import org.eclipse.birt.chart.model.ChartWithAxes;
import org.eclipse.birt.chart.model.attribute.AxisType;
import org.eclipse.birt.chart.model.attribute.Bounds;
import org.eclipse.birt.chart.model.attribute.impl.BoundsImpl;
import org.eclipse.birt.chart.model.component.Axis;
import org.eclipse.birt.chart.util.PluginSettings;
import org.eclipse.birt.chart.model.impl.ChartWithAxesImpl;
import org.eclipse.birt.chart.model.attribute.impl.
    ColorDefinitionImpl;
import org.eclipse.birt.chart.model.layout.Plot;
import org.eclipse.birt.chart.model.layout.Legend;
import org.eclipse.birt.chart.model.attribute.
    IntersectionType;
import org.eclipse.birt.chart.model.attribute.LegendItemType;
import org.eclipse.birt.chart.model.attribute.TickStyle;
import org.eclipse.birt.chart.model.data.NumberDataSet;
import org.eclipse.birt.chart.model.data.impl.
    NumberDataSetImpl;
import org.eclipse.birt.chart.model.data.TextDataSet;
import org.eclipse.birt.chart.model.component.Series;
import org.eclipse.birt.chart.model.data.SeriesDefinition;
import org.eclipse.birt.chart.model.type.BarSeries;
import org.eclipse.birt.chart.model.data.impl.
    SeriesDefinitionImpl;
import org.eclipse.birt.chart.model.attribute.Position;
import org.eclipse.birt.chart.model.data.impl.TextDataSetImpl;
import org.eclipse.birt.chart.model.type.impl.BarSeriesImpl;
import org.eclipse.birt.chart.model.component.impl.SeriesImpl;

/*
 * The selector of charts in Swing JPanel.
 */

public final class SwingChartingApp extends JPanel implements
    IUpdateNotifier,
    ComponentListener
{

```

```

private static final long serialVersionUID = 1L;
private boolean bNeedsGeneration = true;
private GeneratedChartState gcs = null;
private Chart cm = null;
private IDeviceRenderer idr = null;
private Map contextMap;

/*
 * Create the layout with a container for displaying a chart
 */

public static void main( String[ ] args )
{
    SwingChartingApp scv = new SwingChartingApp( );
    JFrame jf = new JFrame( );
    jf.setDefaultCloseOperation( JFrame.DISPOSE_ON_CLOSE );
    jf.addComponentListener( scv );
    Container co = jf.getContentPane( );
    co.setLayout( new BorderLayout( ) );
    co.add( scv, BorderLayout.CENTER );

    Dimension dScreen = Toolkit.
        getDefaultToolkit( ).getScreenSize( );
    Dimension dApp = new Dimension( 800, 600 );
    jf.setSize( dApp );
    jf.setLocation( ( dScreen.width - dApp.width ) / 2,
        ( dScreen.height - dApp.height ) / 2 );
    jf.setTitle( scv.getClass( ).getName( ) + " [device="
        + scv.idr.getClass( ).getName( )
        + "]");//$NON-NLS-1$
    jf.show( );
}

/*
 * Connect with a SWING device to render the graphics.
 */

SwingChartingApp( )
{
    contextMap = new HashMap( );
    final PluginSettings ps = PluginSettings.instance( );
    try
    {
        idr = ps.getDevice( "dv.SWING" );//$NON-NLS-1$
    }
    catch ( ChartException ex )
    {
        ex.printStackTrace( );
    }
    cm = createBarChart( );
}

```

```

/* Build a simple bar chart */

public static final Chart createBarChart( )
{
    ChartWithAxes cwaBar = ChartWithAxesImpl.create( );

    /* Plot */
    cwaBar.getBlock( ).setBackgroundColor( ColorDefinitionImpl.WHITE( ) );
    cwaBar.getBlock( ).getOutline( ).setVisible( true );
    Plot p = cwaBar.getPlot( );
    p.getClientArea( ).setBackgroundColor(
        ColorDefinitionImpl.create( 255, 255, 225 ) );
    p.getOutline( ).setVisible( false );

    /* Title */
    cwaBar.getTitle( ).getLabel( ).getCaption( ).setValue( "Bar Chart" );

    /* Legend */
    Legend lg = cwaBar.getLegend( );
    lg.getText( ).setFont( ).setSize( 16 );
    lg.setItemType( LegendItemType.CATEGORIES_LITERAL );

    /* X-Axis */
    Axis xAxisPrimary = cwaBar.getPrimaryBaseAxes( )[0];
    xAxisPrimary.setType( AxisType.TEXT_LITERAL );
    xAxisPrimary.getMajorGrid( ).setTickStyle( TickStyle.BELOW_LITERAL );
    xAxisPrimary.getOrigin( ).setType( IntersectionType.VALUE_LITERAL );
    xAxisPrimary.getTitle( ).setVisible( true );

    /* Y-Axis */
    Axis yAxisPrimary = cwaBar.getPrimaryOrthogonalAxis( xAxisPrimary );
    yAxisPrimary.getMajorGrid( ).setTickStyle( TickStyle.LEFT_LITERAL );
    yAxisPrimary.setType( AxisType.LINEAR_LITERAL );
    yAxisPrimary.getLabel( ).getCaption( ).getFont( ).setRotation( 90 );

    /* Data Sets */
    TextDataSet categoryValues =
        TextDataSetImpl.create( new String[]{ "Item 1", "Item 2", "Item 3" } );
    NumberDataSet orthoValues = NumberDataSetImpl.create( new double[]{ 25, 35, 15 } );

    /* X-Series */
    Series seCategory = SeriesImpl.create( );
    seCategory.setDataSet( categoryValues );

```

```

        SeriesDefinition sdX = SeriesDefinitionImpl.create( );
        sdX.getSeriesPalette( ).shift( 0 );
        xAxisPrimary.getSeriesDefinitions( ).add( sdX );
        sdX.getSeries( ).add( seCategory );

        /* Y-Series */
        BarSeries bs = (BarSeries) BarSeriesImpl.create( );
        bs.setDataSet( orthoValues );
        bs.setRiserOutline( null );
        bs.getLabel( ).setVisible( true );
        bs.setLabelPosition( Position.INSIDE_LITERAL );

        SeriesDefinition sdY = SeriesDefinitionImpl.create( );
        yAxisPrimary.getSeriesDefinitions( ).add( sdY );
        sdY.getSeries( ).add( bs );
        return cwaBar;
    }

    public void regenerateChart( )
    {
        bNeedsGeneration = true;
        repaint( );
    }

    public void repaintChart( )
    {
        repaint( );
    }

    public Object peerInstance( )
    {
        return this;
    }

    public Chart getDesignTimeModel( )
    {
        return cm;
    }

    public Object getContext( Object key )
    {
        return contextMap.get( key );
    }

    public Object putContext( Object key, Object value )
    {
        return contextMap.put( key, value );
    }

    public Object removeContext( Object key )
    {
        return contextMap.remove( key );
    }
}

```

```

public Chart getRunTimeModel( )
{
    return gcs.getChartModel( );
}

public void paint( Graphics g )
{
    super.paint( g );
    Graphics2D g2d = (Graphics2D) g;
    idr.setProperty( IDeviceRenderer.GRAPHICS_CONTEXT, g2d );
    idr.setProperty( IDeviceRenderer.UPDATE_NOTIFIER, this );
    Dimension d = getSize( );
    Bounds bo = BoundsImpl.create( 0, 0, d.width, d.height );
    bo.scale( 72d / idr.getDisplayServer( ).getDpiResolution( ) );
    Generator gr = Generator.instance( );

    if ( bNeedsGeneration )
    {
        bNeedsGeneration = false;
        try
        {
            gcs = gr.build( idr.getDisplayServer( ),
                cm,
                bo,
                null,
                null,
                null );
        }
        catch ( ChartException ex ) {
            System.out.println( ex );
        }
    }
    try
    {
        gr.render( idr, gcs );
    }
    catch ( ChartException ex ) { System.out.println( ex ); }
}

public void componentHidden( ComponentEvent e ) { }

public void componentMoved( ComponentEvent e ) { }

public void componentResized( ComponentEvent e )
{
    bNeedsGeneration = true;
}

public void componentShown( ComponentEvent e ) { }
}

```

Understanding the chart programming examples

The org.eclipse.birt.chart.examples plug-in is a collection of chart programming examples. You can access the Java source code of the examples by extracting the plug-in's JAR file to your workspace, then importing those files as a project. You must include the JAR files in the BIRT home folder in your build path. Add further JAR files from the chart engine's ChartSDK plugins folder and the Eclipse home plugins folder as necessary to resolve build errors in individual examples. To run the examples, use the Eclipse Run dialog to set the VM arguments to include a line of the following form, where <Eclipse home> is the location of eclipse.exe:

```
-DBIRT_HOME=<Eclipse home>"
```

The examples are located in subdirectories of the plug-in's src/org/eclipse/birt/chart/examples directory, called EXAMPLES_ROOT. Most of the examples consist of a Java application that displays a chart. The application classes, which have a main() method, are called viewer applications and their class names end in Viewer. Typically, these examples use one or more additional classes to build the chart. The following sections provide brief summaries of the examples in the chart examples plug-in.

api.data examples

The api.data package contains three examples, one that displays charts in a Java Swing environment and two that modify chart items in a report design.

DataCharts example

The DataCharts example consists of DataChartsViewer, a Java Swing application that uses the DataCharts class to build a chart. DataCharts displays hard-coded data values in the charts. Depending on user selection, the application builds one of the following kinds of charts:

- A bar chart that has multiple y-axes
- A bar chart that has multiple y-series
- A pie chart that has a minimum slice

GroupOnXSeries example

The GroupOnXSeries example is a Java application that reads a BIRT report design and modifies and saves it. The original report design, NonGroupOnXSeries.rptdesign, contains a chart report item that uses data from a scripted data source. The chart item has no grouping on the X series. The GroupOnXSeries Java application modifies the design so that the chart report item does group on the X series. The application saves the modified report design as GroupOnXSeries.rptdesign. You can open these report designs and preview the reports to see the effect of this modification.

GroupOnYAxis example

The GroupOnYAxis example is a Java application that reads a BIRT report design and modifies and saves it. The original report design, NonGroupOnYAxis.rptdesign, contains a chart report item that uses data from a scripted data source. The chart item has no grouping on the y-axis. The GroupOnYAxis Java application modifies the design so that the chart report item does group on the y-axis. The application saves the modified BIRT report design as GroupOnYAxis.rptdesign. You can open these report designs and preview the reports to see the effect of this modification.

api.data.autobinding example

This example is an Eclipse SWT application that consists of the AutoDataBindingViewer class. This class instantiates an SWT Display object and adds a chart to it. The application creates data row structures, which it binds to the chart. Then the application renders the chart.

api.format example

This example is a Java Swing application that consists of the FormatCharts and FormatChartsViewer Java classes. The FormatChartsViewer class displays an interface that presents choices to the user. Based on the user choice, FormatChartsViewer calls static methods in the FormatCharts class to build a chart. FormatChartsViewer then renders the chart. The methods in FormatCharts modify the following chart properties:

- Axis format
- Colored by category
- Legend title
- Percentage value
- Plot format
- Series format

api.interactivity examples

This set of related example applications demonstrate chart interactivity features in the three Java frameworks, SVG, Swing, and SWT. The three viewer applications are SvgInteractivityViewer, SwingInteractivityViewer, and SwtInteractivityViewer. All three viewer classes display an interface that presents the same interactivity choices to the user. Based on the user choice, the viewer class calls static methods in the InteractivityCharts class to build an interactive chart. The viewer then renders the chart. The types of interactivity illustrated in these charts are:

- Displaying tooltips
- Executing call-back code

- Highlighting a series
- Linking to an external site by using a URL
- Toggling the visibility of a series

api.pdf example

This example is a Java application that builds a simple chart and renders it as a PDF file. The classes in the PDFChartGenerator example are ChartModels and PDFChartGenerator. ChartModels has a single method that builds a simple chart using hard-coded data values. PDFChartGenerator uses the BIRT charting API to render the chart into PDF format. The application saves the PDF file as test.pdf.

api.preference example

This example shows how a Java servlet can process URL parameters to set style preferences for a chart. The servlet class, PreferenceServlet, uses the ChartModels class to generate a chart. The servlet uses the style parameters in the LabelStyleProcessor class to affect the style of a label in the chart. The example also includes a help page, Help.htm, that explains how to:

- Develop chart pages using JSPs and servlets.
- Run the Preference example.
- Set up Eclipse to work with Tomcat.

api.processor example

This example builds a simple chart and applies styles to text in the chart. The example consists of StyleChartViewer, an SWT application, and StyleProcessor, which implements the IStyleProcessor interface to create a style object. StyleChartViewer creates a chart and applies the style to text in the chart. Finally, StyleChartViewer renders the chart.

api.script examples

This example consists of two SWT applications, JavaScriptViewer and JavaViewer. Both applications present the same set of choices to the user. The appearance of the chart that appears for a particular user choice is the same for both viewers. Each choice calls a static method in the ScriptCharts class to create a chart and displays the event handlers that the chart implements. JavaScriptViewer calls ScriptCharts methods to build charts that have JavaScript event handlers. JavaViewer calls ScriptCharts methods to build charts that have Java event handlers.

The ScriptCharts class illustrates the techniques for creating charts with report element event handlers. This class has methods to create charts with report element event handlers written in JavaScript. Each JavaScript event

handler is defined as a single string in ScriptCharts. A further set of methods create charts with the same functionality with report element event handlers written in Java. The Java event handlers are Java classes that are located in EXAMPLES_ROOT/api/script/java. The ScriptCharts methods that define a Java event handler pass a string containing the path of the Java class.

api.viewer examples

The api.viewer package contains six example applications that create a wide variety of charts. Each class creates and displays a set of charts based on user choices. Each viewer class, except SwingLiveChartViewer.java, calls static methods in the PrimitiveCharts class to create the chart to display. PrimitiveCharts uses hard-coded data values for each chart.

Chart3DViewer example

Chart3DViewer.java is an SWT application that displays the following chart types:

- 3D area chart
- 3D bar chart
- 3D line chart

CurveFittingViewer example

CurveFittingViewer.java is an SWT application that displays the following chart types:

- Curve fitting area chart
- Curve fitting bar chart
- Curve fitting line chart
- Curve fitting stock chart

DialChartViewer example

DialChartViewer.java is an SWT application that displays the following chart types:

- Multiple-dial, multiple-region chart
- Multiple-dial, single-region chart
- Single-dial, multiple-region chart
- Single-dial, single-region chart

SwingChartViewerSelector example

SwingChartViewerSelector.java is a Swing application that provides the user with the ability to show the same data values in different ways. For each

chart type, the user can choose to show the chart as two-dimensional or two-dimensional with depth. The user can also choose to display the chart with axes transposed, with the values shown as percentages, or on a logarithmic scale. Some choices are available only for charts with axes.

SwingChartViewerSelector displays the following chart types:

- Area chart
- Bar and line stacked chart
- Bar chart
- Bar chart that has two series
- Bubble chart
- Difference chart
- Line chart
- Pie chart
- Pie chart that has four series
- Scatter chart
- Stock chart

SwingLiveChartViewer example

SwingChartLiveChartViewer.java is a Swing application that displays a live animated chart with scrolling data.

SWTChartViewerSelector example

SWTChartViewerSelector.java is an SWT application that displays the same user interface choices and chart types as SwingChartViewerSelector.java.

builder example

The builder example consists of two Java classes, ChartWizardLauncher and DefaultDataProviderImpl. ChartWizardLauncher attempts to read a chart from testCharts.chart. If the file exists, it modifies that file. If the file does not exist, the application creates a new file. ChartWizardLauncher uses the BIRT chart wizard to create the chart. DefaultDataProviderImpl provides a basic implementation of a simulated data service.

report examples

The three report examples are all Java applications that have no user interface. These examples use the BIRT design engine to build a new BIRT report design file in the chart example plug-in's output folder. All of the report examples add the following components to the report design file in the order shown:

- Master page
- Data source
- Data set
- A chart element in the body of the report

All three report examples use the BIRT charting API to add a chart to the body slot of the report design. After you run these applications, you can open the new report design file and preview the report to see the chart.

MeterChartExample example

The MeterChartExample example adds a meter chart to the report design. The name of the report design is MeterChartExample.rptdesign.

SalesReport example

The SalesReport example creates styles and adds a pie chart to the report design. The name of the report design is SalesReport.rptdesign.

StockReport example

The StockReport example adds a stock chart to the report design. The name of the report design is StockAnalysis.rptdesign.

report.design examples

The three report design examples in this folder demonstrate techniques for displaying chart elements in a report design.

BarChartWithinTableGroup.rptdesign shows how to use a chart within a table element. BarChartWithJavascript.rptdesign demonstrates how to use JavaScript to modify the rendered chart when the user views the report. DynamicSeriesPieChart.rptdesign shows how to customize a pie chart.

report.design.script examples

The report designs in this folder contain chart elements that use the same JavaScript event handlers that the api.script examples create dynamically. These report designs show how the scripts appear in BIRT Report Designer.

view example

The view folder and its subfolders include the content for the Eclipse view, Chart Examples. To open this view, from the Eclipse main menu, choose Window>Show View>Other. On Show View, expand Report and Chart Design, then select Chart Examples and choose OK. By default, the Chart Examples view appears below the main window, in the same position as the console window. To use the Chart Examples view, expand a node on the left

of the view, then select an item. The selected chart appears on the right of the view, as shown in Figure 15-1.

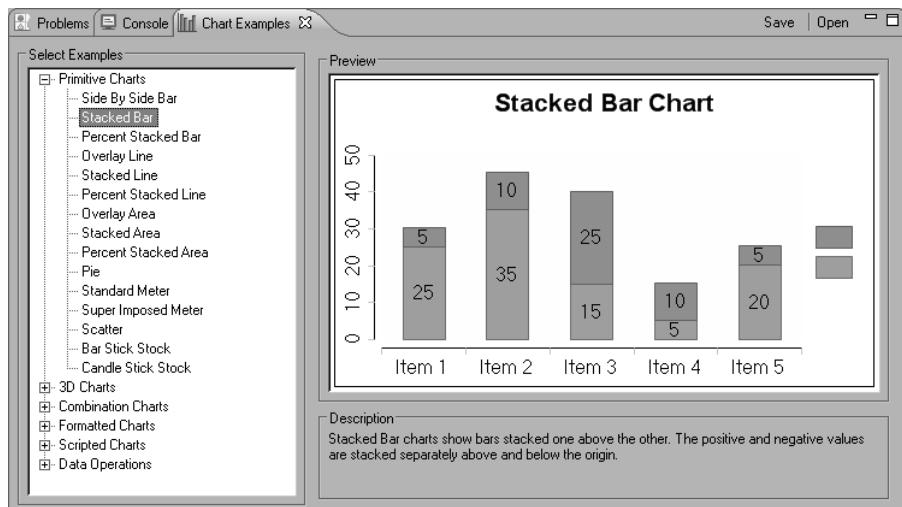


Figure 15-1 Chart Examples view, showing the stacked bar chart example

To view the source code of the application that generates the selected chart, choose Open from the toolbar. To save the XML of the chart item structure, choose Save from the toolbar.

This page intentionally left blank

V

Working with the Extension Framework



This page intentionally left blank

16

Building the BIRT Project

This chapter explains how to build the BIRT project. The chapter also explains how to build a new web viewer that employs your modifications to BIRT.

This information is primarily for contributors to the BIRT open source project and to those who want to create a custom version of BIRT. You do not need to build BIRT to extend BIRT or write scripts for BIRT.

After the build process completes, you can explore the BIRT source code and make changes to the code to suit your needs. Any changes that you make to the BIRT source are immediately implemented in the BIRT Report Designer. To implement your changes in a web application, you must build and export a new web viewer.

About building the BIRT project

Building the BIRT project consists of the following tasks:

- Installing a working version of BIRT Report Designer

By first installing the same version of BIRT Report Designer that you want to build, you ensure that your Eclipse environment is correct and that you have all the necessary components for building BIRT.

- Configuring the Eclipse workspace to compile BIRT

You may need to change the compiler settings to match those that are required to build BIRT.

- Downloading and extracting the correct version of the BIRT source code
You download the BIRT source code from the BIRT web site. The version you download and the working version of BIRT on your system must be identical.
- Importing and building the BIRT projects

The act of importing the BIRT projects initiates the build process. The build process takes a few minutes and should complete with no errors.

Installing a working version of BIRT

To ensure that you have all the components that are necessary for building BIRT, first install a working version of BIRT Report Designer. You must ensure that all the plug-ins and auxiliary files BIRT requires are present and that these resources are the versions the source code requires.

The simplest and surest way to know that you have all the necessary components of a working BIRT installation is to choose the all-in-one installation option. After installing this package, download the BIRT source code. Follow the instructions in the installation chapters.

After installing BIRT, launch Eclipse and verify that you can build a simple report and that you can preview the report in both the BIRT previewer and in the web viewer. Typically, this test is sufficient to guarantee that you have a valid working version of BIRT. Of course, you can run more extensive tests to confirm access to any custom items in your environment, for example, custom data sources.

Configuring the Eclipse workspace to compile BIRT

The 2.2.1 version of the BIRT source code uses some features that are only present in JDK version 1.5 and higher. This JDK version is often called Java 5.0. In order for BIRT to build successfully, you need to set the Eclipse compiler compliance to Java 5.0. You can set this version for your Eclipse workspace by starting Eclipse and setting Eclipse preferences.

If you create a new workspace or switch to a different workspace, you can lose these compiler settings. In this case, you must reset them as described in the following instructions.

How to set Eclipse workspace compiler preferences

- 1 Choose Window->Preferences. Preferences appears, as shown in Figure 16-1.

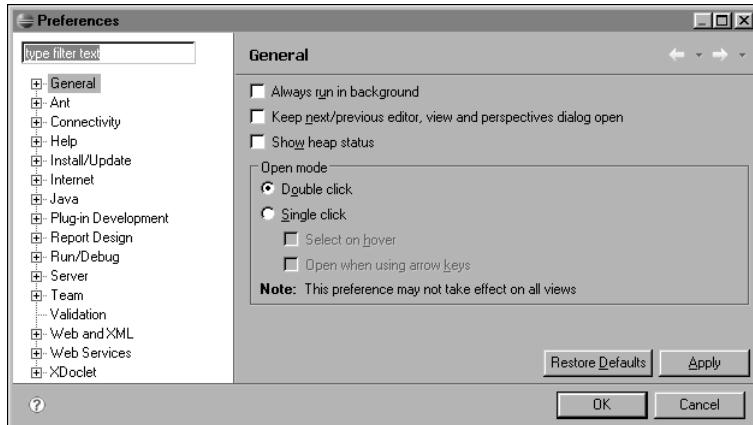


Figure 16-1 The Preferences dialog

- 2 Expand the Java entry in the tree and select Compiler.
- 3 In JDK Compliance, as shown in Figure 16-2, perform the following steps:
 - Select 5.0 in Compiler compliance level.
 - Deselect Use default compliance settings.
 - In Generated .class files compatibility, select 5.0.
 - In Source compatibility, select 5.0.

In Classfile Generation, accept the default settings and choose OK.

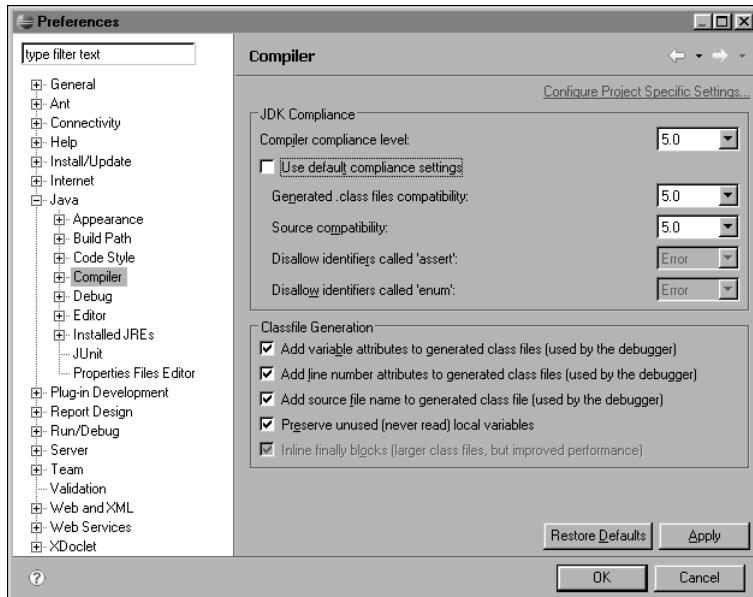


Figure 16-2 Compiler preferences

If a message appears asking if you want to do a full rebuild now, choose No. This message appears only if you changed the compiler settings and there are Java projects in the workspace.

Downloading and extracting the correct version of the BIRT source code

Download the BIRT source code from the same page from which you downloaded the working version of BIRT. The source archive is in the section labeled BIRT Source Code. The archive file has a name similar to birt-source-2_2_1.zip. If you install the source code from a different BIRT build, Eclipse uses the source code in preference to the existing plug-ins.

Extract the source code archive into a workspace directory. You may want to create a workspace that is separate from workspaces that contain reports. This separate workspace is not a requirement, but it will keep your report workspace less cluttered.

Importing, building, and testing the BIRT project

The BIRT source download contains everything that you need to build BIRT. The download contains the feature files, plug-in files, and source files for BIRT. A single Eclipse import operation imports and builds the projects.

How to import and build the BIRT project

- 1 To import the BIRT project, choose File→Import. Import appears as shown in Figure 16-3.

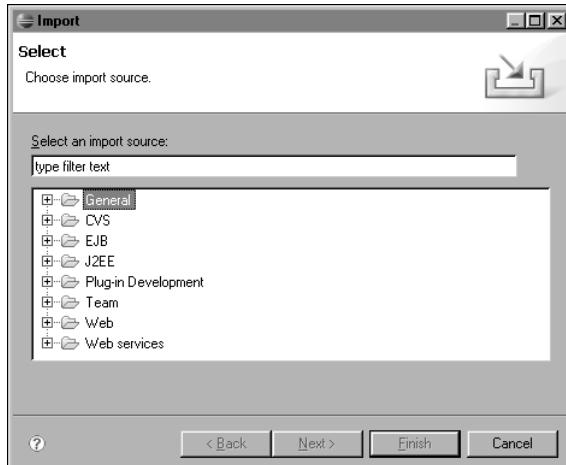


Figure 16-3 Project Import

- 2** Expand the General node and choose Existing Projects into Workspace, as shown in Figure 16-4.

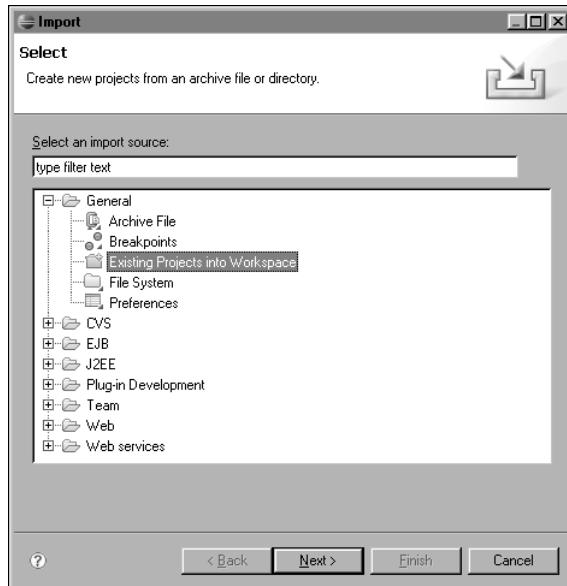


Figure 16-4 Import Project showing expanded General node

- 3** Choose Next. On Import Projects, choose the Browse button next to Select root directory and navigate to the workspace. A list of all the BIRT projects appears in the Projects pane, as shown in Figure 16-5.

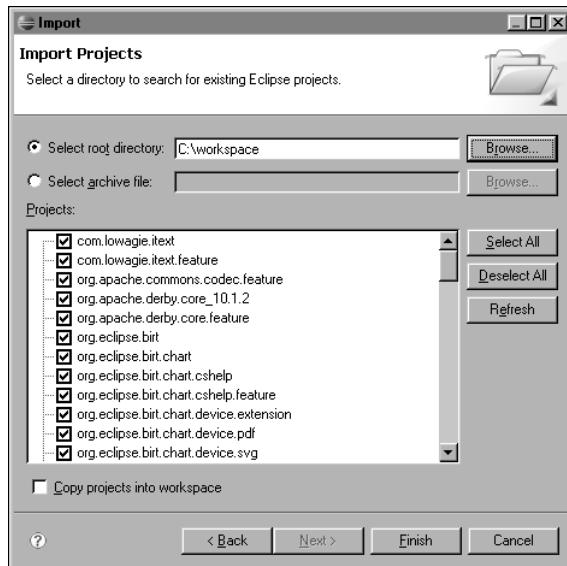


Figure 16-5 List of BIRT projects

- 4** Choose Select All and then choose Finish.

Eclipse refreshes the workspace, then builds all the projects in a process that can take several minutes. When the build finishes with no errors, the workspace looks like the one in Figure 16-6.

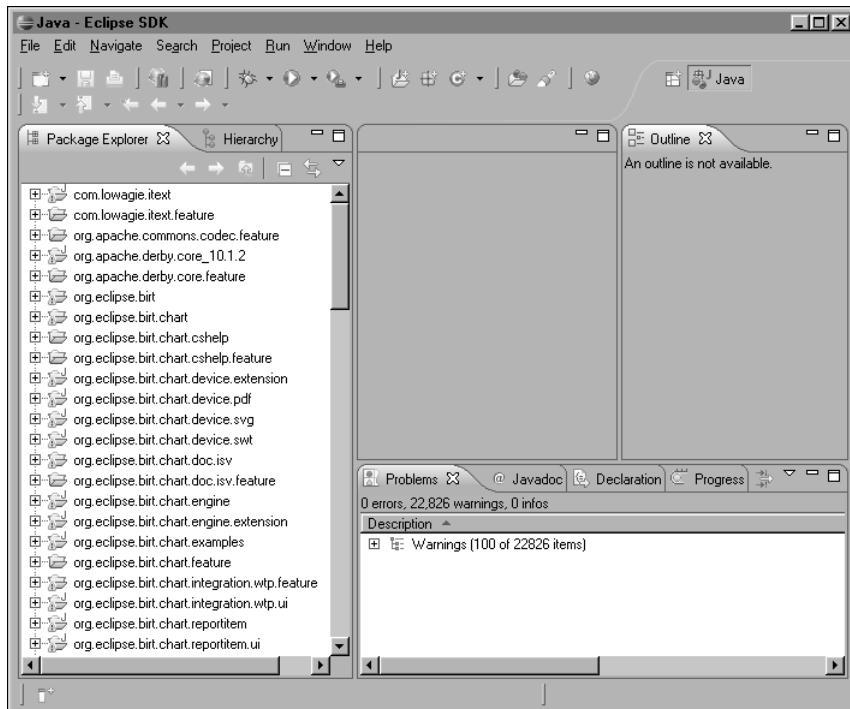


Figure 16-6 Build results with no errors

How to test the BIRT project

You test the BIRT project by running the org.eclipse.birt package as an Eclipse application.

- 1** In Package Explorer, select the org.eclipse.birt package.

- 2** From the Eclipse menu, choose Run→Run As→Eclipse Application.

Eclipse builds a new default workspace. On a Windows system, the name of the workspace is C:\runtime-EclipseApplication. Eclipse then opens a new instance of the Eclipse workbench that shows the welcome page.

- 3** Close the welcome page.

- 4** Open the Report Design perspective. Choose Window→Open Perspective→Report Design. If Report Design is not available, choose Other. On Open Perspective, select Report Design, then choose OK.

You can now create report projects and designs, using this perspective that Eclipse built from the BIRT source code.

Building new JAR files to display BIRT output

If you make changes to the BIRT source, you also need to recreate the files that the web viewer uses to display the BIRT output. For a report, this file is viewservlets.jar. For stand-alone chart viewing, this file is chart-viewer.jar.

Building the viewservlets.jar file

You build viewservlets.jar in the org.eclipse.birt.report.viewer plug-in project. The following procedure shows how to build this file using Eclipse.

How to regenerate the viewservlets.jar file

- 1 From the Java perspective, choose Window→Open Perspective→Java.
- 2 In Package Explorer, expand the node org.eclipse.birt.report.viewer.
- 3 In org.eclipse.birt.report.viewer, expand META-INF and right-click MANIFEST.MF.
- 4 From the context menu, choose PDE Tools→Create Ant Build File, as shown in Figure 16-7.

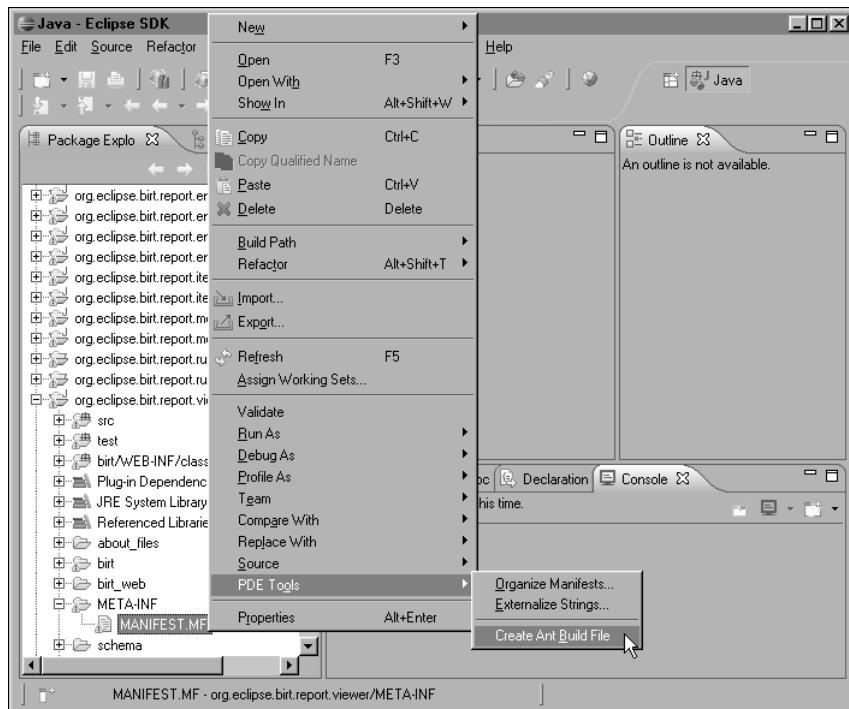


Figure 16-7 Choosing Create Ant Build File

Create Ant Build File creates build.xml in the root folder of org.eclipse.birt.report.viewer.

- 5 In the org.eclipse.birt.report.viewer folder, right-click on build.xml and choose Run As ➤ 2 Ant Build, as shown in Figure 16-8.

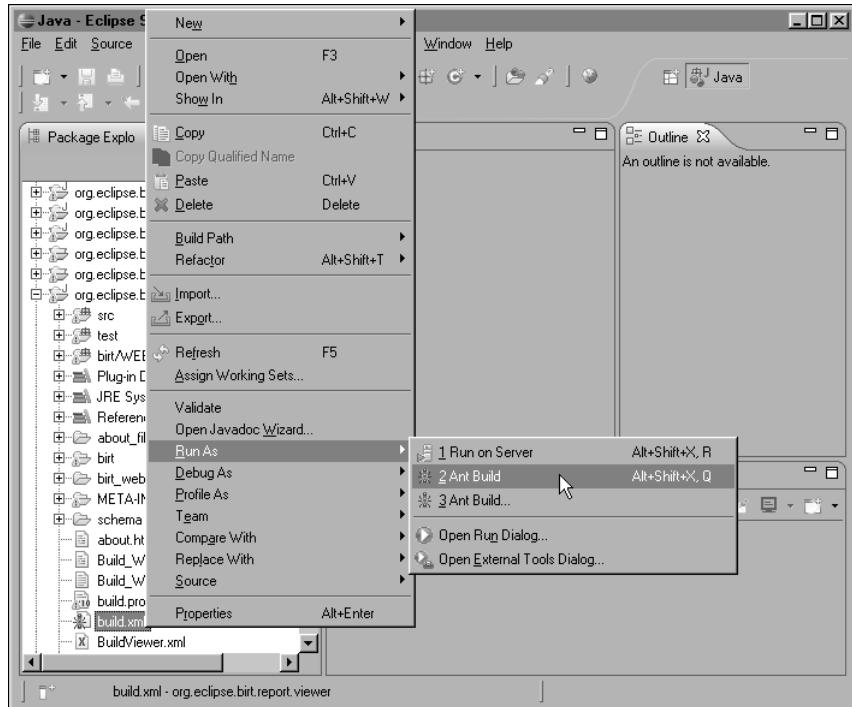


Figure 16-8 Running the Ant build to create view servlets.jar

The Ant build process takes several minutes and creates view servlets.jar in org.eclipse.birt.report.viewer/birt/WEB-INF/lib.

Building the chart-viewer.jar file

If you make changes to the charting source, you probably want to generate a new chart-viewer.jar file. You can use a similar procedure to the one for recreating view servlets.jar. To build chart-viewer.jar, you perform the procedure in org.eclipse.birt.chart.viewer. The build process creates chart-viewer.jar in the org.eclipse.birt.chart.viewer root directory.

17

Extending BIRT

This chapter provides an overview of the BIRT extension framework and shows how to create and deploy a BIRT extension using the Eclipse Plug-in Development Environment (PDE).

Overview of the extension framework

The Eclipse platform is an open source, integrated system of application development tools that you implement and extend using a plug-in interface. Eclipse provides a set of core plug-ins that configure the basic services for the platform's framework. A platform developer can build and integrate new tools in this application development system.

BIRT is a set of plug-in extensions that enable a developer to add reporting functionality to an application. The BIRT APIs define extension points that allow a developer to add custom functionality to the BIRT framework.

Eclipse makes BIRT source code available to the developer community in the CVS repository.

In the Eclipse installation, the name of a plug-in directory contains an appended version number. This book omits the version number from the names of the plug-in directories. For example, the book abbreviates the name of the plug-in directory, `org.eclipse.birt.report.data.oda.jdbc_2.2.1.v20070919`, to `org.eclipse.birt.report.data.oda.jdbc`.

The following sections provide a general description of how to make an extension to a defined extension point in the BIRT release 2.2.1 framework using the Eclipse PDE.

Understanding the structure of a BIRT plug-in

An Eclipse plug-in implements the following components:

- Extension point schema definition

An XML document that specifies a grammar that you must follow when defining the elements of a plug-in extension in the Eclipse PDE

- Plug-in manifest

An XML document that describes the plug-in's activation framework to the Eclipse run-time environment

- Plug-in run-time class

A Java class that defines the methods for starting, managing, and stopping a plug-in instance

The following sections provide detailed descriptions of these Eclipse plug-in components.

Understanding an extension point schema definition file

A plug-in directory typically contains an XML extension point schema definition (.exsd) file in a schema subdirectory. The XML schema documents the elements, attributes, and types used by the extension point. The Eclipse PDE uses this information to describe the elements and attributes in the property editors and other facilities of the Eclipse platform.

You use the Eclipse PDE to develop the plug-in content, test, and deploy a plug-in. The Eclipse PDE automatically generates the plugin.xml, build.properties, and run-time archive files.

The file, \$INSTALL_DIR\ eclipse\plugins\org.eclipse.birt.report.designer.ui\schema\reportitemUI.exsd, documents the settings for a report item extension to the BIRT Report Designer user interface. The XML schema file, reportitemUI.exsd, has the following structure:

- <schema> is the root element that sets the target namespace and contains all other elements and their attributes.
- <annotation> contains the following attributes:
 - <appinfo> provides the following items:
 - Machine-readable metadata that Eclipse uses to identify the plug-in
 - Text-based information that appears in the PDE Extensions page and HTML extension point description
 - <documentation> provides user information that appears in the PDE's HTML extension point description.

- <element> declares a reference for the model and optional user interface extensions, such as figure, label, image, builder, property page, palette, editor, outline, and description. Each extension element is a complex type containing attributes and annotations, as described below:
 - model is an extension element that specifies the Report Object Model (ROM) name for the report item extension.
 - Each user interface extension element specifies the following items:
 - Extension element name.
 - Fully qualified name of the Java class implementing the interface specified for the extension element. For example, builder implements the interface, org.eclipse.birt.report.designer.ui.extensions .IReportItemBuilderUI.

Listing 17-1 is a partial schema example showing reportitemUI.exsd. The ellipses (...) mark the places in the code where lines are omitted.

Listing 17-1 Partial example schema for Report Item UI

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Schema file written by PDE -->
<schema targetNamespace="org.eclipse.birt.report.designer.ui">
  <annotation>
    <appInfo>
      <meta.schema
        plugin="org.eclipse.birt.report.designer.ui"
        id="reportitemUI"
        name="Report Item UI Extension Point"/>
    </appInfo>
    <documentation>
      This extension point is used in conjunction with the
      Report Item extension point defined in the model. It
      is used to register the GUI to be used for the
      Extended report item.
    </documentation>
  </annotation>
  <element name="extension">
    <complexType>
      <sequence>
        ...
        <element ref="model"/>
        <element ref="builder" minOccurs="0" maxOccurs="1"/>
        <element ref="palette" minOccurs="0" maxOccurs="1"/>
        <element ref="editor" minOccurs="0" maxOccurs="1"/>
        <element ref="outline" minOccurs="0" maxOccurs="1"/>
        <element ref="description" minOccurs="0"
          maxOccurs="1"/>
      </sequence>
      <attribute name="point" type="string" use="required">
```

```

    ...
    </attribute>
</complexType>
</element>
<element name="model">
    <complexType>
        <attribute name="extensionName" type="string"
            use="required">
            <annotation>
                <documentation>
                    The ROM Report Item Extension name that maps
                    to this UI
                </documentation>
            </annotation>
        </attribute>
    </complexType>
</element>
...
<element name="builder">
    <annotation>
        <documentation>
            Optional Builder for the element inside the Editor.
            Instantiated when a new item is dragged from the
            palette inside the editor.
        </documentation>
    </annotation>
    <complexType>
        <attribute name="class" type="string">
            <annotation>
                <documentation>
                    a fully qualified name of the Java class
                    implementing org.eclipse.birt.report
                    .designer.ui.extensions.IReportItemBuilderUI
                </documentation>
            <appInfo>
                <meta.attribute kind="java"/>
            </appInfo>
            </annotation>
        </attribute>
    </complexType>
</element>
...
</schema>
```

Understanding a plug-in manifest file

You install an Eclipse plug-in in a subdirectory of the \$INSTALL_DIR/eclipse/plugins directory. The plug-in manifest file, plugin.xml, describes the plug-in's activation framework to the Eclipse run-time environment.

At run time, Eclipse scans the subdirectories in \$INSTALL_DIR/eclipse/plugins, parses the contents of each plug-in manifest file, and caches the information in the plug-in registry. If the Eclipse run time requires an extension, Eclipse lazily loads the plug-in, using the registry information to instantiate the plug-in's objects. The run-time environment for the BIRT Report Engine functions in a similar way.

The plug-in manifest file declares the plug-in's required code and extension points to the plug-in registry. The plug-in run-time class provides the code segment. By lazily loading the plug-in's code segment, the run-time environment minimizes start-up time and conserves memory resources.

The plug-in manifest file, plugin.xml, has the following structure:

- <plugin> is the root element.
- <extension> specifies the extension points and the related elements and attributes that define the processing capabilities of the plug-in component.

Listing 17-2 shows the contents of the plug-in manifest file, org.eclipse.birt.sample.reportitem.rotatedlabel/plugin.xml. This file describes the required classes and extension points for the BIRT report item extension sample, rotated label.

Listing 17-2 Sample plug-in manifest file

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
    id="rotatedLabel"
    name="Rotated Label Extension"
    point="org.eclipse.birt.report.designer.ui
          .reportitemUI">
    <reportItemLabelUI
      class="org.eclipse.birt.sample.reportitem
            .rotatedlabel.RotatedLabelUI"/>
    <model extensionName="RotatedLabel"/>
    <palette icon="icons/rotatedlabel.jpg"/>
    <editor
      canResize="true"
      showInDesigner="true"
      showInMasterPage="true"/>
    <outline icon="icons/rotatedlabel.jpg"/>
  </extension>
  <extension
    id="rotatedLabel"
    name="Rotated Label Extension"
    point="org.eclipse.birt.report.model.reportItemModel">
    <reportItem
```

```

class="org.eclipse.birt.sample.reportitem
      .rotatedlabel.RotatedLabelItemFactoryImpl"
extensionName="RotatedLabel">
<property
    defaultDisplayName="Display Text"
    defaultValue="Rotated Label"
    name="displayText"
    type="string"/>
<property
    defaultDisplayName="Rotation Angle"
    defaultValue="-45"
    name="rotationAngle"
    type="string"/>
</reportItem>
</extension>
<extension
    id="rotatedLabel"
    name="Rotated Label Extension"
    point="org.eclipse.birt.report.engine.reportitem
          Presentation">
<reportItem
    class="org.eclipse.birt.sample.reportitem.rotated
          label.RotatedLabelPresentationImpl"
    name="RotatedLabel"/> </extension>
<extension
    point="org.eclipse.birt.report.designer.ui
          .elementAdapters">
<adaptable
    class="org.eclipse.birt.report.model.api
          .ExtendedItemHandle">
<adapter
    factory="org.eclipse.birt.sample.reportitem
          .rotatedlabel.views
          .RotatedLabelPageGeneratorFactory"
    id="ReportDesign.AttributeView
          .RotatedLabelPageGenerator"
    priority="1"
    singleton="false"
    type="org.eclipse.birt.report.designer.ui
          .views.IPageGenerator">
<enablement>
<test
    forcePluginActivation="true"
    property="ExtendItemHandle.extensionName"
    value="RotatedLabel">
</test>
</enablement>
</adapter>
</adaptable>
</extension>
</plugin>

```

Understanding a plug-in run-time class

A plug-in runs within an instance of a plug-in run-time class. A plug-in run-time class extends `org.eclipse.core.runtime.Plugin`, the abstract superclass for all plug-in run-time class implementations. The `Plugin` run-time class defines the methods for starting, managing, and stopping a plug-in instance.

The `Plugin` run-time class typically contains a reference to an Open Services Gateway Initiative (OSGi) resource bundle that manages the execution context. `Plugin` implements the interface, `org.osgi.framework.BundleActivator`, which installs, starts, stops, and uninstalls the OSGi resource bundle. The OSGi resource bundle implements a service registry to support the following services:

- Installing the resource bundle
- Subscribing to an event
- Registering a service object
- Retrieving a service reference

The OSGi platform provides a secure, managed, extensible Java framework for deploying, downloading, and managing service applications. For more information about the OSGi platform, visit the OSGi Alliance web site at <http://www.osgi.org/>. For more information about the Java run-time and OSGi APIs, see the Javadoc for the Platform Plug-in Developer Guide in Eclipse Help.

Listing 17-3 is a code example showing the life cycle and resource bundle methods for the report item plug-in, rotated label.

Listing 17-3 Sample code for the rotated label report item plug-in

```
package org.eclipse.birt.sample.reportitem.rotatedlabel;

import org.eclipse.core.runtime.Plugin;
import org.osgi.framework.BundleContext;
import java.util.*;

/**
 * The main plugin class to be used in the desktop.
 */
public class RotatedLabelPlugin extends Plugin {
    // The Plugin ID
    public final static String ID =
        "org.eclipse.birt.sample.reportitem.rotatedlabel";
    //The shared instance.
    private static RotatedLabelPlugin plugin;
    //Resource bundle.
    private ResourceBundle resourceBundle;

    /**

```

```

    * The constructor.
    */
public RotatedLabelPlugin() {
    super();
    plugin = this;
    try {
        resourceBundle = ResourceBundle.getBundle
            ("org.eclipse.birt.sample.reportitem.rotatedlabel
             .RotatedLabelPluginResources");
    } catch (MissingResourceException x) {
        resourceBundle = null;
    }
}

/**
 * This method is called upon plug-in activation
 */
public void start(BundleContext context) throws Exception {
    super.start(context);
}

/**
 * This method is called when the plug-in is stopped
 */
public void stop(BundleContext context) throws Exception {
    super.stop(context);
}

/**
 * Returns the shared instance.
 */
public static RotatedLabelPlugin getDefault() {
    return plugin;
}

/**
 * Returns the string from the plugin's resource bundle,
 * or 'key' if not found.
 */
public static String getResourceString(String key) {
    ResourceBundle bundle =
        RotatedLabelPlugin.getDefault().getResourceBundle();
    try {
        return (bundle != null) ? bundle.getString(key) : key;
    } catch (MissingResourceException e) {
        return key;
    }
}

```

```
/***
 * Returns the plugin's resource bundle,
 */
public ResourceBundle getResourceBundle() {
    return ResourceBundle;
}
}
```

Working with the Eclipse PDE

The Eclipse PDE is an integrated design tool that you use to create, develop, test, debug, and deploy a plug-in. The PDE provides wizards, editors, views, and launchers to assist you in developing a plug-in.

The Eclipse PDE provides a wizard to assist you in setting up a plug-in project and creating the framework for a plug-in extension. In the Plug-in Development perspective, you can use the New Plug-in Project wizard to assist you in setting up a plug-in project and creating the framework for a plug-in extension. The PDE wizard automatically generates the plug-in manifest file, plugin.xml, and optionally, the Java plug-in run-time class.

How to choose the Plug-in Development perspective

To access the PDE, you must choose the Plug-in Development perspective. To open the Plug-in Development perspective, perform the following tasks:

- 1 From the Eclipse menu, choose Window→Open Perspective→Other. Open Perspective appears.
- 2 Select Plug-in Development, as shown in Figure 17-1.

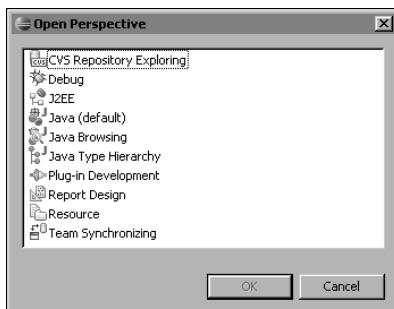


Figure 17-1 Selecting a perspective

Choose OK. The Plug-in Development perspective appears.

How to set up a new plug-in project

To access the New Plug-in Project wizard and create a project, perform the following tasks:

- 1 From the PDE menu, choose File→New→Project. New Project appears.

- 2** In Wizards, expand Plug-in Development, and select Plug-in Project, as shown in Figure 17-2.

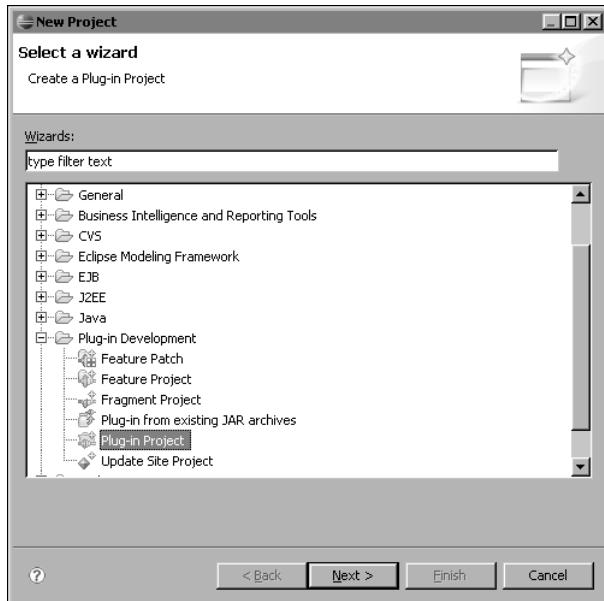


Figure 17-2 New Project

Choose Next. New Plug-in Project appears, as shown in Figure 17-3.

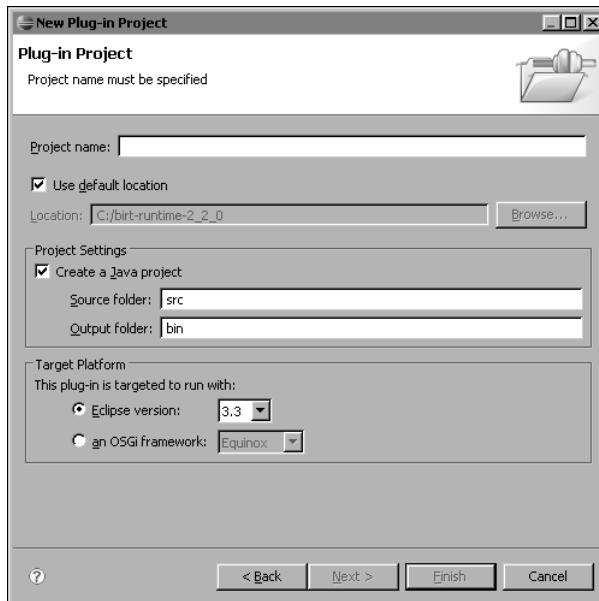


Figure 17-3 New Plug-in Project

Understanding plug-in project properties

Using the New Plug-in Project wizard, you can define the following properties for the plug-in:

- Project settings
 - Name
 - Location
 - Source and output folders
 - Target platform, specifying the Eclipse version or OSGi framework
- Plug-in content
 - Properties such as ID, version, name, provider, and the run-time library classpath
 - Choose to generate an activator, a Java class that controls the plug-in's life cycle

In Eclipse 3.3, specifying the OSGi framework creates an OSGi bundle manifest, META-INF/ MANIFEST.MF, which contains a set of manifest headers that provide descriptive information about the bundle.

Eclipse 3.3 uses an implementation of the OSGi R4 framework specification. This framework defines the directives that specify the access rules for an exported package and the headers that facilitate class loading, start-up time, filtering, and other features.

BIRT release 2.2.1 uses the OSGi framework that comes with the Eclipse 3.3 platform. You must implement any plug-in that extends a BIRT release 2.2.1 extension point as an OSGi bundle.

Inside the Eclipse environment, you can develop and deploy a plug-in for BIRT Report Designer. Outside of the Eclipse environment, you can develop and deploy a plug-in for a web viewer. BIRT Web Viewer is a sample J2EE web application consisting of servlets and JSPs that encapsulates the BIRT Report Engine API to generate reports.

Understanding the Eclipse PDE Workbench

The Eclipse PDE supports host and run-time instances of the workbench project. The host instance provides the development environment. The run-time instance allows you to launch a plug-in to test it.

Figure 17-4 shows the project for the report item extension sample, rotated label, in the host instance of the PDE Workbench. In the host instance, the PDE Workbench provides the following view and editor components:

- Package Explorer provides an expandable view of the plug-in package.
- Outline provides an expandable view of the project settings.

- PDE Manifest Editor displays a page containing the project settings for the currently selected item in Outline.

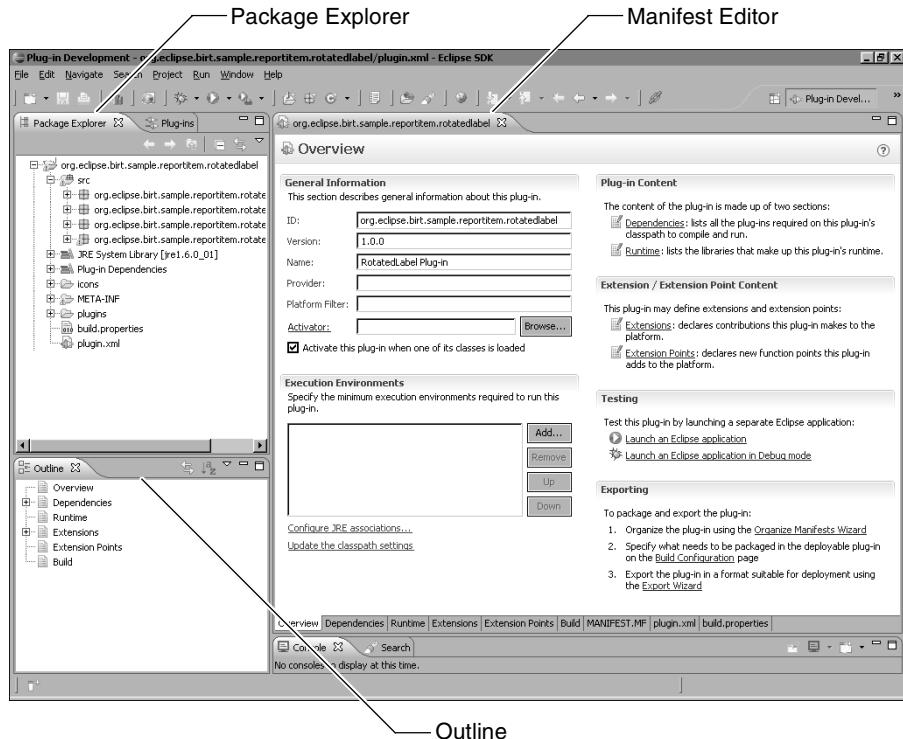


Figure 17-4 The host instance of the PDE Workbench

In PDE Manifest Editor, you specify project settings and edit related files to create the plug-in framework on the following pages:

- Overview

Lists general information such as the plug-in ID, version, name, provider, platform filter, and activator class. This page also contains sections that link to the plug-in content pages, extensions, launchers for testing and debugging, deployment wizards, and the settings for the execution environment. In Figure 17-4, PDE Manifest Editor displays the Overview page.

- Dependencies

Lists the plug-ins that must be on the classpath to compile and run.

- Runtime

Declares the packages that the plug-in exposes to clients, the package visibility to other plug-ins, and the libraries and folders in the plug-in classpath.

- Extensions
Declares the extensions that the plug-in makes to the platform.
 - Extension Points
Declares the new extension points that the plug-in adds to the platform.
 - Build
Displays the build configuration settings. A change to a setting on this page updates the file, build.properties.
 - MANIFEST.MF
Displays an editable page containing the header settings for the manifest file, MANIFEST.MF, that provide descriptive information about an OSGi bundle.
 - Plug-in.xml
Displays an editable page containing the settings for the plug-in manifest file, plugin.xml.
 - Build.properties
Displays an editable page containing the settings for the file, build.properties.
- A modification to a setting in a PDE Manifest Editor page automatically updates the corresponding plug-in manifest or build properties file.

Creating the structure of a plug-in extension

Use the host instance of the PDE Workbench to create the basic structure of a plug-in extension by performing the following tasks:

- Specifying the plug-in dependencies
- Verifying the plug-in run-time archive
- Specifying the plug-in extension

How to specify the plug-in dependencies

- 1 On PDE Manifest Editor, choose Overview.
- 2 In Plug-in Content, choose Dependencies.
- 3 In Required Plug-ins, choose Add. Plug-in Selection appears, as shown in Figure 17-5.

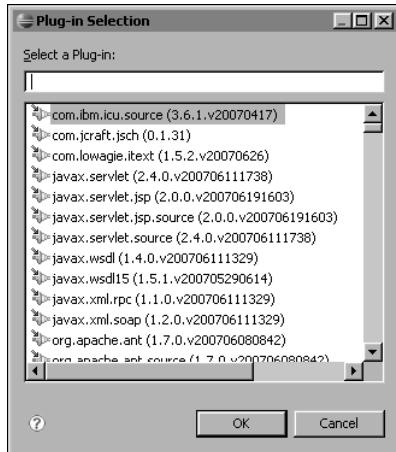


Figure 17-5 Plug-in Selection

- 4** In Plug-in Selection, select a plug-in, such as the following example:

`org.eclipse.birt.report.designer.ui`

Choose OK.

- 5** Repeat steps 2 and 3 to add more plug-ins to the list of required plug-ins in the Dependencies page.

In Required Plug-ins, the order of the list determines the sequence in which a plug-in loads at run time. Use Up and Down to change the loading order as necessary.

Figure 17-6 shows an example of a list of dependencies for a plug-in extension.

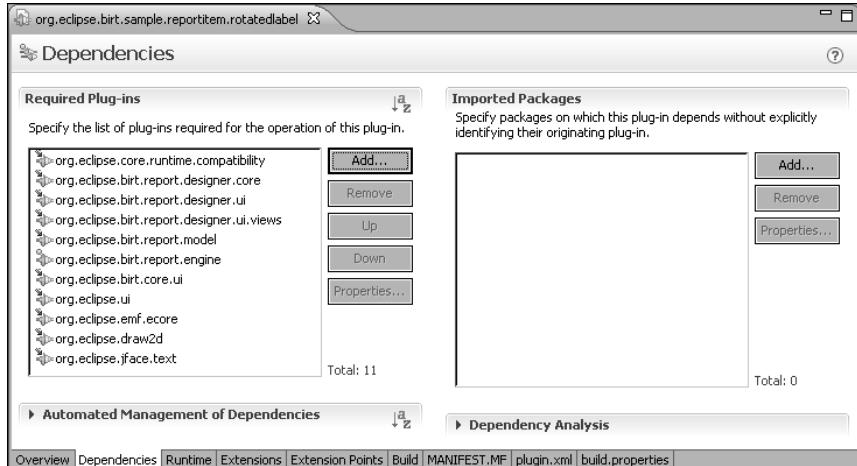


Figure 17-6 The Dependencies page

How to verify the plug-in run-time archive

- 1 On PDE Manifest Editor, choose Runtime. Runtime appears, as shown in Figure 17-7.

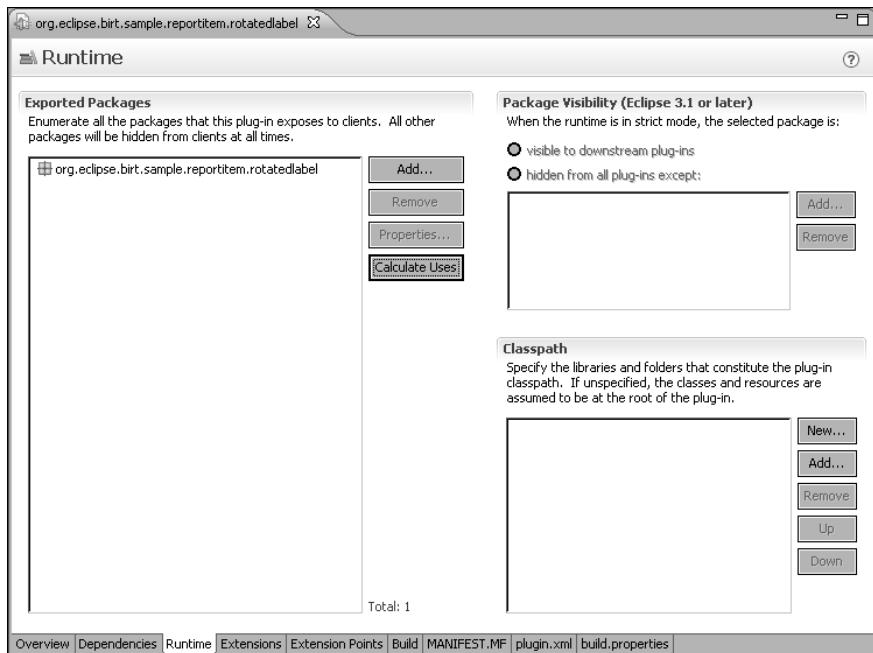


Figure 17-7 The Runtime page

- 2 In Runtime, perform the following tasks:
 - In Exported Packages, list all the packages that the plug-in exposes to clients.
 - In Package Visibility, when the plug-in is in strict run-time mode, indicate whether a selected package is one of the following options:
 - ❑ Visible to downstream plug-ins
 - ❑ Hidden except for the specified plug-ins
 - In Classpath, choose Add to add the name of an archive file or folder to the classpath manually.

How to specify the plug-in extension

- 1 On PDE Manifest Editor, choose Extensions.
- 2 In All Extensions, choose Add. New Extension appears.
- 3 On Extension Point Selection, in Extension Points, select a plug-in, such as the following example:

`org.eclipse.birt.report.designer.ui.reportitemUI`

New Extension appears, as shown in Figure 17-8.

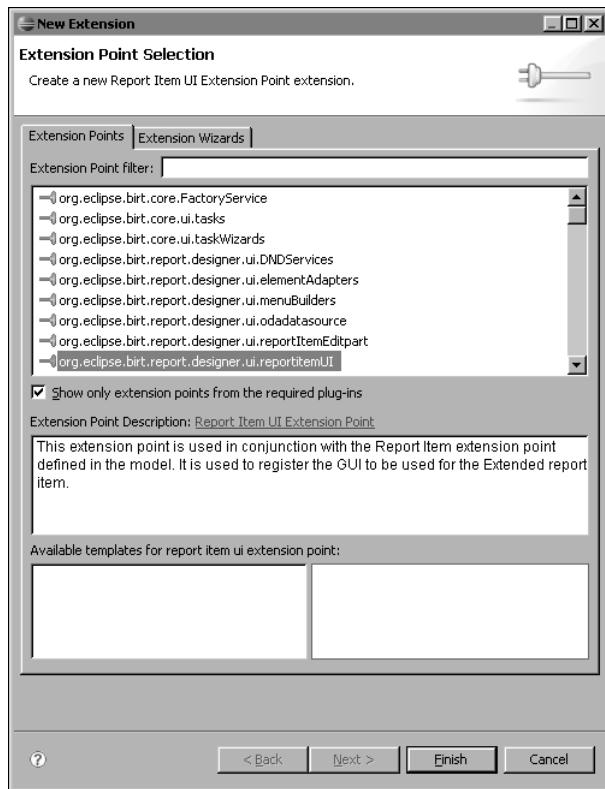


Figure 17-8 New Extension—Extension Point Selection

Choose Finish. Extensions appears, as shown in Figure 17-9.

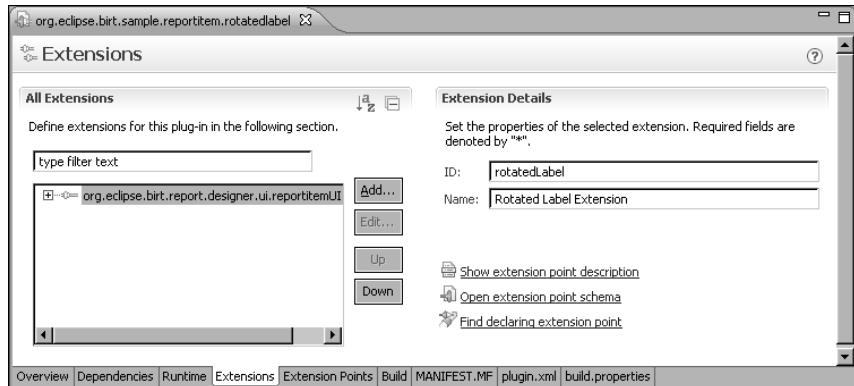


Figure 17-9 The Extensions page

- 4 Repeat steps 2 and 3 to add more plug-ins to the list of required extension points in the Extensions page.

Creating the plug-in extension content

The XML schema specifies a grammar that you must follow when creating an extension in the Eclipse PDE. When you select an element of an extension in the Extensions page of the PDE, Eclipse uses the XML schema to populate the Extension Element Details section with the list of valid attributes and values for the element.

On Extensions, if you choose Find declaring extension point, the PDE searches for the extension point currently selected in All Extensions. If you choose Show extension point description, the PDE generates an HTML page containing the information documented in the XML schema and displays the page in a viewer. If you choose Open extension point schema, the PDE opens the .exsd file that contains the XML schema definitions.

This section discusses the following tasks:

- Searching for and viewing extension point information
- Specifying plug-in extension content
- Specifying a build configuration

How to search for and view extension point information

- 1 In the Eclipse PDE, select a plug-in extension in All Extensions.
- 2 In Extension Details, choose Find declaring extension point.

Search appears. In Figure 17-10, Search lists one match, org.eclipse.birt.report.designer.ui.reportitemUI.

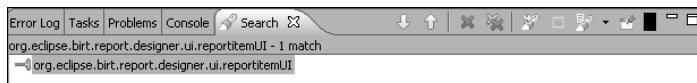


Figure 17-10 Search showing a single match

- 3 In Search, double-click on the match, such as org.eclipse.birt.report.designer.ui.reportitemUI. In PDE Manifest Editor, a window appears, displaying the contents of the file, org.eclipse.birt.report.designer.ui/plugin.xml, as shown in Figure 17-11.



Figure 17-11 Plugin.xml showing three extension points

Plugin.xml describes the extension points, odadatasource, reportItemUI, menuBuilders, elementAdapters, reportItemEditpart, and DNDServices.

- 4 In PDE Manifest Editor, close the window displaying contents of the plugin.xml file. Choose Extensions.
- 5 In Extension Details, choose Open extension point schema.

A viewer opens, displaying the HTML document for the extension point. In Figure 17-12, the viewer displays Report Item UI Extension Point, containing information extracted from the XML schema, \$INSTALL_DIR\ eclipse\plugins\org.eclipse.birt.report.designer.ui\schema\ reportitemUI.exsd.

The screenshot shows a browser window with the title 'Report Item UI Extension Point'. Below the title, there is a section labeled 'Identifier: org.eclipse.birt.report.designer.ui.reportitemUI'. Underneath this, 'Since: 1.0' is listed. A bolded 'Description:' section follows, stating: 'This extension point is used in conjunction with the Report Item extension point defined in the model. It is used to register the GUI to be used for the Extended report item.' Below the description, a 'Configuration Markup:' section contains an XML snippet:

```
<!ELEMENT extension ((reportItemFigureUI | reportItemLabelUI | reportItemImageUI) ,  
model , builder? , palette? , editor? , outline? , description?)>  
<!ATTLIST extension  
point CDATA #REQUIRED  
id CDATA #IMPLIED  
name CDATA #IMPLIED>  
  
<!ELEMENT model EMPTY>  
<!ATTLIST model  
extensionName CDATA #REQUIRED>  
• extensionName - The ROM Report Item Extension name that maps to this UI
```

Figure 17-12 Extension point description

In the HTML document, Configuration Markup displays the attribute list for the extension point. Scroll down to view all the contents of the HTML document, including the optional set of user interface elements for the report item extension, such as builder, palette, editor, outline, and description.

How to specify plug-in extension content

- 1 In PDE Manifest Editor, choose Extensions.
- 2 In All Extensions, right-click on an extension point, such as org.eclipse.birt.report.designer.ui.reportItemUI, and choose New-><extension point element>.

Figure 17-13 shows how to select the extension point element, reportItemLabelUI.

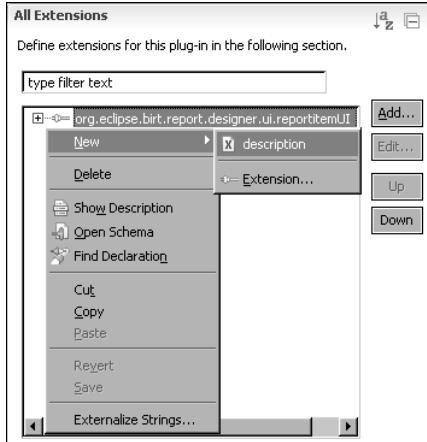


Figure 17-13 Context menu for selecting an extension point element

Extensions appears, displaying the extension element and its details, as shown in Figure 17-14.

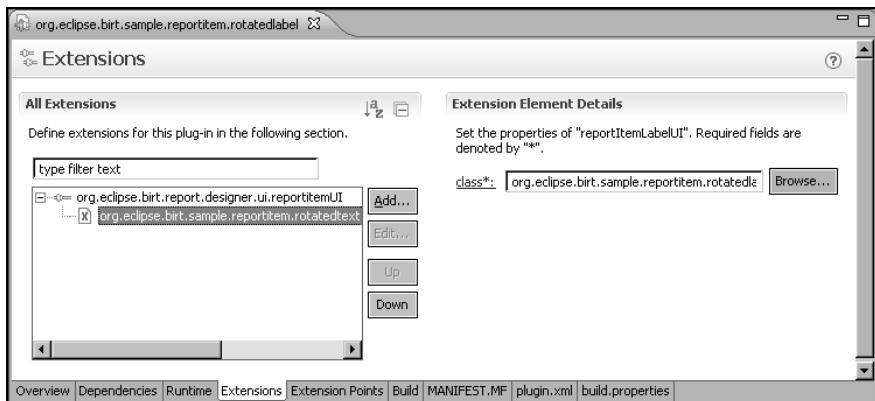


Figure 17-14 The Extensions page

In this example, All Extensions lists the extension, `org.eclipse.birt.sample.reportitem.rotatedlabel.RotatedLabelUI` (`rotatedItemLabelUI`), and Extension Element Details lists `rotatedItemLabelUI` properties. In Extension Element Details, the label for a required attribute, such as `class`, contains an asterisk.

- 3 To view the annotation for a property listed in Extension Element Details, hover the cursor over the property label.

A Tooltip appears, displaying the annotation for the property from the XML schema. Figure 17-15 shows the annotation for the `class` property for the example extension element, `rotatedItemLabelUI`.

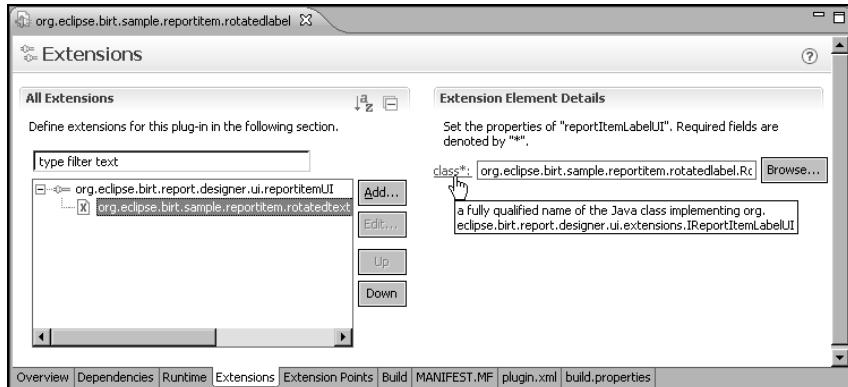


Figure 17-15 Annotation for an extension element

- 4 To specify the class attributes for an extension element, choose class in Extension Element Details.

If no class file exists, Java Attribute Editor appears, as shown in Figure 17-16. If the class file exists, the class file opens in PDE Manifest Editor.

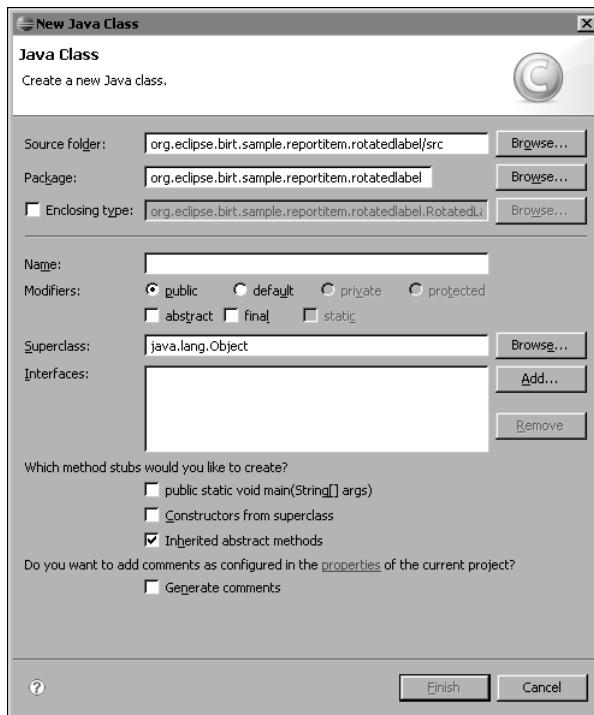


Figure 17-16 Java Attribute Editor

On Java Attribute Editor, you can modify or add to the settings for the following class properties:

- Source folder
- Package
- Enclosing type
- Class name
- Modifiers, such as public, default, private, protected, abstract, final, and static
- Superclass
- Interfaces
- Method stubs, such as main, constructors, and inherited abstract methods
- Comments

Choose Finish.

- 5** To add more elements and attributes to a selected extension point, repeat steps 1 and 2.

Figure 17-17 shows the full list of extension points required for the sample report item extension, org.eclipse.birt.sample.reportitem.rotatedlabel.

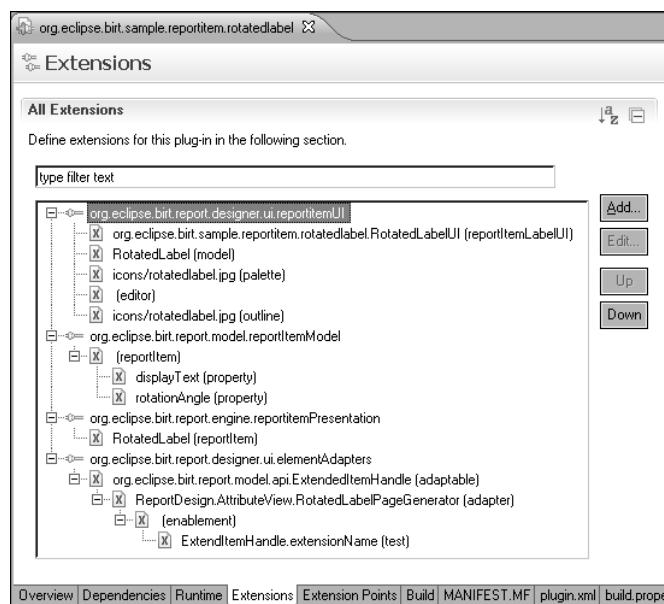


Figure 17-17 List of required extension points

Building a plug-in extension

In Eclipse PDE Manifest Editor, Build allows you to specify the build configuration, including the following items:

- Runtime Information

Defines the libraries, the source folders to compile into each library, and the compilation order.

- Binary Build

Selects the files and folders to include in the binary build.

- Source Build

Selects the files and folders to include in the source build. Source Build is not typically required. Source Build uses the org.eclipse.pde.core.source extension point that allows the PDE to find source archives for libraries in other Eclipse plug-ins.

How to specify a build configuration

- 1 On PDE Manifest Editor, choose Build. Build Configuration appears. Figure 17-18 shows Build Configuration.

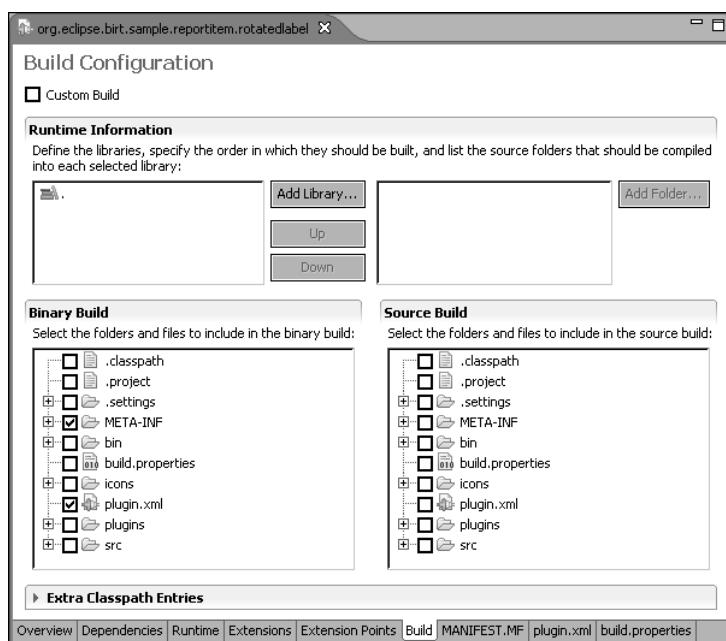


Figure 17-18 Build Configuration

- 2 In Runtime Information, add a new library by choosing Add Library. Add Entry appears.

Enter the new library name or select a run-time library from the list, as shown in Figure 17-19. Choose OK.

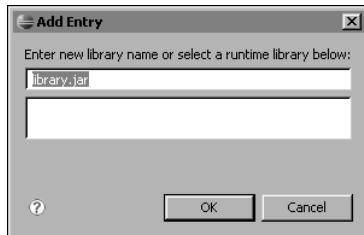


Figure 17-19 Add Entry

- 3** To change the compilation order of a library, change its position in the list. In Runtime Information, select the library and choose Up or Down.

Figure 17-20 shows Runtime Information with mylibrary.jar selected and Down enabled.

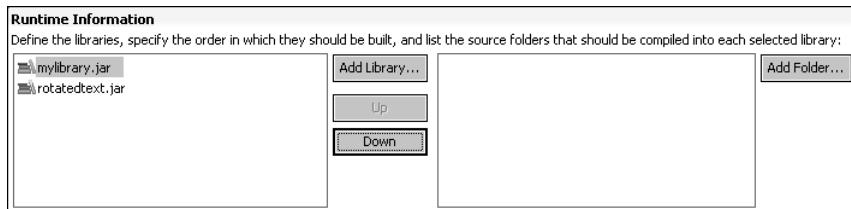


Figure 17-20 Changing the compilation order of a library

- 4** To add a folder to a library, choose Add Folder. New Source Folder appears, as shown in Figure 17-21.



Figure 17-21 New Source Folder

Select a folder, such as src, and choose OK. Runtime Information appears, as shown in Figure 17-22.

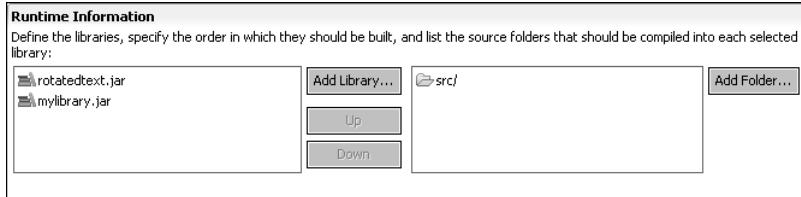


Figure 17-22 Runtime Information

- 5** In Binary Build, include a folder in the binary build by selecting the folder. Figure 17-23 shows the icons folder selected.

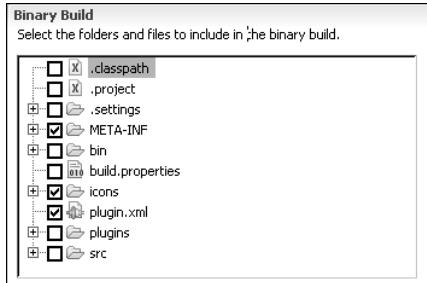


Figure 17-23 Binary Build

- 6** From the Eclipse menu, choose Project->Build All, to build a project.

Alternatively, you can choose Project->Build Automatically to build the project continually as you make changes to the code.

Generating an Ant build script

The Eclipse PDE can generate an Ant build script for compiling plug-in code, based on the settings in the build.properties file. The generated script is an XML file in which the elements are the required tasks for the build operation. The Ant build tool compiles the project, using the specified Java compiler.

Ant is an open source Java application available from the Apache Software Foundation. For more information on Ant and the Apache Software Foundation, visit the web site at <http://ant.apache.org>.

How to generate an Ant build script

In Package Explorer, right-click the project's plugin.xml file and choose PDE Tools->Create Ant Build File. PDE Tools creates an Ant script file, build.xml, in the project folder.

Testing a plug-in extension

You can launch an instance of the run-time workbench to test and debug the plug-in extension.

How to launch a run-time workbench

- 1 On PDE Manifest Editor, choose Overview. Overview appears as shown in Figure 17-24.

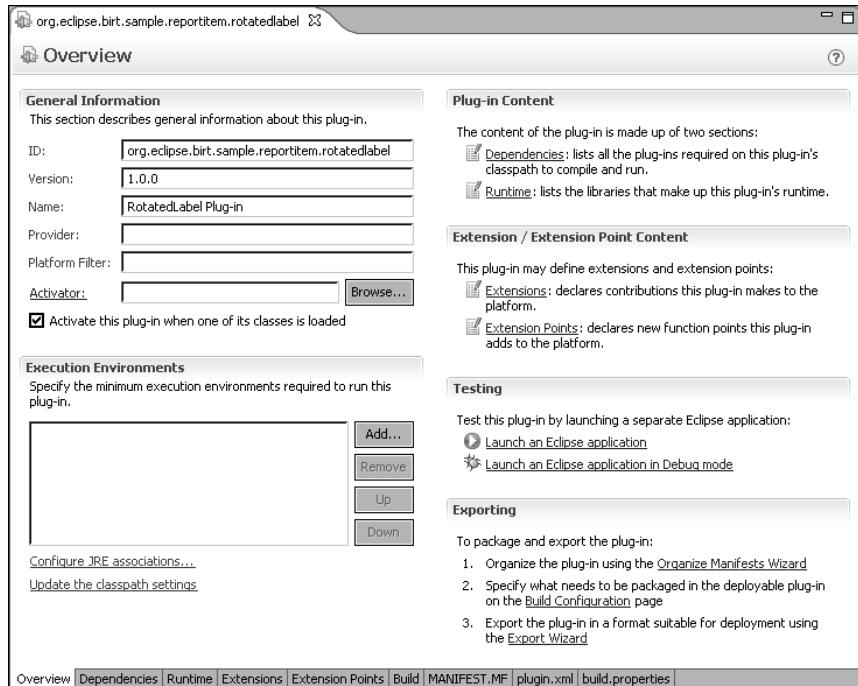


Figure 17-24 Overview showing testing and debugging options

- 2 In Testing, choose Launch an Eclipse application. Eclipse launches the run-time workbench.

In the report item extension example, Report Design—Eclipse SDK appears. In the run-time workbench, you must create a new report design project to use the report label extension.

Deploying the extension plug-in

You can use the Export Wizard to produce a distributable archive file that contains the plug-in code and other resources. A user can find a software update and extract the contents of the archive file to an Eclipse installation using the Feature Updates and Product Configuration managers. A plug-in developer can create and manage an update site using the Update Site Editor in the Eclipse PDE.

How to deploy a plug-in extension

- 1 In the Eclipse PDE Manifest Editor, choose Overview.

- 2** In Exporting, choose Export Wizard. Export appears.
- 3** In Available Plug-ins and Fragments, select the plug-in to export. For example, select org.eclipse.birt.sample.reportitem.rotatedlabel.
- 4** In Export Destination, specify Archive file or Directory. For example, in Directory, type:
`C:\birt-runtime-2_2_0\ReportEngine\plugins`
- 5** In Export Options, select one of the following options, if necessary:
 - Include source code
 - Package plug-ins as individual JAR archives
 - Save as Ant script

Export appears as shown in Figure 17-25. Choose Finish to export the plug-in to the specified destination.

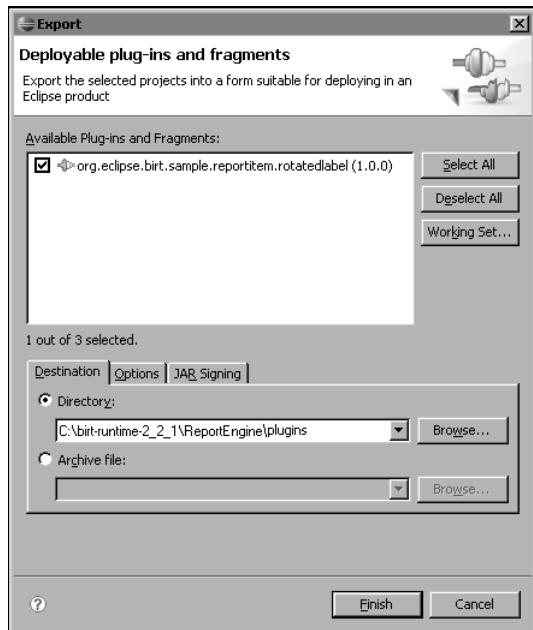


Figure 17-25 Exporting a plug-in

Installing feature updates and managing the Eclipse configuration

If all the dependent resources are available in the new environment, Eclipse can discover and activate the plug-in in the run-time environment. In this type of unmanaged distribution and installation, the user must find and install updates to the plug-in if a release occurs in the future.

A plug-in developer can also use a more structured approach and group plug-ins into features. A feature is a set of plug-ins that you install and manage together.

Features contain information that enable the Feature Updates and Product Configuration managers to locate published updates and discover new related features. Updates are typically published in a special internet directory called an update site, created and managed by a plug-in developer using the Update Site Editor.

How to install feature updates and manage the Eclipse configuration

The Eclipse PDE provides wizards and editors that support the use of features and update sites. Choose Help→Software Updates→Find and Install to access Feature Updates. Feature Updates allows you to search for updates and new features, as shown in Figure 17-26.

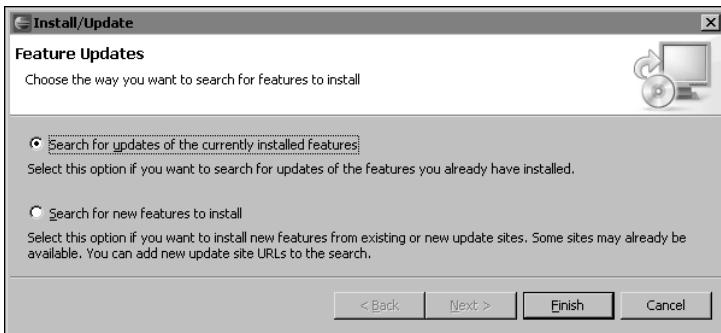


Figure 17-26 Searching for feature updates

Choose Help→Software Updates→Manage Configuration to access Product Configuration. Figure 17-27 shows Product Configuration.

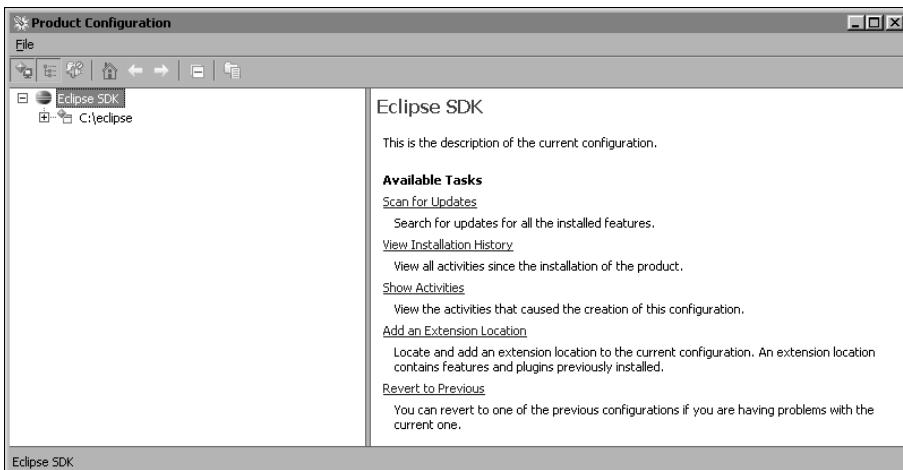


Figure 17-27 Product Configuration

Product Configuration supports performing the following tasks:

- Scan for updates.
- View installation history.
- Show activities that created the current configuration.
- Add an extension location that contains features and plug-ins previously installed.
- Revert to a previous configuration.

Creating an update site project

You create an update site by building an update site project in the Eclipse PDE workspace. The update site project contains a manifest file, site.xml, that lists the features and plug-ins packages.

The build operation for an update site puts the JAR files for features in a features folder and the JAR files for plug-ins in a plug-ins folder. The Eclipse PDE also provides support for uploading an update site to a remote server or local file system for distribution.

How to create an update site project

- 1 From the Eclipse menu, choose File->New->Project. New Project appears.
- 2 In Wizards, open Plug-in Development and select Update Site Project, as shown in Figure 17-28. Choose Next. Update Site Project appears.

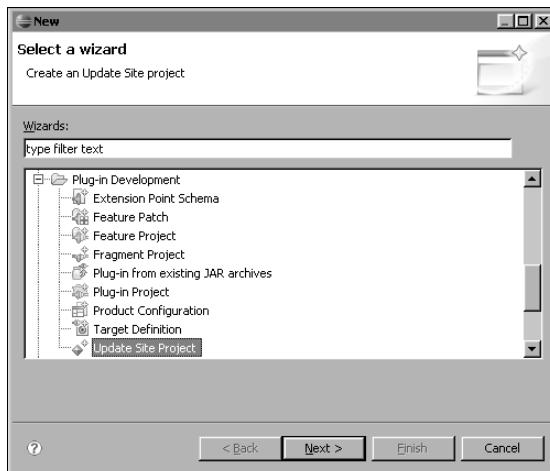


Figure 17-28 Selecting Update Site Project wizard

- 3 On Update Site Project, specify the following items:
 - Project name
 - Project contents directory, such as C:/birt-runtime-2_2_0/BIRT Update Site

- Web resources

- Select the option, Generate web page listing of all available features within the site.

Creates index.html, site.css, and site.xls files for displaying the contents of the update site.

- Web resources location

Change this setting to the web resources location. The default value is web.

Update Site Project appears as shown in Figure 17-29. Choose Finish. Update Site Map appears as shown in Figure 17-30.

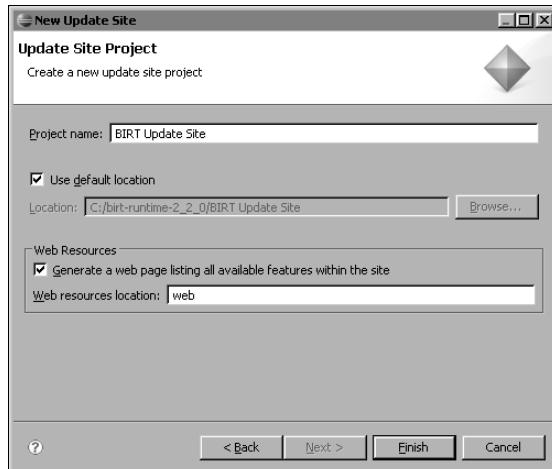


Figure 17-29 Creating a new update site project

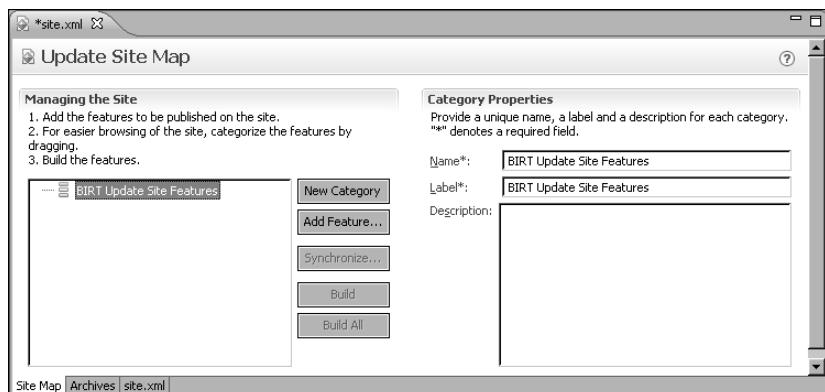


Figure 17-30 Update Site Map

4 Choose New Category to create a feature category.

5 Choose Add Feature to add a feature to a selected category.

For more information about deploying a plug-in, installing features, managing a product configuration, and building an update site, see the Javadoc for the Platform Plug-in Developer Guide in Eclipse Help.

Downloading the code for the extension examples

This book provides examples for the following types of BIRT extensions:

- Report item

The example shows how to build a rotated label report item plug-in and add the report item to the BIRT Report Designer using the defined extension points. This plug-in renders the label of a report item as an image. The extension rotates the image in a report design to display the label at a specified angle.

- Report rendering

The example shows how to extend the emitter interfaces to build and deploy a report rendering plug-in that runs in the BIRT Report Engine environment. The CSV extension example is a plug-in that writes the table data in a report to a file in CSV format. The XML extension example is a plug-in that writes the table data in a report to a file in XML format.

- ODA drivers

The CSV ODA driver example is a plug-in that reads data from a CSV file. The Hibernate ODA driver example uses HQL to provide a SQL-transparent extension that makes the ODA extension portable to all relational databases.

These examples also show how to develop an ODA extension to the BIRT Report Designer 2.1 user interface so that a report developer can select an extended ODA driver.

- Plug-in fragment

The plug-in fragment example adds Spanish and Japanese localization features for the BIRT Report Viewer to an existing plug-in.

You can download the source code for these extension examples at <http://www.actuate.com/birt/contributions>.

18

Developing a Report Item Extension

This chapter describes how to create a BIRT extension in the Eclipse PDE using the sample report item extension, rotated label. You learn about creating a BIRT report item extension in the following sections:

- Understanding a report item extension
- Developing the sample report item extension
- Understanding the sample report item extension
- Building, deploying, and testing the rotated label report item plug-in

Understanding a report item extension

A report item extension adds a new type of report item to the BIRT framework by implementing multiple extension points. A plug-in that defines an extension point typically contains an XML extension point schema definition (.exsd) file in a schema subdirectory. This XML schema describes the elements, attributes, and types used by the extension point to the Eclipse PDE environment.

To add a new report item, a report item extension implements the following extension points:

- Report item model
 - org.eclipse.birt.report.model.reportItemModel specifies the report item extension point for the ROM. The XML schema file, org.eclipse.birt.report.model/schema/reportItem.exsd, describes this extension point.

- Report item user interface

`org.eclipse.birt.report.designer.ui.reportitemUI` specifies the report item extension point for the user interface in the report layout editor and report item palette. The XML schema file, `org.eclipse.birt.report.designer.ui/schema/reportitemUI.exsd`, describes this extension point.

- Report item query (optional)

`org.eclipse.birt.report.engine.reportitemQuery` specifies the extension point for query preparation support in the BIRT designer and report engine. A query preparation extension is optional. If the report item does not require query preparation, you can omit the query extension. The XML schema file, `org.eclipse.birt.report.engine/schema/reportitemQuery.exsd`, describes this extension point.

- Report item run time (optional)

`org.eclipse.birt.report.engine.reportitemGeneration` specifies the extension point for instantiating, processing, and persisting a new report item at report generation time. The XML schema file, `org.eclipse.birt.report.engine/schema/reportitemGeneration.exsd`, describes this extension point.

`org.eclipse.birt.report.engine.reportitemPresentation` specifies the extension point for instantiating, processing, and rendering a new report item at report presentation time. The XML schema file, `org.eclipse.birt.report.engine/schema/reportitemPresentation.exsd`, describes this extension point.

- Report item UI property page adapters

`org.eclipse.birt.report.designer.ui.elementAdapters` specifies the extension points for the adapters for the report item property page UI. The XML schema file, `org.eclipse.birt.report.designer.ui/schema/elementAdapters.exsd`, describes this extension point.

- Report item emitter (optional)

`org.eclipse.birt.report.engine.emitters` specifies the extension point for support of a new output format in the presentation engine. The XML schema file, `org.eclipse.birt.report.engine/schema/emitters.exsd`, describes this extension point.

At run time, the BIRT Report Engine performs the following processing on a report item before rendering the final output:

- Query preparation

Gathers the data binding information and expressions defined for the report, passing the information to the report engine. The data engine prepares the data access strategy based on this information.

- Generation

Creates the instances of report items and fetches the data.

- **Presentation**
Renders the report item to a supported data primitive, such as an image, string, HTML segment, or custom data component.
- **Emitter**
Converts the output to a specified format, such as HTML or PDF.

This chapter provides an example of a custom report item extension, `org.eclipse.birt.sample.reportitem.rotatedlabel`. The sample code for the rotated label report item extension creates a text element that renders label text at a specified angle.

The standard report item plug-in, chart, is a more complex example of a report item extension. The BIRT chart plug-in implements user interface and report engine extensions that support a report design using any of the following chart types:

- Bar
- Line
- Pie
- Scatter
- Stock

For reference documentation for the report item API, see the Javadoc for the `org.eclipse.birt.report.engine.extension` package in BIRT Programmer Reference in Eclipse Help.

Developing the sample report item extension

The Report Item extension framework allows a report developer to create a customized report item in the palette of BIRT Report Designer. You can use a report item extension in a report design in the same way you use a standard report item, such as Label, Text, Grid, Table, or Chart.

The sample code for the rotated label report item extension creates a label element that renders text at a specified angle. This section describes the steps required to implement the `org.eclipse.birt.sample.reportitem.rotatedlabel` sample. To implement the rotated label report item extension, perform the following tasks:

- Configure the plug-in project.

You can build the rotated label report item plug-in manually by following the instructions in this chapter.

- Add the report item to the Report Designer UI.

Extend the Report Item UI extension point,
`org.eclipse.birt.report.designer.ui.reportItemUI`.

- Add the report item property pages to the Report Designer UI.
Extend the Report Item UI property page extension point,
`org.eclipse.birt.report.designer.ui.elementAdapters`.
- Add the report item definition to the ROM.
Extend the Report Item Model extension point,
`org.eclipse.birt.report.model.reportItemModel`.
- Add the report item run-time behavior.
Extend the Presentation extension point,
`org.eclipse.birt.report.engine.reportItemPresentation`.
- Deploy the report item extension.
Export the rotated label report item plug-in folder from your workspace to the `eclipse\plugins` folder. You do not need to export the plug-in folder to test the extension when you launch it as an Eclipse application in the PDE.

You can download the source code for the rotated label report item extension example at <http://www.actuate.com/birt/contributions>.

Downloading BIRT source code from the CVS repository

Eclipse makes BIRT source code available to the developer community in the CVS repository. You work only with the Java classes in the `org.eclipse.birt.sample.reportitem.rotatedlabel` plug-in.

To compile, you do not need the source code for any required plug-ins. By default, the system uses the JAR files in the `$INSTALL_DIR\clipse\plugin` folder.

These plug-ins must be in the classpath to compile successfully. To debug, you may need the source code for all the required BIRT plug-ins.

The rotated label report item extension depends on the following Eclipse plug-ins:

- `org.eclipse.core.runtime.compatibility`
- `org.eclipse.birt.report.designer.core`
- `org.eclipse.birt.report.designer.ui`
- `org.eclipse.birt.report.designer.ui.views`
- `org.eclipse.birt.report.engine`
- `org.eclipse.birt.core.ui`
- `org.eclipse.ui`
- `org.eclipse.draw2d`

Creating a rotated label report item plug-in project

Create a new plug-in project for the rotated label report item extension in the Eclipse PDE.

How to create the plug-in project

- 1 In the Eclipse PDE, choose File→New→Project. New Project appears.
- 2 In Select a wizard, select Plug-in Project. Choose Next. New Plug-in Project appears.
- 3 In Plug-in Project, modify the settings, as shown in Table 18-1.

Table 18-1 Settings for Plug-in Project fields

Section	Option	Value
Plug-in Project	Project name	org.eclipse.birt.sample.reportitem.rotatedlabel
	Use default location	Selected
	Location	Not available when you select Use default location
Project Settings	Create a Java project	Selected
	Source folder	src
	Output folder	bin
Target Platform	Eclipse version	3.3
	OSGi framework	Deselected

Plug-in Project appears, as shown in Figure 18-1.

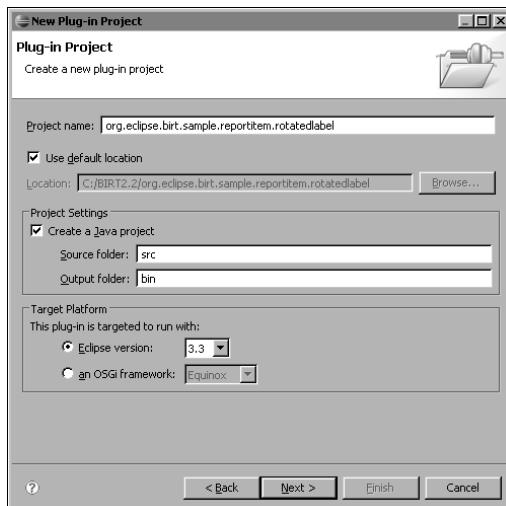


Figure 18-1 Plug-in Project settings

Choose Next. Plug-in Content appears.

- 4 In Plug-in Content, modify the settings as shown in Table 18-2.

Table 18-2 Plug-in Content settings

Section	Option	Value
Plug-in Properties	Plug-in ID	org.eclipse.birt.sample.reportitem.rotatedlabel
	Plug-in Version	1.0.0
	Plug-in Name	RotatedLabel Plug-in
	Plug-in Provider	yourCompany.com or leave blank
Plug-in Options	Classpath	rotatedLabel.jar or leave blank
	Generate an activator, a Java class that controls the plug-in's life cycle	Selected
	Activator	org.eclipse.birt.sample.reportitem.rotatedlabel.RotatedLabelPlugin
	This plug-in will make contributions to the UI	Deselected
Rich Client Application	Would you like to create a rich client application?	No

New Plug-in Content appears, as shown in Figure 18-2. Choose Finish.

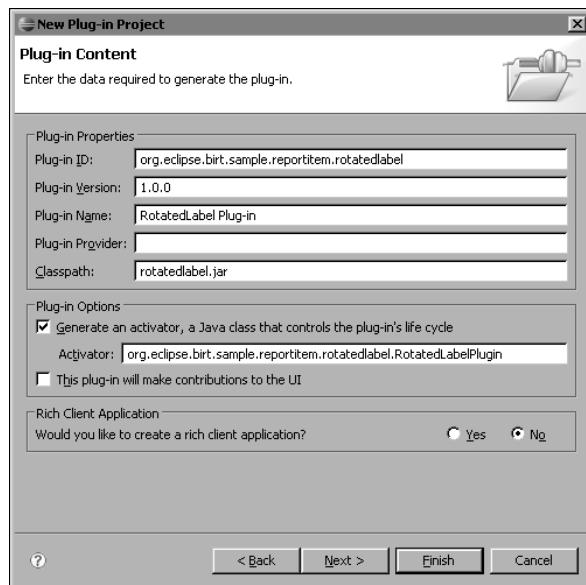


Figure 18-2 Plug-in Content showing settings for a new plug-in project

The rotated label report item extension project appears in the Eclipse PDE Workbench, as shown in Figure 18-3.

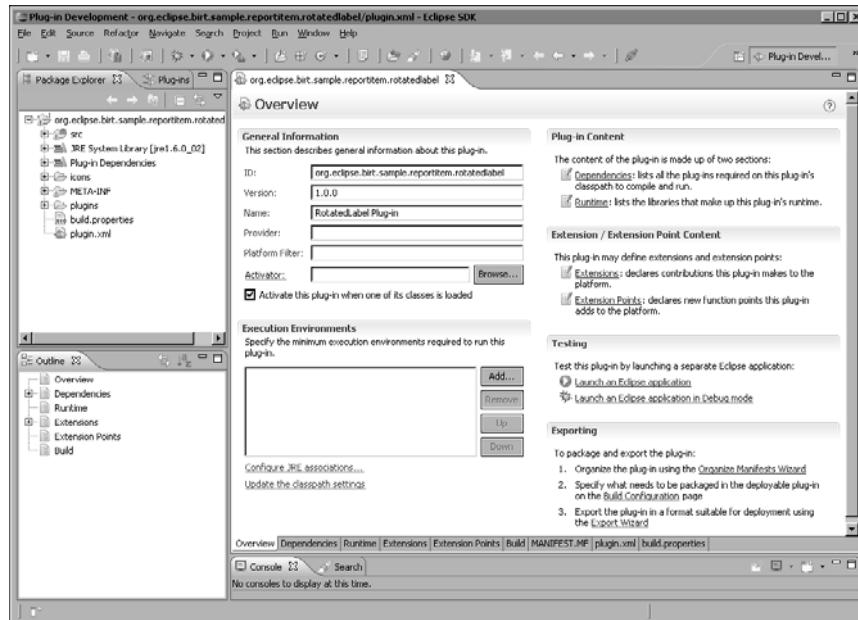


Figure 18-3 Plug-in project in the Eclipse PDE Workbench

Defining the dependencies for the rotated label report item extension

In this task, you specify the list of plug-ins that must be available on the classpath of the rotated label report item extension to compile and run.

How to specify the dependencies

- 1 On PDE Manifest Editor, choose Overview.
- 2 In Plug-in Content, choose Dependencies. Required Plug-ins contains the following plug-ins:

org.eclipse.ui
org.eclipse.core.runtime

- 3 In Required Plug-ins, perform the following tasks:

- 1 Select org.eclipse.ui and choose Remove.
- 2 Select org.eclipse.core.runtime and choose Remove.

org.eclipse.core.runtime and org.eclipse.ui no longer appear in Required Plug-ins.

- 4 In Required Plug-ins, choose Add. Plug-in Selection appears.
- 5 In Plug-in Selection, hold down CTRL and select the following plug-ins:
 - org.eclipse.core.runtime.compatibility
 - org.eclipse.birt.report.designer.core
 - org.eclipse.birt.report.designer.ui
 - org.eclipse.birt.report.designer.ui.views
 - org.eclipse.birt.report.engine
 - org.eclipse.birt.core.ui
 - org.eclipse.ui
 - org.eclipse.draw2d

Choose OK. Dependencies appears, as shown in Figure 18-4.

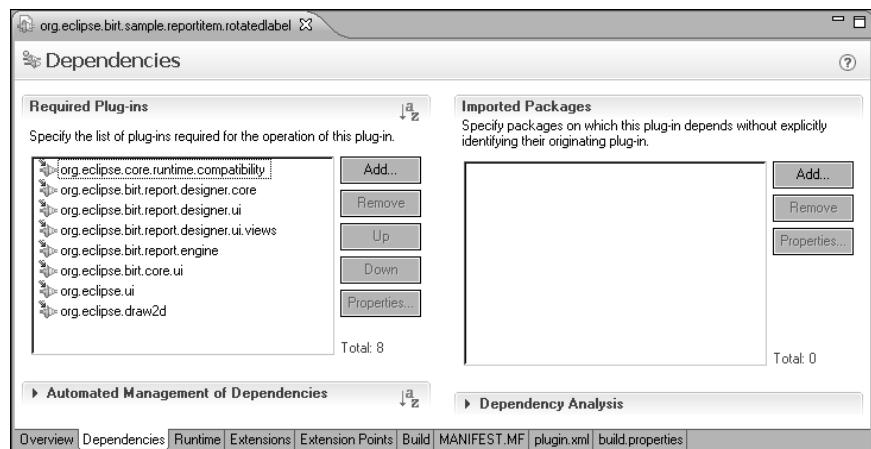


Figure 18-4 Dependencies showing required plug-ins

The order of the list determines the sequence in which a plug-in loads at run time. Use Up and Down to change the loading order as necessary, as shown in Figure 18-4.

You can optimize the dependency list using the Organize Manifests Wizard. To access the wizard in the PDE Manifest Editor, choose Overview. In Exporting, choose Organize the plug-in using the Organize Manifests Wizard. On the Organize Manifests Wizard, select the Remove unresolved packages option, as shown in Figure 18-5.

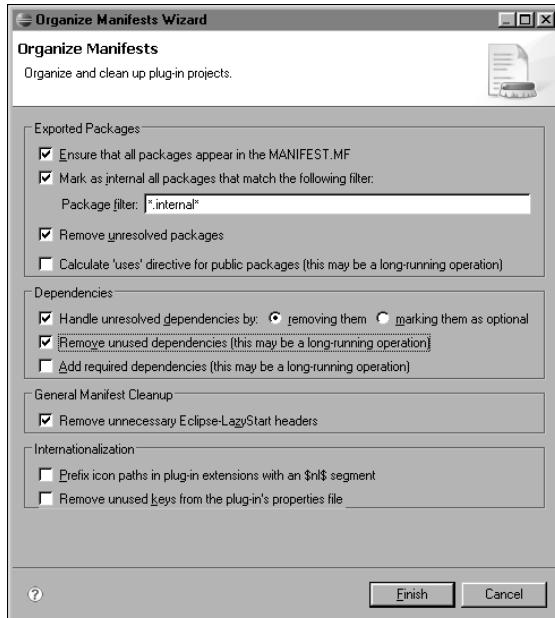


Figure 18-5 Organize Manifests Wizard

Specifying the run-time package for the rotated label report item extension

On Runtime, you specify exported packages, package visibility, the libraries, and folders on the plug-in classpath. In the rotated label report item plug-in, the only package that you must make visible to other plug-ins is org.eclipse.birt.sample.reportitem.rotatedlabel.

On PDE Manifest Editor, choose Runtime. On Runtime, in Exported Packages, verify that the org.eclipse.birt.sample.reportitem.rotatedlabel package appears in the list as shown in Figure 18-6.

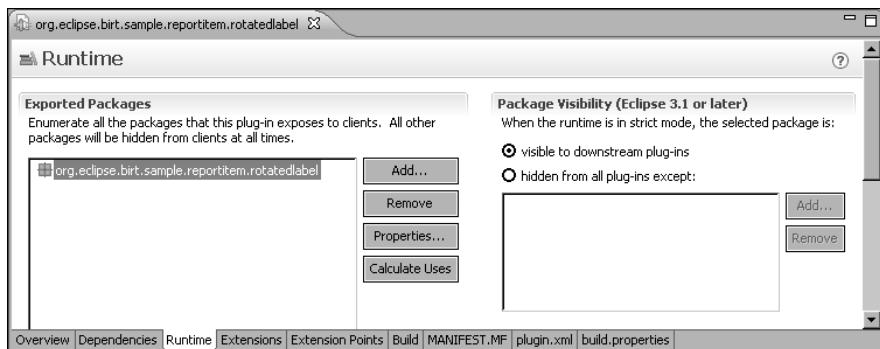


Figure 18-6 The Runtime page

Declaring the report item extension points

In this next step, you specify the extension points required to implement the rotated label report item extension and add the extension element details. The Eclipse PDE uses the XML schema defined for each extension point to provide the list of valid attributes and values specified for the extension elements.

The rotated label report item extension implements the following extension points:

- `org.eclipse.birt.report.designer.ui.reportitemUI`
Registers the graphical user interface (GUI) to use for the report item extension
- `org.eclipse.birt.report.model.reportItemModel`
Specifies how to represent and persist the report item extension in the ROM
- `org.eclipse.birt.report.engine.reportItemPresentation`
Specifies how to instantiate, process, and render a new report item at report presentation time
- `org.eclipse.birt.report.designer.ui.elementAdapters`
Specifies the adapters for the report item property page UI

The XML schema specifies the following properties that identify each extension point in the run-time environment:

- `ID`
Optional identifier of the extension instance
- `Name`
Optional name of the extension instance

The extension point, `org.eclipse.birt.report.designer.ui.reportitemUI`, specifies the following extension elements:

- `reportItemLabelUI`
Fully qualified name of the Java class that gets the display text for the report item component in BIRT Report Designer
- `model`
ROM report item extension name that maps to this UI component
- `palette`
Icon to show and the category in which the icon appears in the Palette

- editor

Flags indicating whether the editor shows in the MasterPage and Designer UI and is resizable in the Editor

- outline

Icon to show in the outline view

The extension point, `org.eclipse.birt.report.model.reportItemModel`, specifies the following extension element properties:

- extensionName

Internal unique name of the report item extension

- class

Fully qualified name of the Java class that implements the `org.eclipse.birt.report.model.api.extension.IReportItemFactory` interface

For `RotatedLabel`, `reportItem` specifies the following property extension elements:

- displayText

- rotationAngle

These extension elements specify the following properties:

- name

Internal unique name of the property extension element

- type

Data type, such as integer or string

- defaultValue

Default value of the property extension element

- defaultDisplayName

Display name to use if no localized I18N display name exists

The extension point, `org.eclipse.birt.report.engine.reportItemPresentation`, specifies the following `reportItem` extension elements:

- name

Unique name of the report item extension

- class

Fully qualified name of the Java class that implements the `org.eclipse.birt.report.engine.extension.IReportItemPresentation` interface

The extension point, org.eclipse.birt.report.designer.ui.elementAdapters, creates an adapter class that implements the IPageGenerator interface. The org.eclipse.birt.report.model.api.ExtendedItemHandle class implements the interfaces which make the extension point adaptable. The adapter specifies the following properties for the property pages:

- id
Unique name of ReportDesign.AttributeView
.RotatedLabelPageGenerator
- type
Implements org.eclipse.birt.report.designer.ui.views.IPageGenerator
- factory
Registers and creates the IPageGenerator instance
- singleton
Indicates whether one or more adapter elements can simultaneously exist
- priority
Indicates priority in run-time environment

The adapter also specifies and enables the following test properties:

- property
Name of property to test
- value
Expected value of the property to convert to a Java base type
- forcePluginActivation
Indicates whether to load the plug-in contributing the property tester

How to specify the extension points

- 1 On PDE Manifest Editor, choose Extensions.
- 2 In All Extensions, choose Add. New Extension appears.
- 3 On New Extension—Extension Points, in Available extension points, select the following plug-in:

org.eclipse.birt.report.designer.ui.reportitemUI

New Extension—Extension Points appears, as shown in Figure 18-7. Choose Finish.

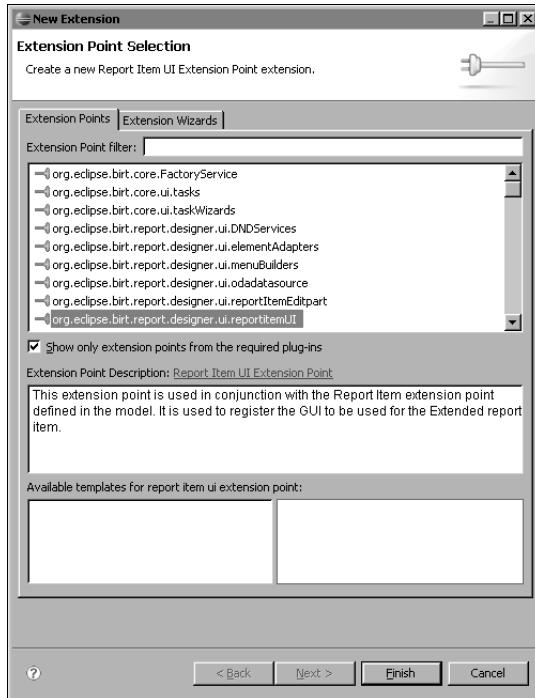


Figure 18-7 Extension points in New Extension

- 4 Repeat steps 2 and 3 to add the following extension points to the list of required extension points in the Extensions page:
 - org.eclipse.birt.report.model.reportItemModel
 - org.eclipse.birt.report.engine.reportitemPresentation
 - org.eclipse.birt.report.designer.ui.elementAdapters

Figure 18-8 shows the list of extension points required for the report item extension example.

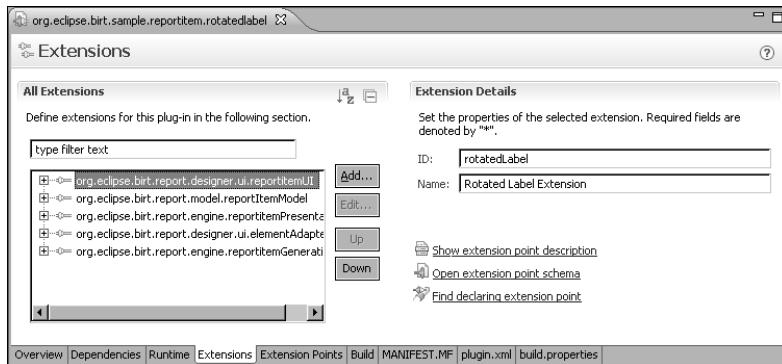


Figure 18-8 Required extension points for a report item extension

How to add the extension details

Perform the following tasks:

- 1 On Extensions, in All Extensions, select org.eclipse.birt.report.designer.ui.reportItemUI.
- 2 In Extension Details, set the following property values as shown in Table 18-3.

Table 18-3 Properties for reportItemUI extension

Property	Value
ID	rotatedLabel
Name	Rotated Label Extension

- 3 Repeat step 2 to add the property values shown in Table 18-3 to the extension details for the other extensions.

Creating the plug-in extension content

The XML schema specifies a grammar that you must follow when creating an extension in the Eclipse PDE. When you select an element of an extension in the Extensions page of the PDE, Eclipse uses the XML schema to populate the Extension Element Details section with the list of valid attributes and values for the element.

How to specify the plug-in extension content

- 1 In PDE Manifest Editor, choose Extensions.
- 2 In All Extensions, right-click org.eclipse.birt.report.designer.ui.reportItemUI and choose New→reportItemLabelUI, as shown in Figure 18-9.

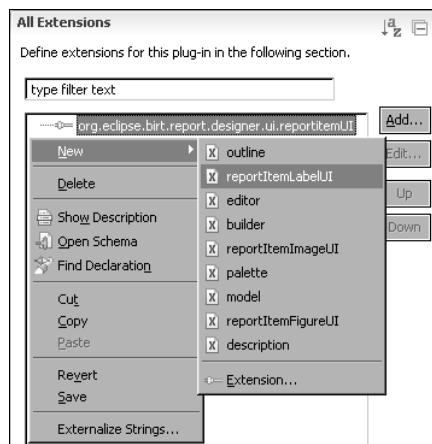


Figure 18-9 Selecting an extension element

All Extensions lists the extension, org.eclipse.birt.sample.reportitem.rotatedlabel.ReportItemLabelUI (rotatedItemLabelUI), and Extension Element Details lists rotatedItemLabelUI properties.

- 3 In Extension Element Details, type the following fully qualified class name:

```
org.eclipse.birt.sample.reportitem.rotatedlabel  
.RotatedLabelUI
```

Extensions appears, as shown in Figure 18-10.

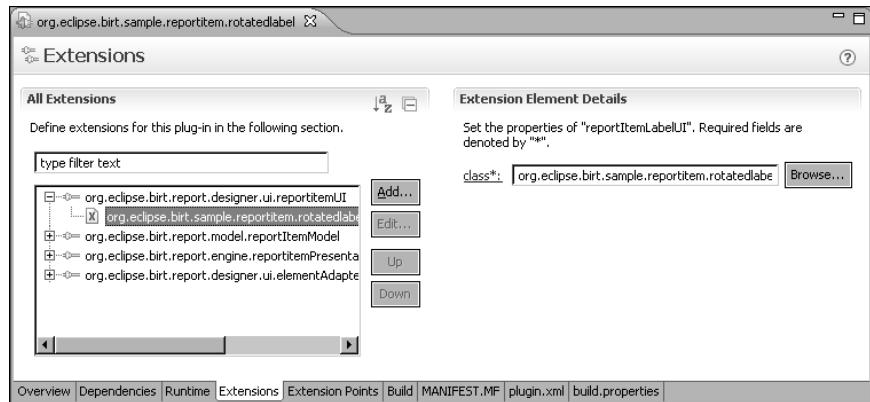


Figure 18-10 Properties for rotatedItemLabelUI

- 4 In All Extensions, right-click org.eclipse.birt.report.designer.ui.reportitemUI again and repeatedly choose New-><extension element> to add the following extension elements, corresponding extension element properties, and values, as shown in Table 18-4.

Table 18-4 Properties for other reportitemUI extension elements

Extension element	Property	Value
model	extensionName	RotatedLabel
palette	icon	icons/rotatedlabel.jpg
editor	showInMasterPage	true
	showInDesigner	true
outline	canResize	true
	icon	icons/rotatedlabel.jpg

- 5 In All Extensions, right-click org.eclipse.birt.report.model.reportItemModel and choose New->reportItem, as shown in Figure 18-11.

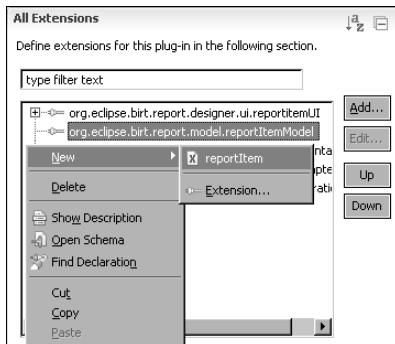


Figure 18-11 Choosing a new report item

- 6 In Extension Element Details, add the reportItem properties shown in Table 18-5.

Table 18-5 Property values for reportItem

Extension element	Property	Value
reportItem	extensionName	RotatedLabel
	class	org.eclipse.birt.sample.reportitem.rotatedlabel.RotatedLabelItemFactoryImpl

- 7 In All Extensions, in org.eclipse.birt.report.model.reportItemModel, perform the following tasks:

- 1 Right-click reportItem again and choose New->property to add the extension element properties shown in Table 18-6.

Table 18-6 Property values for displayText

Extension element	Property	Value
property	name	displayText
	type	string
	defaultValue	Rotated Label
	defaultDisplayName	Display Text

- 2 Right-click reportItem and choose New->property to add the extension element properties shown in Table 18-7.

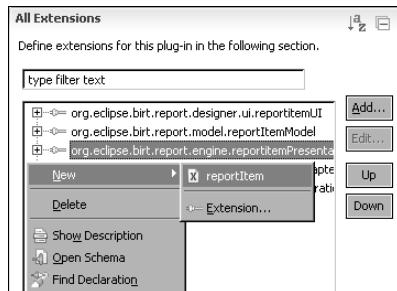
Table 18-7 Property values for rotationAngle

Extension element	Property	Value
property	name	rotationAngle
	type	string

Table 18-7 Property values for rotationAngle (*continued*)

Extension element	Property	Value
property (continued)	defaultValue	-45
	defaultDisplayName	Rotation Angle

- 8 In All Extensions, right-click org.eclipse.birt.report.engine.reportItemPresentation and choose New->reportItem, as shown in Figure 18-12.

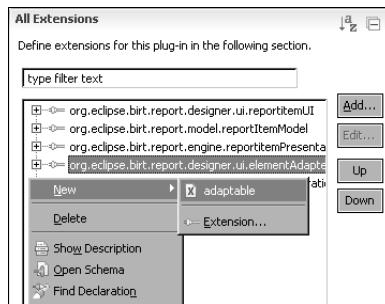
**Figure 18-12** Choosing reportItem

- 9 In Extension Element Details, add the reportItem property shown in Table 18-8.

Table 18-8 Property values for reportItem

Extension element	Property	Value
reportItem	name	RotatedLabel
	class	org.eclipse.birt.sample.reportitem.rotatedlabel.RotatedLabelPresentationImpl

- 10 In All Extensions, right-click org.eclipse.birt.report.designer.ui.elementAdapters and choose New->adaptable, as shown in Figure 18-13.

**Figure 18-13** Choosing adaptable

- 11** In Extension Element Details, add the adaptable property shown in Table 18-9.

Table 18-9 Property value for adaptable

Extension element	Property	Value
adaptable	class	org.eclipse.birt.report.model.api.ExtendedItemHandle

- 12** In All Extensions, right-click org.eclipse.birt.report.model.api.ExtendedItemHandle (adaptable) and choose New→adapter, as shown in Figure 18-14.

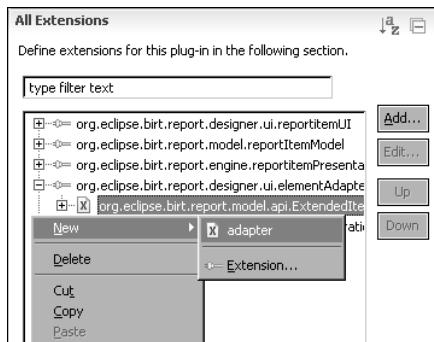


Figure 18-14 Choosing adapter

- 13** In Extension Element Details, add the adapter properties shown in Table 18-10.

Table 18-10 Property values for adapter

Extension element	Property	Value
adapter	id	ReportDesign.AttributeView.RotatedLabelPageGenerator
	type	org.eclipse.birt.report.designer.ui.views.IPageGenerator
	factory	org.eclipse.birt.sample.reportitem.rotatedlabel.views.RotatedLabelPageGeneratorFactory

Table 18-10 Property values for adapter (continued)

Extension element	Property	Value
adapter (continued)	singleton	false
	priority	1

- 14** In All Extensions, right-click ReportDesign.AttributeView.RotatedLabelPageGenerator (adapter) and choose New ➤ enablement, as shown in Figure 18-15.

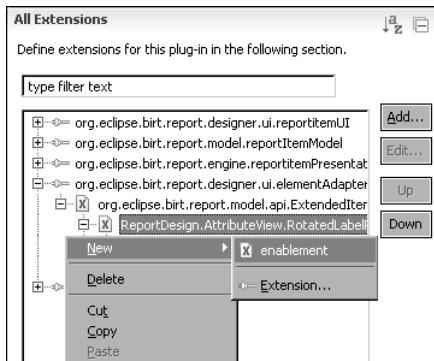


Figure 18-15 Choosing enablement

- 15** In All Extensions, right-click (enablement) and choose New ➤ test, as shown in Figure 18-16.

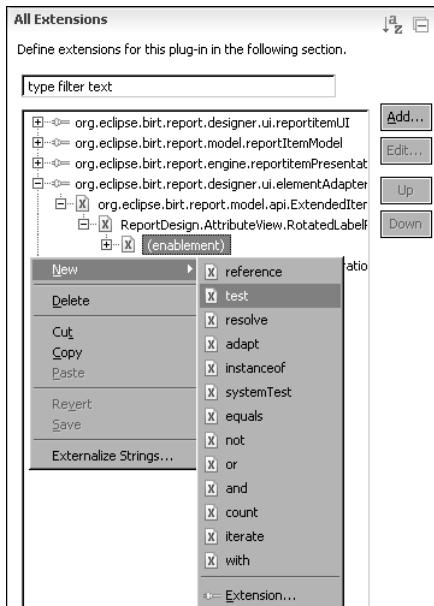


Figure 18-16 Choosing test

- 16** In Extension Element Details, add the ExtendItemHandle.extensionName (test) properties shown in Table 18-11.

Table 18-11 Property values for test

Extension element	Property	Value
test	property	ExtendItemHandle.extensionName
	value	RotatedLabel
	forcePluginActivation	true

Figure 18-17 shows the full list of extension points and elements required for the example, org.eclipse.birt.sample.reportitem.rotatedlabel.

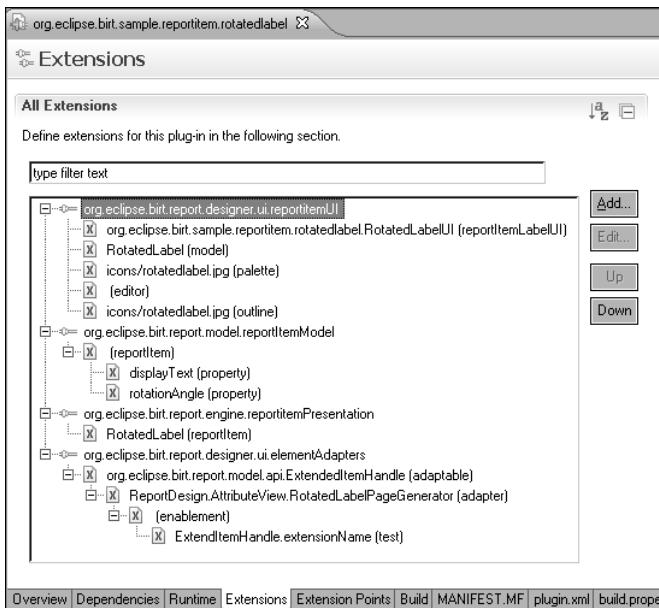


Figure 18-17 Extension points for rotated label report item extension

Understanding the rotated label report item extension

The rotated label report item plug-in provides the functionality required at run time to render the label of a report item as an image and rotate the image in the report design to display the label at the specified angle. The following sections provide a general description of the code-based extensions a developer must make to complete the development of the rotated report item extension after defining the plug-in framework in the Eclipse PDE.

The rotated label report item extension implements the following interfaces and classes:

- **org.eclipse.core.runtime.Plugin**
Defines the basic methods for starting, managing, and stopping the plug-in instance. RotatedLabelPlugin is the plug-in run-time class for the report item example.
- **org.eclipse.birt.report.designer.ui.extensions**
 - **IReportItemLabelProvider**
Defines the interface for the accessor method that provides the label text. RotatedLabelUI is the adapter class that extends org.eclipse.birt.report.designer.ui.extensions.ReportItemLabelProvider to implement this interface.
- **org.eclipse.birt.report.engine.extension**
 - **IRowSet**
Defines the interface to a row set. Provides metadata, grouping level, and row navigation methods.
 - **IReportItemPresentation**
Defines the interface for presentation of a report item extension. IReportItemPresentation sets the locale, resolution, output, and image formats, and processes the extended item in the report presentation environment. RotatedLabelPresentationImpl extends ReportItemPresentationBase, which is the adapter class that implements this interface.
- **org.eclipse.birt.report.model.api**
 - **DesignElementHandle**
Functions as the base class for all report elements. Derived classes provide specialized methods for each element type. RotatedLabelItemFactoryImpl uses this class to create a new rotated label report item when the user drags and drops the item from the Palette view to the report design.
 - **ExtendedItemHandle**
Provides a handle to an extended item that appears in a section of a report. The extended report item can have properties such as size, position, style, visibility rules, or a binding to a data source. RotatedLabelPresentationImpl uses this class when rendering the rotated label report item in the report.
- **org.eclipse.birt.report.model.api.extension**
 - **IMessages**

Defines the interface for getting a localized message from a message file using a resource key.

- **IReportItem**

Defines the interface for an instance of an extended report element. There is a one-to-one correspondence between the BIRT report item and this implementation. ReportItem is the adapter class that implements this interface.

- **IReportItemFactory**

Defines the interface for the factory that creates an instance of the extended element, IReportItem. IReportItem stores the model data and serializes the model state. ReportItemFactory is the adapter class that implements this interface.

RotatedLabelItemFactoryImpl uses these interfaces and classes to create a new rotated label report item in a report design.

- **org.eclipse.birt.report.model.elements.Style**

Extends org.eclipse.birt.report.model.core.StyleElement, the base class for report element styles, and implements org.eclipse.birt.report.model.elements.interfaces.IStyleModel, the interface for storing style element constants. GraphicsUtil uses these style elements to get the values of Font and FontSize properties for the report label.

- **org.eclipse.birt.report.designer.ui.views.attributes.providers.CategoryProviderFactory**

Provides the category information for Property Editor pages such as Borders, Margin, Section, Table of Contents, and Bookmark.

RotatedLabelCategoryProviderFactory extends CategoryProviderFactory to provide the categories of the Property Editor pages for the rotated label report item.

- **org.eclipse.birt.report.designer.internal.ui.views.attributes.page.AttributePage**

Creates the general page content. RotatedLabelGeneralPage extends AttributePage to build a customized UI for the General page in the Property Editor, adding the following properties:

- Display text
- Rotation Angle
- Font
- Size

The following sections contain implementation details for the most important classes in the rotated label report item extension.

Understanding RotatedLabelItemFactoryImpl

The RotatedLabelItemFactoryImpl class instantiates a new report item when the user drags a rotated label report item from Palette and drops the report item in the BIRT Report Designer Editor. RotatedLabelItemFactoryImpl extends the adapter class, org.eclipse.birt.report.model.api.extension.ReportItemFactory.

The newReportItem() method receives a reference to DesignElementHandle, which provides the interface to the BIRT report model. The newReportItem() method instantiates the new report item, as shown in Listing 18-1.

Listing 18-1 The newReportItem() method

```
public class RotatedTextItemFactoryImpl  
    extends ReportItemFactory implements IMessages  
{  
    public IReportItem newReportItem( DesignElementHandle deh )  
    {  
        return new ReportItem( );  
        ...  
    }  
}
```

Understanding RotatedLabelUI

In the RotatedLabelUI class, the RotatedLabelUI.getLabel() method provides the text representation for the label to BIRT Report Designer. RotatedLabelUI extends the adapter class, org.eclipse.birt.report.designer.ui.extensions.ReportItemLabelProvider. Listing 18-2 shows the code for the getLabel() method.

Listing 18-2 The getLabel() method

```
public class RotatedLabelUI extends ReportItemLabelProvider  
{  
    public String getLabel( ExtendedItemHandle handle )  
    {  
        if ( handle.getProperty( "displayText" ) != null ) {  
            return ( String ) handle.getProperty( "displayText" );  
        } else {  
            return "Rotated Label";  
        }  
    }  
}
```

Understanding RotatedLabelPresentationImpl

The RotatedLabelPresentationImpl class specifies how to process and render the report item at presentation time. RotatedLabelPresentationImpl extends the org.eclipse.birt.report.engine.extension.ReportItemPresentationBase class.

The method, `onRowSets()`, renders the rotated label report item as an image, rotated by the angle specified in the report design, as shown in Listing 18-3.

Listing 18-3 The `onRowSets()` method

```
public Object onRowSets(IRowSet[ ] rowSets) throws
    BirtException
{
    if ( modelHandle == null )
    {
        return null;
    }
    graphicsUtil = new GraphicsUtil( );
    org.eclipse.swt.graphics.Image rotatedImage =
        graphicsUtil.createRotatedText( modelHandle );
    ImageLoader imageLoader = new ImageLoader( );
    imageLoader.data = new ImageData[ ]
    {
        rotatedImage.getImageData();
    }
    ByteArrayOutputStream baos =
        new ByteArrayOutputStream();
    imageLoader.save( baos, SWT.IMAGE_JPEG );
    return baos.toByteArray();
}
```

Understanding GraphicsUtil

The `GraphicsUtil` class creates the image containing the specified text and rotates the text image to the specified angle, using the following methods:

- `createRotatedText()`

This method performs the following operations:

- Gets the display text and rotation angle properties
- Sets the display text font and determines the font metrics
- Creates an image the same size as the display text String
- Draws the display text as an image
- Calls the `rotateImage()` method to rotate the image at the specified angle
- Disposes of the operating system resources used to render the image
- Returns the image object

Listing 18-4 shows the code for `createRotatedText()` method.

Listing 18-4 The createRotatedText() method

```
public Image createRotatedText( ExtendedItemHandle
    modelHandle )
{
    Image stringImage;
    Image image;
    GC gc;
    String text = "";
    if ( modelHandle.getProperty( "displayText" ) != null ) {
        text = ( String ) modelHandle.getProperty(
            "displayText" );
    }
    Integer angle = -45;
    if ( modelHandle.getProperty( "rotationAngle" ) != null ) {
        try {
            angle = Integer.parseInt( (String)
                modelHandle.getProperty( "rotationAngle" ) );
        }
        catch( NumberFormatException e ) {
            angle = -45;
        }
    }
    String fontFamily = "Arial";
    if ( modelHandle.getProperty(Style.FONT_FAMILY_PROP ) !=
        null ) {
        fontFamily = ( String ) modelHandle.getProperty(
            Style.FONT_FAMILY_PROP );
    }
    StyleHandle labelStyle = modelHandle.getPrivateStyle();
    DimensionHandle fontSize =
        (DimensionHandle) labelStyle.getFontSize();

    int height = 12;
    String units = fontSize.getUnits();
    if ( units.compareToIgnoreCase("pt") == 0 )
    {
        height = (int) fontSize.getMeasure();
    }
    if ( display == null ) SWT.error
        ( SWT.ERROR_THREAD_INVALID_ACCESS );

    FontData fontData = new FontData( fontFamily, height, 0 );
    Font font = new Font( display, fontData );
    try
    {
        gc = new GC( display );
```

```

        gc.setFont( font );
        gc.getFontMetrics();
        Point pt = gc.textExtent( text );
        gc.dispose();
        stringImage = new Image( display, pt.x, pt.y );
        gc = new GC( stringImage );
        gc.setFont( font );
        gc.drawText( text, 0, 0 );
        image = rotateImage( stringImage, angle.doubleValue() );
        gc.dispose();
        stringImage.dispose();
        return image;
    }
    catch( Exception e )
    {
        e.printStackTrace();
    }
    return null;
}

```

■ **rotateImage()**

This method rotates the image and determines the width, height, and point of origin for the image, as shown in Listing 18-5.

Listing 18-5 The rotateImage() method

```

private Image rotateImage ( Image img, double degrees )
{
    double positiveDegrees = ( degrees % 360 ) +
        ( ( degrees < 0 ) ? 360 : 0 );
    double degreesMod90 = positiveDegrees % 90;
    double radians = Math.toRadians( positiveDegrees );
    double radiansMod90 = Math.toRadians( degreesMod90 );
    if ( positiveDegrees == 0 )
        return img;
    int quadrant = 0;
    if ( positiveDegrees < 90 )
        quadrant = 1;
    else if ( ( positiveDegrees >= 90 ) &&
        ( positiveDegrees < 180 ) )
        quadrant = 2;
    else if ( ( positiveDegrees >= 180 ) &&
        ( positiveDegrees < 270 ) )
        quadrant = 3;
    else if ( positiveDegrees >= 270 )
        quadrant = 4;
    int height = img.getBounds().height;
    int width = img.getBounds().width;

```

```

        double side1 = ( Math.sin( radiansMod90 ) * height ) +
                      ( Math.cos( radiansMod90 ) * width );
        double side2 = ( Math.cos( radiansMod90 ) * height ) +
                      ( Math.sin( radiansMod90 ) * width );
        double h = 0;
        int newWidth = 0, newHeight = 0;
        if ( ( quadrant == 1 ) || ( quadrant == 3 ) ) {
            h = ( Math.sin( radiansMod90 ) * height );
            newWidth = ( int )side1;
            newHeight = ( int )side2;
        } else {
            h = ( Math.sin( radiansMod90 ) * width );
            newWidth = ( int )side2;
            newHeight = ( int )side1;
        }
        int shiftX = ( int )( Math.cos( radians ) * h ) -
                     ( ( quadrant == 3 ) || ( quadrant == 4 )
                     ? width : 0 );
        int shiftY = ( int )( Math.sin( radians ) * h ) +
                     ( ( quadrant == 2 ) || ( quadrant == 3 )
                     ? height : 0 );
        Image newImg = new Image( display, newWidth, newHeight );
        GC newGC = new GC( newImg );
        Transform tr = new Transform( display );
        tr.rotate( ( float )positiveDegrees );
        newGC.setTransform( tr );
        newGC.setBackground( display.getSystemColor(
            SWT.COLOR_WHITE ) );
        newGC.drawImage( img, shiftX, -shiftY );
        newGC.dispose( );
        return newImg;
    }
}

```

Understanding RotatedLabelCategoryProvider Factory

In the class, the getCategoryProvider() method provides the category information for Property Editor pages such as Borders, Margin, Section, Table of Contents, and Bookmark for the rotated label report item. Listing 18-6 shows the code for the getCategoryProvider() method, which specifies the category properties and associated classes.

Listing 18-6 The getCategoryProvider() method

```

public ICategoryProvider getCategoryProvider( Object input )
{
    CategoryProvider provider = new CategoryProvider(
        new String[ ]{

```

```

CategoryProviderFactory.CATEGORY_KEY_GENERAL,
CategoryProviderFactory.CATEGORY_KEY_BORDERS,
CategoryProviderFactory.CATEGORY_KEY_MARGIN,
CategoryProviderFactory.CATEGORY_KEY_ALTTEXT,
CategoryProviderFactory.CATEGORY_KEY_SECTION,
CategoryProviderFactory.CATEGORY_KEY_VISIBILITY,
CategoryProviderFactory.CATEGORY_KEY_TOC,
CategoryProviderFactory.CATEGORY_KEY_BOOKMARK,
CategoryProviderFactory.CATEGORY_KEY_USERPROPERTIES,
CategoryProviderFactory
    .CATEGORY_KEY_NAMEDEXPRESSIONS,
CategoryProviderFactory
    .CATEGORY_KEY_ADVANCEPROPERTY,
}, new String[ ]{
    "DataPageGenerator.List.General",
    "DataPageGenerator.List.Borders",
    "DataPageGenerator.List.Margin",
    "ImagePageGenerator.List.AltText",
    "DataPageGenerator.List.Section",
    "DataPageGenerator.List.Visibility",
    "DataPageGenerator.List.TOC",
    "DataPageGenerator.List.Bookmark",
    "ReportPageGenerator.List.UserProperties",
    "ReportPageGenerator.List.NamedExpressions",
    "ReportPageGenerator.List.AdvancedProperty",
}, new Class[ ]{
    RotatedLabelGeneralPage.class,
    BordersPage.class,
    ItemMarginPage.class,
    AlterPage.class,
    SectionPage.class,
    VisibilityPage.class,
    TOCExpressionPage.class,
    BookMarkExpressionPage.class,
    UserPropertiesPage.class,
    NamedExpressionsPage.class,
    AdvancePropertyPage.class,
} );
return provider;
}

```

Understanding RotatedLabelGeneralPage

In the class, the buildUI() method creates the content for the General page in the Property Editor, adding the following display text controls:

- Rotation Angle
- Font
- Size

Listing 18-7 shows the code for the buildUI() method.

Listing 18-7 The buildUI() method

```
public void buildUI( Composite parent )
{
    super.buildUI( parent );
    container.setLayout( WidgetUtil.createGridLayout( 5, 15 ) );

    LibraryDescriptorProvider provider = new
    LibraryDescriptorProvider();
    librarySection = new TextSection(
    provider.getDisplayName(),
        container,
        true );
    librarySection.setProvider( provider );
    librarySection.setGridPlaceholder( 2, true );
    addSection( RotatedLabelPageSectionID.LIBRARY,
    librarySection );

    // display text property
    separatorSection = new SeperatorSection(
        container, SWT.HORIZONTAL );
    addSection( RotatedLabelPageSectionID.SEPARATOR,
    separatorSection );
    TextPropertyDescriptorProvider nameProvider =
        new TextPropertyDescriptorProvider( "displayText",
            ReportDesignConstants.EXTENDED_ITEM );
    TextSection nameSection = new TextSection(
        "Display text:",
        container,
        true );
    nameSection.setProvider( nameProvider );
    nameSection.setGridPlaceholder( 3, true );
    nameSection.setWidth(200 );
    addSection( RotatedLabelPageSectionID.DISPLAY_TEXT,
    nameSection );

    // rotation angle property
    TextPropertyDescriptorProvider angleProvider =
        new TextPropertyDescriptorProvider( "rotationAngle",
            ReportDesignConstants.EXTENDED_ITEM );
    TextSection angleSection =
        new TextSection( "Rotation Angle:",
            container,
            true );
    angleSection.setProvider( angleProvider );
    angleSection.setGridPlaceholder( 3, true );
    angleSection.setWidth( 200 );
```

```

        addSection( RotatedLabelPageSectionID.ROTATION_ANGLE,
                    angleSection );

    //font family property
    ComboPropertyDescriptorProvider fontFamilyProvider =
        new ComboPropertyDescriptorProvider(
            StyleHandle.FONT_FAMILY_PROP,
            ReportDesignConstants.STYLE_ELEMENT );
    ComboSection fontFamilySection =
        new ComboSection( fontFamilyProvider.getDisplayName( ),
                         container,
                         true );
    fontFamilySection.setProvider( fontFamilyProvider );
    fontFamilySection.setLayoutNum( 2 );
    fontFamilySection.setWidth( 200 );
    addSection( PageSectionId.LABEL_FONT_FAMILY,
                fontFamilySection );

    //font size property
    FontSizePropertyDescriptorProvider fontSizeProvider =
        new FontSizePropertyDescriptorProvider(
            StyleHandle.FONT_SIZE_PROP,
            ReportDesignConstants.STYLE_ELEMENT );
    FontSizeSection fontSizeSection = new FontSizeSection(
        fontSizeProvider.getDisplayName( ),
        container,
        true );
    fontSizeSection.setProvider( fontSizeProvider );
    fontSizeSection.setLayoutNum( 4 );
    fontSizeSection.setGridPlaceholder( 2, true );
    fontSizeSection.setWidth( 200 );
    addSection( PageSectionId.LABEL_FONT_SIZE,
                fontSizeSection );
    createSections( );
    layoutSections( );
}

```

Deploying and testing the rotated label report item plug-in

After building the plug-in, the Eclipse PDE provides support for deploying and testing the plug-in in a run-time environment. The following sections describe the steps to deploy and test the rotated label report item plug-in example.

Deploying a report item extension

To deploy the rotated label report item plug-in and integrate the extension with the BIRT Report Designer, use the Export wizard or manually copy the org.eclipse.birt.sample.reportitem.rotatedtext plug-in from your workspace to the eclipse\plugins directory. For testing, you can deploy the report item extension to the BIRT run-time directory.

Launching the rotated label report item plug-in

On PDE Manifest Editor, in Overview, the Testing section contains links to launch a plug-in as a separate Eclipse application in either Run or Debug mode. Figure 18-18 shows Overview for the rotated label report item extension example in the host instance of the PDE Workbench.

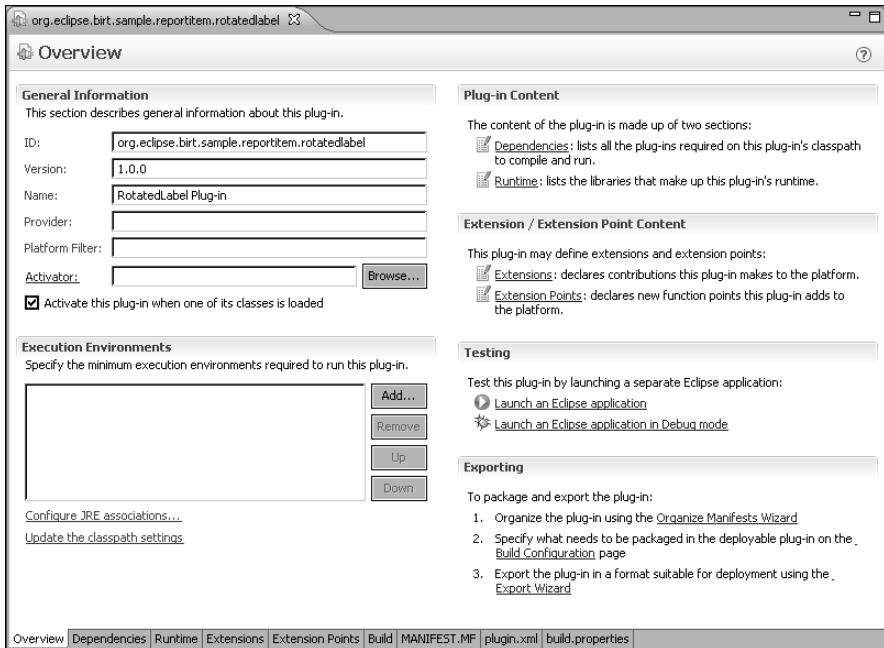


Figure 18-18 Overview information for the rotated label report item extension

How to launch a run-time workbench

- 1 On PDE Manifest Editor, choose Overview. In Testing, choose Launch an Eclipse application. Eclipse launches the run-time workbench.
- 2 In Report Design, choose File→New→Project. New Project appears. In Wizards, choose Report Project, as shown in Figure 18-19.

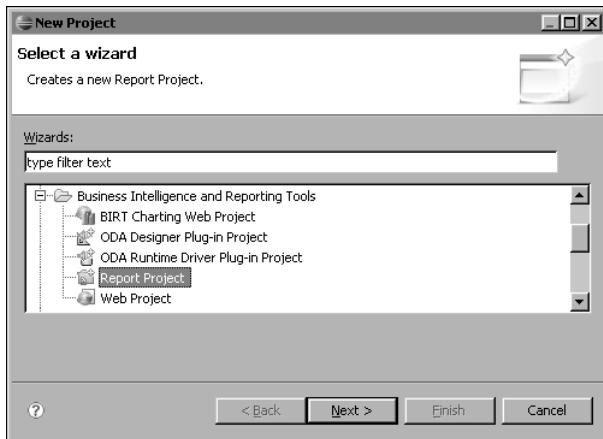


Figure 18-19 Selecting Report Project in New Project

Choose Next. New Report Project appears.

- 3** In Project name, type:

`testRotatedLabel`

- 4** To specify the run-time location, deselect Use default location. In Location, type a location:

`C:\runtime-RotatedLabel\testRotatedLabel`

Choose Finish. Sample report item appears in the Navigator.

- 5** In Report Design—Eclipse Platform, choose File→New→Report. New Report appears, as shown in Figure 18-20.

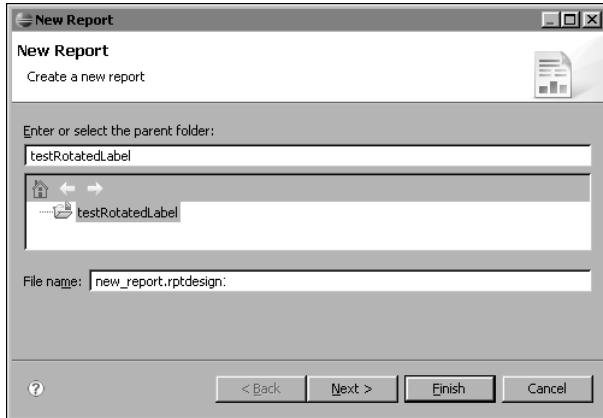


Figure 18-20 New Report

- 6** In File name, type a file name if you want to change the default file name. Choose Next. New Report displays the report templates.

- 7 In Report templates, choose Blank Report. Choose Finish. The layout editor displays the report design, new_report.rptdesign. Palette contains the RotatedText report item.
- 8 From Palette, drag RotatedLabel to Layout, as shown in Figure 18-21.

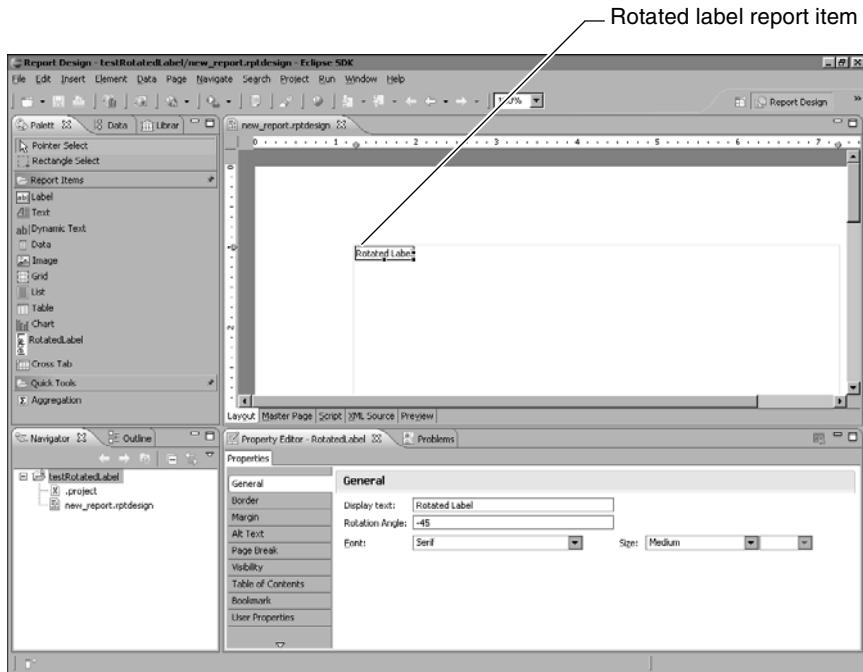


Figure 18-21 Rotated label report item in the report design

- 9 In new_report.rptdesign, choose Preview. The preview appears, displaying the rotated label report item, as shown in Figure 18-22.

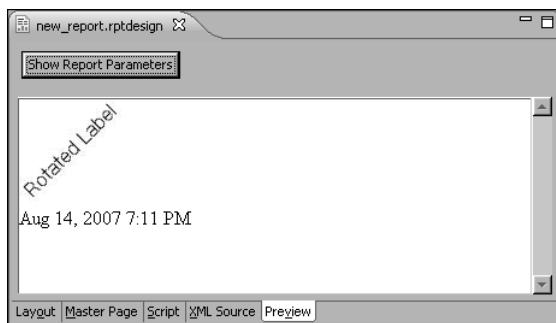


Figure 18-22 The rotated label in the report preview

This page intentionally left blank

19

Developing a Report Rendering Extension

This chapter describes how to develop a report rendering extension using the Eclipse PDE with sample CSV and XML report rendering extensions as the examples. You learn how to develop a BIRT report rendering extension in the following sections:

- Understanding a report rendering extension
- Developing a CSV report rendering extension
- Developing an XML report rendering extension

Understanding a report rendering extension

BIRT Report Engine provides report rendering extensions that render a report in HTML, PDF, XLS, PostScript, and Microsoft Word and PowerPoint. In BIRT release 2.2.1, the BIRT report rendering API supports rendering a report in a customized format, such as CSV or XML.

This chapter provides sample implementations of customized CSV and XML report rendering extensions, `org.eclipse.birt.report.engine.emitter.csv` and `org.eclipse.birt.report.engine.emitter.xml`. The sample code creates plug-ins that write the data contents of a report to a file in the specified format.

A BIRT plug-in typically loads and runs in the BIRT Report Engine environment rather than the Eclipse run-time environment. BIRT implements a separate plug-in loading framework for the BIRT Report Engine, giving the BIRT Report Engine complete control of report execution. A BIRT engine plug-in extension is functionally similar to an Eclipse plug-in extension.

A rendering extension adds an emitter to the BIRT Report Engine framework by implementing the extension point, org.eclipse.birt.report.engine.emitters. The XML schema file, org.eclipse.birt.report.engine/schema/emitters.exsd, describes this extension point.

The extension point enables support for a new output format in the presentation engine. The BIRT plug-in registry uses this extension point to discover all supported output formats specified for the report engine environment.

This book uses the customized CSV and XML report rendering extensions in this chapter as examples of how to create report rendering extension. The BIRT release 2.2.1 user interface also provides a built-in data extraction feature that can export data from a report document in CSV, tab-separated values (TSV), and Extensible Markup Language (XML) formats.

You can download the source code for the CSV and XML report rendering extension examples at <http://www.actuate.com/birt/contributions>. For reference documentation, see the BIRT Report Engine API Javadoc in Eclipse Help for the org.eclipse.birt.report.engine.emitter and org.eclipse.birt.report.engine.content packages.

Developing a CSV report rendering extension

The CSV report rendering extension extends the functionality defined by the org.eclipse.birt.report.engine.emitter package, which is part of the org.eclipse.birt.report.engine plug-in. In developing the CSV report rendering extension, you perform the following tasks:

- Create a CSV report rendering extension project in the Eclipse PDE.
- Define the dependencies.
- Declare the emitters extension point.
- Implement the emitter interfaces.
- Test the extension in the run-time environment.

Creating a CSV report rendering plug-in project

Create a new plug-in project for the CSV report rendering extension using the Eclipse PDE.

How to create the CSV report rendering plug-in project

- 1 From the Eclipse PDE menu, choose File->New->Project. New Project appears.
- 2 On New Project, select Plug-in Project. Choose Next. New Plug-in Project appears.

- 3** In Plug-in Project, modify the settings, as shown in Table 19-1.

Table 19-1 Values for Plug-in Project fields

Section	Option	Value
Plug-in Project	Project name	org.eclipse.birt.report.engine.emitter.csv
	Use default location	Selected
	Location	Not available when you select Use default location
Project Settings	Create a Java project	Selected
	Source folder	src
	Output folder	bin
Target Platform	Eclipse version	3.3
	an OSGi framework	Not selected

Plug-in Project appears as shown in Figure 19-1. Choose Next. Plug-in Content appears.

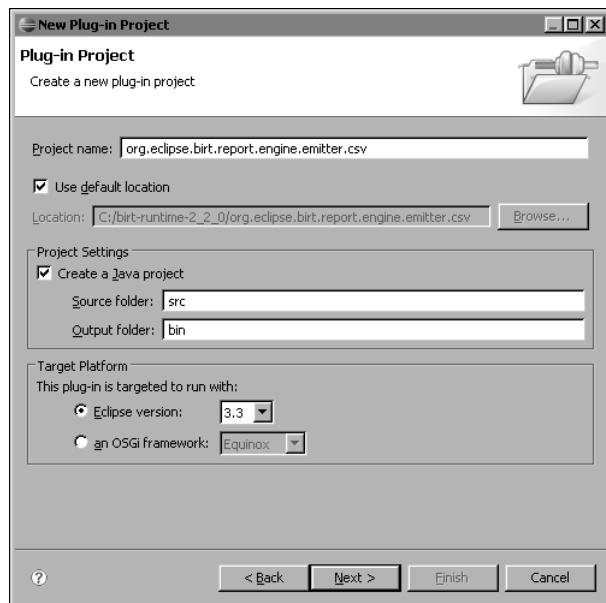


Figure 19-1 Values for Plug-in Project

- 4** In Plug-in Content, modify the settings, as shown in Table 19-2.

Table 19-2 Values for Plug-in Content fields

Section	Option	Value
Plug-in Properties	Plug-in ID	org.eclipse.birt.report.engine.emitter.csv
	Plug-in Version	1.0.0
	Plug-in Name	BIRT CSV Emitter
	Plug-in Provider	yourCompany.com or leave blank
	Classpath	csvEmitter.jar or leave blank
Plug-in Options	Generate an activator, a Java class that controls the plug-in's life cycle	Selected
	Activator	org.eclipse.birt.report.engine.emitter.csv.CsvPlugin
	This plug-in will make contributions to the UI	Not selected
Rich Client Application	Would you like to create a rich client application?	No

Plug-in Content appears as shown in Figure 19-2. Choose Finish.

**Figure 19-2** Values for Plug-in Content

The CSV report rendering extension project appears in the Eclipse PDE workbench, as shown in Figure 19-3.

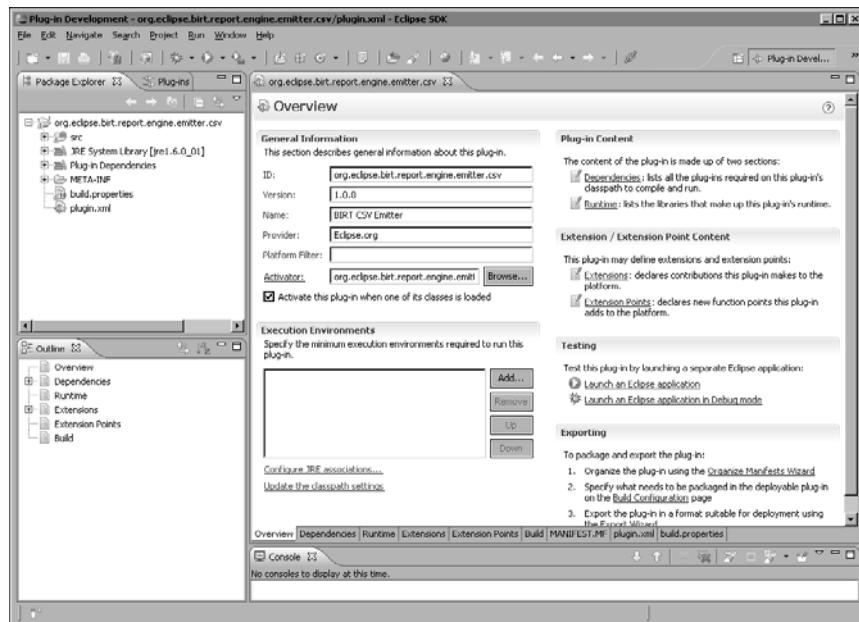


Figure 19-3 CSV report rendering extension project

Defining the dependencies for the CSV report rendering extension

To compile and run the CSV report rendering example, you need to specify the list of plug-ins that must be available on the classpath of the extension.

How to specify the dependencies

- 1 On PDE Manifest Editor, choose Overview.
- 2 In Plug-in Content, choose Dependencies. Required Plug-ins contains the following plug-in:
`org.eclipse.core.runtime`
- 3 In Required Plug-ins, perform the following tasks:
 - 1 Select `org.eclipse.core.runtime` and choose Remove.
`org.eclipse.core.runtime` no longer appears in Required Plug-ins.
 - 2 Choose Add. Plug-in Selection appears.
 - 3 In Plug-in Selection, hold down CTRL and select the `org.eclipse.birt.report.engine` plug-in, `org.eclipse.birt.report.engine`.

Choose OK. Dependencies appears as shown in Figure 19-4.

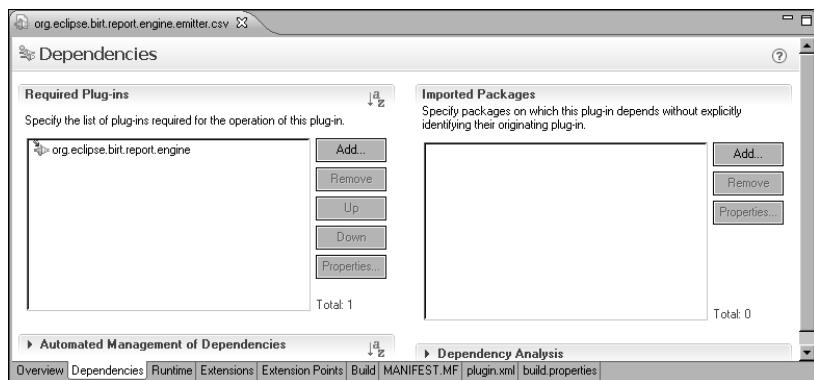


Figure 19-4 The Dependencies page

Declaring the emitters extension point

In this step, you specify the extension point required to implement the CSV report rendering extension and add the extension element details. The extension point, `org.eclipse.birt.report.engine.emitters`, specifies the following properties that identify the extension point:

- ID
Optional identifier of the extension instance
- Name
Optional name of the extension instance

The extension point defines an emitter that specifies the output format for the plug-in, requiring you to define the following extension element properties:

- class
Java class that implements the `IContentEmitter` interface
- format
Output format that the emitter supports, such as `csv`
- mimeType
MIME type for the supported output format, such as `text/csv`
- id
Optional identifier of the emitter extension

You specify the extension point and extension element details using the Eclipse PDE.

How to specify the extension point

- 1 On PDE Manifest Editor, choose Extensions.
- 2 In All Extensions, choose Add. New Extension—Extension Point Selection appears.
- 3 In Available extension points, select the following plug-in:
`org.eclipse.birt.report.engine.emitters`

Choose Finish. Extensions appears, as shown in Figure 19-5.

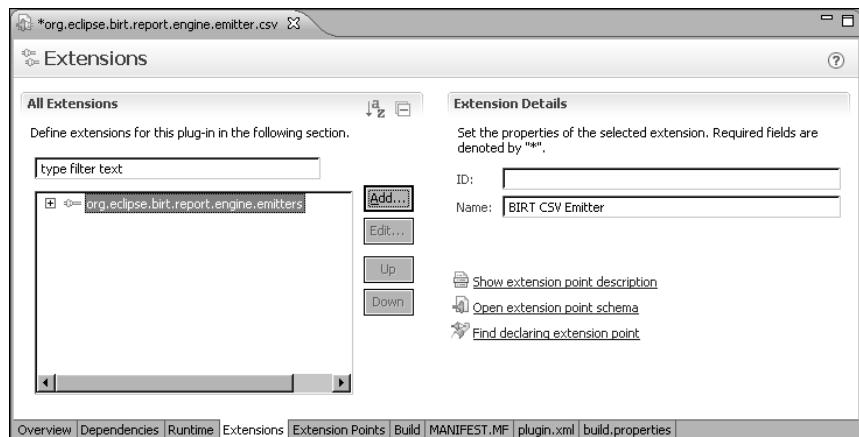


Figure 19-5 Emitter plug-in extension on the Extensions page

All Extensions lists the extension point, `org.eclipse.birt.report.engine.emitters`. Extension Details contains the list of extension details specified in the XML schema file, `emitters.exsd`.

- 4 In All Extensions, right-click the extension point, `org.eclipse.birt.report.engine.emitters`, and choose the extension element, `emitter`, as shown in Figure 19-6.

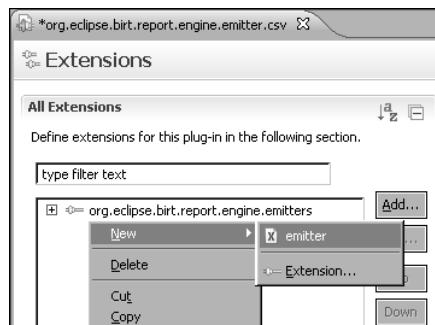


Figure 19-6 Selecting the emitter extension element

Extension element, `emitter`, appears in All Extensions.

- 5** In Extension Element Details, specify the properties for the emitter extension element, emitter, as shown in Table 19-3.

Table 19-3 Property values for the emitter extension element

Property	Value
class	org.eclipse.birt.report.engine.emitter.csv.CSVReportEmitter
format	csv
mimeMimeType	text/csv
id	org.eclipse.birt.report.engine.emitter.csv

Extensions appears as shown in Figure 19-7. PDE Manifest Editor automatically updates plugin.xml.

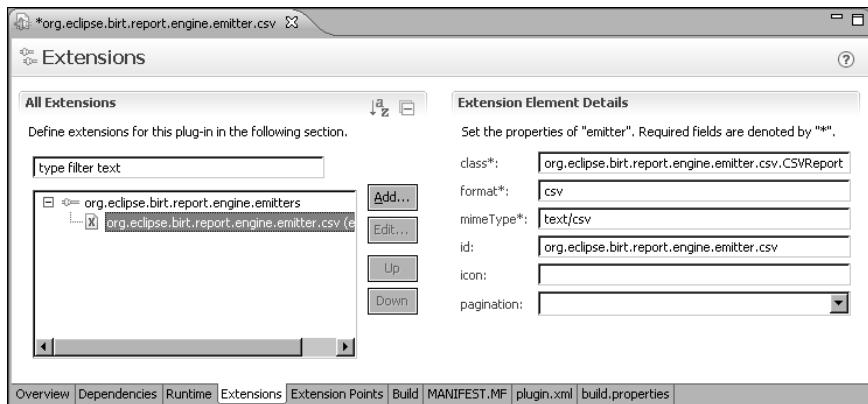


Figure 19-7 Property values for the emitter extension

Understanding the sample CSV report rendering extension

The CSV report rendering extension described in this chapter is a simplified example that illustrates how to create a report rendering plug-in using the Eclipse PDE. The extension extends the report emitter interfaces in `org.eclipse.birt.report.engine.emitter`.

The CSV report rendering extension example exports only the data in the table controls to the CSV output file. The lines of the CSV output file contain only column data separated by commas. The sample CSV report emitter does not export images, charts, or hyperlinks.

The extension example creates the CSV output file in the same folder as the exported report. The output file name is the name of the report with a .csv extension. The extension does not support nested tables.

The following section provides a general description of the code-based extensions a developer must make to complete the development of the CSV report rendering extension after defining the plug-in framework in the Eclipse PDE.

Implementing the emitter interfaces

The org.eclipse.birt.report.engine.emitter plug-in defines the report emitter interfaces used to render the report item elements in a report container, such as a page, table, row, column, cell, label, image, or extended item. The CSV report rendering extension implements parts of the following interfaces and classes in the emitter plug-in:

- `org.eclipse.core.runtime.Plugin`
Defines the basic methods for starting, managing, and stopping the plug-in instance.
- `org.eclipse.birt.report.engine.api.IRenderOption`
Defines the interface that provides the emitter with configuration information. RenderOption is the adapter class that implements IRenderOption.
- `org.eclipse.birt.report.engine.emitter`
 - `IContentEmitter`
Defines the interface for the start and end processing that renders the report items.
 - `IEmitterServices`
Defines the interface the emitter uses to access the following items:
 - ❑ `Emitter configuration`
Provides information on the engine emitter configuration
 - ❑ `Rendering context`
Provides information on the engine rendering context
 - ❑ `Rendering options`
Provides the output configuration information
 - ❑ `Runnable report design`
Defines the methods that get the report design handle, images, and property values, such as report name, title, and description
 - ❑ `Engine task`
Defines the set of operations that renders a report document to the specified output format

Implementing the content interfaces

The org.eclipse.birt.report.engine.content plug-in defines the interfaces for BIRT report items that BIRT Report Engine uses to pass content to an emitter. These content interfaces provide a common protocol for rendering an instance of a content object.

The CSV report rendering extension implements some of these interfaces. Each interface defines accessor methods for properties depending on the type of the content object, as shown in Table 19-4.

Table 19-4 Interfaces that pass content to an emitter

Interface	Properties
IBandContent	Header and footer content in a table or group.
ICellContent	Row and column spans.
IContainerContent	No defined fields or methods. Inherits fields and methods from the superinterfaces, IContent, IEElement, and CSSStylableElement.
IDataContent	Label and help keys, text, and values.
IForeignContent	Raw types and values not handled by BIRT Report Engine.
IIImageContent	URI, MIME type, image source, image map, help key, extension, alternative key and text.
ILabelContent	Label and help keys and text.
IPageContent	Page number, dimensions, orientation, and content style.
IRowContent	Table, group, band, and row.
ITableContent	Table band content, caption, column, and column count.
ITextContent	Text.

These interfaces in the org.eclipse.birt.report.engine.content package inherit from the superinterface, org.eclipse.birt.report.engine.content.IContent. IContent specifies methods that provide access to the following additional interfaces and properties in the package:

- **IContentVisitor**
Defines a visitor interface, typically used by a buffered emitter. The visitor design pattern separates content objects and their operations into different classes. Implementing a visitor design pattern allows a developer to change the operations performed on a collection of objects without changing the structure of the objects and recompiling the object code.
- **ContentType**
Lists the constant field values used to identify content types.

- **IHyperlinkAction**
Defines the interface that allows BIRT Report Engine to pass hyperlink information to an emitter.
- The following interfaces define additional functionality in the org.eclipse.birt.report.engine.content package:
 - **IStyle**
Defines the accessor methods for ROM style properties
 - **IReportContent**
Creates report item content, using the following components:
 - Report design
An instance of org.eclipse.birt.report.engine.ir.Report
 - Table of Contents (TOC) tree
An instance of org.eclipse.birt.report.engine.api.TOCTree
 - CSS engine
An instance of org.eclipse.birt.report.engine.css.engine.CSSEngine

Understanding the CSV report rendering extension package

The implementation package for the CSV report rendering extension example, org.eclipse.birt.report.engine.emitter.csv, contains the following classes:

- **CSVPlugin**
Defines the methods for starting, managing, and stopping a plug-in instance.
- **CSVRenderOption**
Integrates the plug-in with BIRT Report Engine, specifying configuration information. CSVRenderOption extends RenderOption, specifying the output format as CSV.
- **CSVReportEmitter**
Extends org.eclipse.birt.report.engine.emitter.ContentEmitterAdapter. CSVReportEmitter handles the start and end processing that renders the report container.
- **CSVTags.java**
Defines the comma and new line Strings used when writing to the CSV file.

- CSVWriter

CSVWriter writes the data and label contents of the report to the CSV file, using a call to `java.io.PrintWriter.print()`.

The following section contains more specific information about the implementation details for the classes in the CSV report rendering extension package.

Understanding CSVReportEmitter

CSVReportEmitter is the class that extends ContentEmitterAdapter to output the text content of the report items to a CSV file. CSVReportEmitter instantiates the writer and emitter objects.

CSVReportEmitter implements the following methods:

- `CSVReportEmitter()` instantiates the CSV report emitter class as an `org.eclipse.birt.report.engine.presentation.ContentEmitterVisitor` object, to perform emitter operations, as shown in Listing 19-1.

Listing 19-1 The `CSVReportEmitter()` constructor

```
public CSVReportEmitter()
{
    contentVisitor = new ContentEmitterVisitor( this );
}
```

- `initialize()` performs the following operations required to create an output stream that writes the text contents of the report to the CSV file:

- Obtains a reference to the `IEmitterServices` interface. Instantiates the file and output stream objects, using the specified settings.
- Instantiates the CSV writer object.

Listing 19-2 shows the `initialize()` method.

Listing 19-2 The `initialize()` method

```
public void initialize( IEmitterServices services )
{
    this.services = services;
    Object fd = services.getOption(
        RenderOptionBase.OUTPUT_FILE_NAME );
    File file = null;
    try
    {
        if ( fd != null )
        {
            file = new File( fd.toString() );
            File parent = file.getParentFile();
            if ( parent != null && !parent.exists() )
```

```

        {
            parent.mkdirs( );
        }
        out = new BufferedOutputStream( new
            FileOutputStream( file ) );
    }
}
catch ( FileNotFoundException e )
{
    logger.log( Level.WARNING, e.getMessage( ), e );
}
if ( out == null )
{
    Object value = services.getOption
        ( RenderOptionBase.OUTPUT_STREAM );
    if ( value != null && value instanceof OutputStream )
    {
        out = (OutputStream) value;
    }
    else
    {
        try
        {
            file = new File( REPORT_FILE );
            out =
                new BufferedOutputStream
                    ( new FileOutputStream( file ) );
        }
        catch ( FileNotFoundException e )
        {
            logger.log( Level.SEVERE, e.getMessage( ), e );
        }
    }
}
writer = new CSVWriter( );
}

```

- start() performs the following operations:
 - Obtains a reference to the IReportContent interface, containing accessor methods that get the interfaces to the report content emitters
 - Sets the start emitter logging level and writes to the log file
 - Opens the output file and specifies the encoding scheme as UTF-8
 - Starts the CSV writer

Listing 19-3 shows the start() method.

Listing 19-3 The start() method

```
public void start( IReportContent report )
{
    logger.log( Level.FINE,
        "[CSVReportEmitter] Start emitter." );
    this.report = report;
    writer.open( out, "UTF-8" );
    writer.startWriter( );
}
```

- end() performs the following operations:
 - Sets the end report logging level and writes to the log file
 - Ends the write process and closes the CSV writer
 - Closes the output file

Listing 19-4 shows the end() method.

Listing 19-4 The end() method

```
public void end( IReportContent report )
{
    logger.log( Level.FINE,
        "[CSVReportEmitter] End report." );
    writer.endWriter( );
    writer.close( );
    if( out != null )
    {
        try
        {
            out.close( );
        }
        catch ( IOException e )
        {
            logger.log( Level.WARNING, e.getMessage( ), e );
        }
    }
}
```

Understanding the other CSVReportEmitter methods

The CSVReportEmitter class defines the following additional methods, called at different phases of the report generation process, that provide access to emitters, render options, and style information to facilitate BIRT Report Engine processing:

- startTable()

When writing to the CSV file, the CSV rendering extension must consider the cell position in the row because all the cells end with a comma except the last cell in the row.

The startTable() method uses ITableContent.getColumnCount() to get information about table column numbers and to initialize the protected columnNumbers variable, as shown in Listing 19-5.

Listing 19-5 The startTable() method

```
public void startTable( ITableContent table )
{
    assert table != null;
    tableDepth++;
    columnNumbers = table.getColumnCount();
    ...
}
```

- **startRow()**

At the start of each row, startRow() performs the following operations:

- Calls isRowInFooterBand() to determine if the row is in the header or footer band of a table or group
- Sets exportElement to false if the current table element belongs to a table header, footer, or is an image, since this extension only exports label and data elements to the CSV file
- Sets the currentColumn indicator to 0

Listing 19-6 shows the startRow() code.

Listing 19-6 The startRow() method

```
public void startRow( IRowContent row )
{
    assert row != null;
    if ( tableDepth > 1) {
        logger.log( Level.FINE,
                    "[CSVTableEmitter] Nested tables are not supported." );
        return;
    }
    if ( isRowInFooterBand( row ) )
        exportElement = false;

    currentColumn = 0;
}
```

- **isRowInFooterBand()**

If the row is an instance of band content, isRowInFooterBand() checks the band type. If the band type is a footer, the method returns true, as shown in Listing 19-7.

Listing 19-7 The isRowInFooterBand() method

```
boolean isRowInFooterBand( IRowContent row )
{
```

```

IElement parent = row.getParent( );
if ( !( parent instanceof IBandContent ) )
{
    return false;
}
IBandContent band = ( IBandContent )parent;
if ( band.getBandType( ) == IBandContent.BAND_FOOTER )
{
    return true;
}
return false;
}

```

- **startText()**

If the element is exportable, startText() writes the text value to the CSV output file, as shown in Listing 19-8.

Listing 19-8 The startText() method

```

public void startText( ITextContent text )
{
    if ( tableDepth > 1 ) {
        logger.log( Level.FINE,
                    "[CSVTableEmitter] Nested tables are not supported."
                    );
        return;
    }
    String textValue = text.getText( );
    if ( exportElement )
    {
        writer.text( textValue );
    }
}

```

- **endCell()**

If the current cell is not the last column in the row and the element is exportable, endCell() writes a comma to the CSV output file, as shown in Listing 19-9.

Listing 19-9 The endCell() method

```

public void endCell( ICellContent cell )
{
    if ( ( currentColumn < columnNumbers )
        && exportElement )
    {
        writer.closeTag( CSVTags.TAG_COMMA );
    }
}

```

- **endRow()**

At the end of each row, if the element is exportable, `endRow()` writes a new line or carriage return to the CSV output file, as shown in Listing 19-10.

Listing 19-10 The `endRow()` method

```
public void endRow( IRowContent row )
    if ( exportTableElement )
        writer.closeTag( CSVTags.TAG_CR );
    exportElement = true;
}
```

Understanding CSVTags

The CSVTags class defines the contents of the comma and new line tags, as shown in Listing 19-11.

Listing 19-11 The CSVTags class

```
public class CSVTags
{
    public static final String TAG_COMMA = ",";
    public static final String TAG_CR = "\n";
}
```

Understanding CSVWriter

The CSVWriter class writes the closing tags defined in CSVTags, as shown in Listing 19-12.

Listing 19-12 The `closeTag()` method

```
public void closeTag( String tagName )
{
    printWriter.print( tagName );
}
```

Understanding CSVRenderOption

The `org.eclipse.birt.report.engine.emitter.csv.CSVRenderOption` class extends `org.eclipse.birt.report.engine.api.RenderOption` to add the CSV rendering option to the BIRT Report Engine run time, as shown in Listing 19-13.

Listing 19-13 The CSVRenderOption class

```
package org.eclipse.birt.report.engine.emitter.csv;

import org.eclipse.birt.report.engine.api.RenderOption;
```

```
public class CSVRenderOption extends RenderOption {  
    public static final String CSV = "CSV";  
    public CSVRenderOption( ) {  
    }  
}
```

Testing the CSV report rendering plug-in

To test the CSV report rendering example, you create a Java application that runs a report design in an installation of the BIRT run-time engine. BIRT provides a run-time engine that runs in a stand-alone J2EE application server environment and a preview engine that runs in the BIRT Report Designer.

To test the CSV report rendering plug-in, you perform the following tasks:

- Build the org.eclipse.birt.report.engine.emitter.csv plug-in.
- Deploy the plug-in to the BIRT run-time engine directory.
- Launch a run-time instance of the Eclipse PDE.
- Create a Java application that runs a report design and writes the report's table data to a CSV file.
- Create a report design containing a table that maps to a data source and data set.
- Run the application and examine the output in the CSV file.

You must have previously installed the BIRT run-time engine in the test environment. For more information about downloading and installing the BIRT run-time engine, see the sections on installing the BIRT system earlier in this book or visit the Eclipse BIRT web site at <http://www.eclipse.org/birt>.

The following sections describe the steps required to build and export the plug-ins, launch the Eclipse PDE run-time environment, create the Java application and report design, and test the plug-in example.

How to build and export the org.eclipse.birt.report.engine.emitter.csv plug-in

On PDE Manifest Editor, perform the following tasks:

- 1 On Build, specify the binary build configuration for the plug-in for org.eclipse.birt.report.engine.emitter.csv to include the following items:
 - plugin.xml
 - bin\org.eclipse.birt.report.engine.emitter.csv
 - META-INF\MANIFEST.MF

- 2** On Overview, in Exporting, choose the Export Wizard and perform the following tasks:
- 1 In Options, choose Package plug-ins as individual JAR archives, as shown in Figure 19-8.

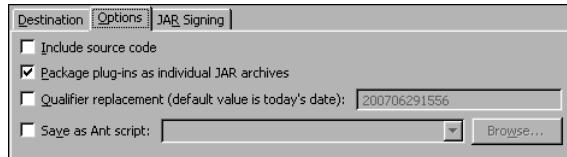


Figure 19-8 Exporting a plug-in option

- 2 In Destination, choose the directory, \$INSTALL_DIR\birt-runtime-2_2_1\ReportEngine, as shown in Figure 19-9. Choose Finish.



Figure 19-9 Exporting a plug-in to BIRT run-time engine

Launching the CSV report rendering plug-in

On PDE Manifest Editor, in Overview, the Testing section contains links to launch a plug-in as a separate Eclipse application in either Run or Debug mode. Figure 19-10 shows Overview for the CSV report rendering extension example in the host instance of the PDE workbench.

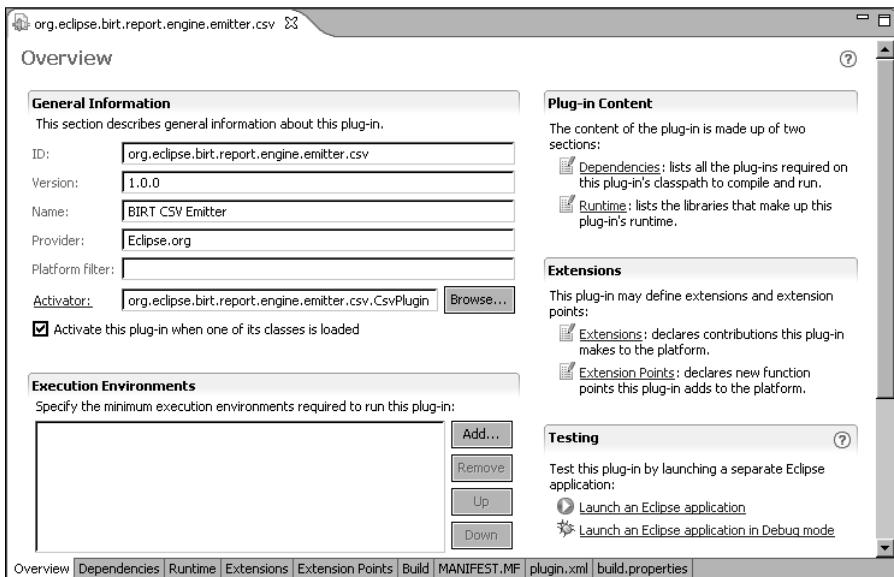


Figure 19-10 Overview information for the CSV report rendering extension

How to launch the CSV report rendering plug-in

- 1 On Eclipse PDE Manifest Editor, in the Testing section of Overview, choose Launch an Eclipse application. The Eclipse PDE launches a run-time instance of the workbench.
- 2 In the run-time instance of the Eclipse PDE workbench, choose Window→Open Perspective→Java. Java opens.

How to create the report execution project

- 1 In Eclipse run-time workbench, choose File→New→Project. New Project appears.
- 2 In New Project—Select a wizard, perform the following tasks:
 - 1 In Wizards, choose Java Project. Choose Next. Create a Java Project appears.
 - 2 In Create a Java Project, perform the following tasks:
 - 1 In Project name, type:
ExecuteReport
 - 2 In Contents, select Create new project in workspace. Choose Next. Java Settings—Source appears.
 - 3 In Java Settings, choose Libraries. Java Settings—Libraries appears.
 - 4 In Libraries, perform the following tasks:
 - 1 Choose Add External JARS. JAR Selection opens.

- 2 On JAR Selection, in Look in, navigate to \$INSTALL_DIR\birt-runtime-2_2_1\ReportEngine\lib and, holding down CTRL, select the following libraries:

 - chartengineapi.jar
 - com.ibm.icu_3.6.1.jar
 - commons-cli-1.0.jar
 - commons-codec-1.3.jar
 - coreapi.jar
 - dataadapterapi.jar
 - dteapi.jar
 - engineapi.jar
 - flute.jar
 - js.jar
 - modelapi.jar
 - modelodaapi.jar
 - odadesignapi.jar
 - org.apache.commons.codec_1.3.0.jar
 - org.eclipse.emf.common_2.2.1.jar
 - org.eclipse.emf.ecore.xmi_2.2.2.jar
 - org.eclipse.emf.ecore_2.2.2.jar
 - org.w3c.css.sac_1.3.0.v200706111724.jar
 - scriptapi.jar
- Choose Open.
- 3 On JAR Selection, in Look in, navigate to \$INSTALL_DIR\birt-runtime-2_2_1\ReportEngine\plugins and select org.eclipse.birt.report.engine.emitter.csv.jar.

Choose Finish. In Package Explorer, the ExecuteReport project appears.

How to create the Java report execution class

- 1 In Eclipse run-time workbench, choose File->New->Class. New Java Class appears.
- 2 On New Java Class, perform the following tasks:

 - 1 In Source folder, type:

ExecuteReport/src

2 In Name, type:

ExecuteReport

3 In Which method stubs would you like to create?, perform the following tasks:

1 Select Public static void main(Strings[] args).

2 Deselect Constructors from superclass.

3 Deselect Inherited abstract methods.

Choose Finish.

In Package Explorer, ExecuteReport.java appears in the ExecuteReport project.

3 Open ExecuteReport.java in Java Editor, and add the required code. The ExecuteReport code is discussed later in this chapter.

4 In Eclipse run-time workbench, compile the project by choosing Project->Build Project.

How to run the CSV report rendering extension

To run the CSV report rendering extension, using the ExecuteReport application, perform the following tasks:

1 In Eclipse run-time workbench, right-click ExecuteReport, and choose Run As->Open Run Dialog from the menu. Run appears.

2 On Run, perform the following tasks:

1 In Java Application, select ExecuteReport, as shown in Figure 19-11.

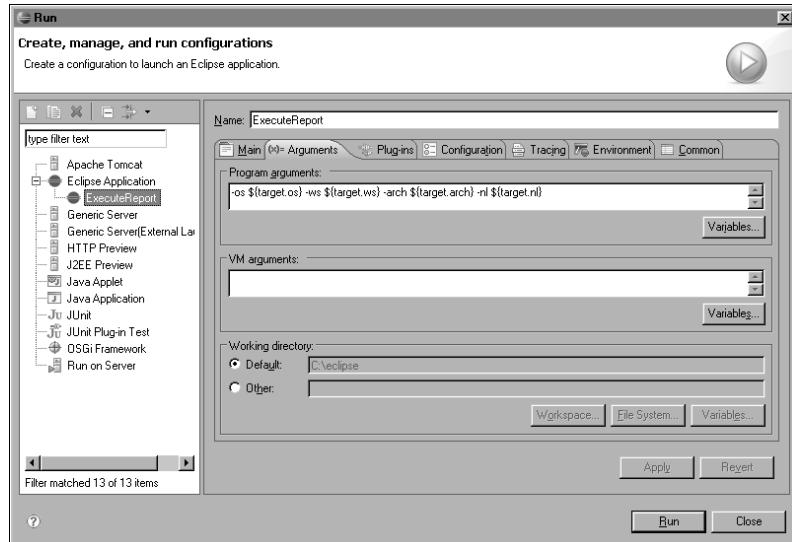


Figure 19-11 Selecting ExecuteReport

- 2 To change the working directory for the launch configuration, perform the following tasks:
 - 1 On Run, choose (x) = Arguments.
 - 2 In Working directory, choose Other. Choose File System. Browse for Folder appears. Select the working directory that contains the ExecuteReports project for the launch configuration. Choose OK.
 - 3 On Run, choose Apply.
- 3 To run the Java application using the launch configuration, choose Run.

How to view the CSV report rendering extension file output

- 1 Navigate to the directory containing the CSV output file. This CSV report rendering extension example writes the CSV file to the following location:
`C:\birt-dev-2_2_1\TestCSVEmitter\ExecuteReport\reports`
- 2 Using a text editor or other tool, open the file, and view its contents.

Figure 19-12 shows the CSV output.

PRODUCTNAME	QUANTITYINSTOCK	MSRP
1969 Harley Davidson Ultimate Chopper	7933	95.7
1952 Alpine Renault 1300	7305	214.3
1996 Moto Guzzi 1100i	6625	118.94
2003 Harley-Davidson Eagle Drag Bike	5582	193.66
1972 Alfa Romeo GTA	3252	136
1962 LanciaA Delta 16V	6791	147.74

Figure 19-12 CSV output

The XML source code for the report design used in this example is discussed later in this chapter.

About ExecuteReport class

The ExecuteReport class runs a BIRT report and renders the output in CSV format, writing the text-based elements of the report to a file. The ExecuteReport class performs the following operations:

- Configures the report engine
- Sets the log configuration and logging level
- Starts the platform and loads the plug-ins
- Gets the report engine factory object from the platform and creates the report engine
- Opens the report design
- Creates a task to run and render the report
- Set the rendering options, such as the output file and format
- Runs the report and destroys the engine
- Shuts down the engine

Listing 19-14 shows the code for the ExecuteReport class in the CSV report rendering extension example.

Listing 19-14 The ExecuteReport class code

```
import java.util.logging.Level;
import org.eclipse.birt.core.framework.Platform;
import org.eclipse.birt.report.engine.api.EngineConfig;
import org.eclipse.birt.report.engine.api.CSVRenderOption;
import org.eclipse.birt.report.engine.api.IReportEngine;
import
    org.eclipse.birt.report.engine.api.IReportEngineFactory;
import org.eclipse.birt.report.engine.api.IReportRunnable;
import org.eclipse.birt.report.engine.api.IRunAndRenderTask;

public class ExecuteReport {

    static void executeReport( ) throws Exception
    {
        IReportEngine engine=null;
        EngineConfig config = null;
        config = new EngineConfig( );
        config.setBIRTHome
            ( "C:/birt-runtime-2_2_1/ReportEngine" );
        config.setLogConfig( "c:/birt/logs", Level.FINE );
        Platform.startup( config );
        IReportEngineFactory factory =
            ( IReportEngineFactory ) Platform.createFactoryObject
            ( IReportEngineFactory
                .EXTENSION_REPORT_ENGINE_FACTORY );
        engine = factory.createReportEngine( config );
        engine.changeLogLevel( Level.WARNING );

        IReportRunnable design =
            engine.openReportDesign
                ( "reports/csvTest.rptdesign" );
        IRunAndRenderTask task =
            engine.createRunAndRenderTask( design );
        String format = "CSV";
        CSVRenderOption csvOptions = new CSVRenderOption( );
        csvOptions.setOutputFormat( format );
        csvOptions.setOutputFileName( "reports/csvTest.csv" );
        task.setRenderOption( csvOptions );

        task.close( );
        engine.destroy( );
        Platform.shutdown( );
        System.out.println("We are done!!!");
    }
    public static void main(String[] args) {
```

```

        try
        {
            executeReport( );
        }
        catch ( Exception e )
        {
            e.printStackTrace();
        }
    }
}

```

About the report design XML code

The XML file for the report design, csvTest.reportdesign, contains the following source code settings, as specified in the report design:

- Data sources, including the ODA plug-in extension ID, driver class, URL, and user
- Data sets, including the ODA JDBC plug-in extension ID, result set properties, and query text
- Page setup, including the page footer
- Body, containing the table structure and properties for the bound data columns, including the header, footer, and detail rows

The report design example specifies a data source that connects to org.eclipse.birt.report.data.oda.sampledb, the BIRT Classic Models sample database. Listing 19-15 shows the XML source code for the report design used to test the CSV rendering example. The sample application runs the report from the reports subfolder in the ExecuteReport project.

Listing 19-15 The report design XML code

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- Written by Eclipse BIRT 2.0 -->
<report xmlns="http://www.eclipse.org/birt/2005/design"
    version="3.2.15" id="1">
    <property name="createdBy">
        Eclipse BIRT Designer Version 2.2.1.r221_v20070924
        Build &lt;2.2.0.v20070924-1550></property>
    <property name="units">in</property>
    <data-sources>
        <oda-data-source
            extensionID=
                "org.eclipse.birt.report.data.oda.jdbc"
                name="Data Source" id="2">
            <property
                name="odaDriverClass">
                org.eclipse.birt.report.data.oda.sampledb.Driver
            </property>
    
```

```
<property
    name="odaURL">jdbc:classicmodels:sampledbs
</property>
<property name="odaUser">ClassicModels</property>
</oda-data-source>
</data-sources>
<data-sets>
    <oda-data-set
        extensionID=
            "org.eclipse.birt.report.data.oda.jdbc
             .JdbcSelectDataSet" name="Data Set" id="3">
        <list-property name="columnHints">
            <structure>
                <property name=
                    "columnName">PRODUCTNAME</property>
                <property name=
                    "displayName">PRODUCTNAME</property>
            </structure>
            <structure>
                <property name=
                    "columnName">QUANTITYINSTOCK</property>
                <property name=
                    "displayName">QUANTITYINSTOCK</property>
            </structure>
            <structure>
                <property name="columnName">MSRP</property>
                <property name="displayName">MSRP</property>
            </structure>
        </list-property>
        <structure name="cachedMetaData">
            <list-property name="resultSet">
                <structure>
                    <property name="position">1</property>
                    <property name=
                        "name">PRODUCTNAME
                    </property>
                    <property
                        name="dataType">string
                    </property>
                </structure>
                <structure>
                    <property name="position">2</property>
                    <property
                        name="name">QUANTITYINSTOCK
                    </property>
                    <property
                        name="dataType">integer
                    </property>
                </structure>
                <structure>
                    <property name="position">3</property>
                    <property name="name">MSRP</property>
                    <property name="dataType">float</property>
                </structure>
            </list-property>
        </structure>
    </oda-data-set>
</data-sets>
```

```

        </structure>
        </list-property>
    </structure>
<property name="dataSource">Data Source</property>
    <list-property name="resultSet">
        <structure>
            <property name="position">1</property>
            <property name="name">PRODUCTNAME</property>
            <property name=
                "nativeName">PRODUCTNAME</property>
            <property name="dataType">string</property>
            <property name=
                "nativeDataType">12</property>
        </structure>
        <structure>
            <property name="position">2</property>
            <property name=
                "name">QUANTITYINSTOCK</property>
            <property name=
                "nativeName">QUANTITYINSTOCK</property>
            <property name="dataType">integer</property>
            <property name="nativeDataType">4</property>
        </structure>
        <structure>
            <property name="position">3</property>
            <property name="name">MSRP</property>
            <property name="nativeName">MSRP</property>
            <property name="dataType">float</property>
            <property name="nativeDataType">8</property>
        </structure>
    </list-property>
<property name="queryText">
    select CLASSICMODELS.PRODUCTS.PRODUCTNAME,
           CLASSICMODELS.PRODUCTS.QUANTITYINSTOCK,
           CLASSICMODELS.PRODUCTS.MSRP
    from CLASSICMODELS.PRODUCTS</property>
</oda-data-set>
</data-sets>
<page-setup>
    <simple-master-page name="Simple MasterPage" id="4">
        <page-footer>
            <text id="5">
                <property name="contentType">html</property>
                <text-property name="content">
                    <![CDATA[<value-of>new Date()</value-of>]]>
                </text-property>
            </text>
        </page-footer>
    </simple-master-page>
</page-setup>
<body>
    <table id="6">
        <property name="width">100%</property>

```

```
<property name="dataSet">Data Set</property>
<list-property name="boundDataColumns">
    <structure>
        <property name="name">PRODUCTNAME</property>
        <expression
            name="expression">dataSetRow["PRODUCTNAME"]</expression>
    </structure>
    <structure>
        <property
            name="name">QUANTITYINSTOCK</property>
        <expression
            name="expression">dataSetRow["QUANTITYINSTOCK"]</expression>
    </structure>
    <structure>
        <property name="name">MSRP</property>
        <expression
            name="expression">dataSetRow["MSRP"]</expression>
    </structure>
</list-property>
<column id="28"/>
<column id="29"/>
<column id="30"/>
<header>
    <row id="7">
        <cell id="8">
            <property name="colSpan">3</property>
            <property name="rowSpan">1</property>
            <property name="textAlign">center</property>
            <label id="9">
                <property
                    name="fontSize">x-large</property>
                <property
                    name="fontWeight">bold</property>
                <property
                    name="textAlign">center</property>
                <list-property name="visibility">
                    <structure>
                        <property name="format">all</property>
                        <expression name=
                            "valueExpr">true</expression>
                    </structure>
                </list-property>
            <text-property
                name="text">Report
            </text-property>
```

```

        </label>
    </cell>
</row>
<row id="10">
    <cell id="11">
        <label id="12">
            <text-property
                name="text">PRODUCTNAME
            </text-property>
        </label>
    </cell>
    <cell id="13">
        <label id="14">
            <text-property
                name="text">QUANTITYINSTOCK
            </text-property>
        </label>
    </cell>
    <cell id="15">
        <label id="16">
            <text-property
                name="text">MSRP
            </text-property>
        </label>
    </cell>
</row>
</header>
<detail>
    <row id="17">
        <cell id="18">
            <data id="19">
                <property
                    name="resultSetColumn">PRODUCTNAME
                </property>
            </data>
        </cell>
        <cell id="20">
            <data id="21">
                <property
                    name="resultSetColumn">QUANTITYINSTOCK
                </property>
            </data>
        </cell>
        <cell id="22">
            <data id="23">
                <property
                    name="resultSetColumn">MSRP
                </property>
            </data>
        </cell>
    </row>
</detail>
<footer>

```

```

<row id="24">
<cell id="25"/>
<cell id="26"/>
<cell id="27"/>
</row>
</footer>
</table>
</body>
</report>

```

BIRT Report Engine can render a report design for output using a standard emitter extension or a customized emitter extension, such as this CSV rendering example.

Developing an XML report rendering extension

The sample XML report rendering extension is a plug-in that can export BIRT report data in XML format. Typically, report developers render BIRT report data to XML to enable sharing data with another application.

For example, business-to-business (B2B) systems must transmit data to customers and trading partners in a consistent way that supports interoperability according to Electronic Data Interchange (EDI) standards. These systems use specialized forms of XML such as Electronic Business eXtensible Markup Language (ebXML). A custom XML report rendering extension can render BIRT report data in a format that is consistent with this established standard.

The sample XML report rendering extension contains the following features:

- Exports BIRT report data in XML format
The XML report rendering plug-in renders each report element and writes to the output file, <report_name>.xml.
- Defines a public API for rendering BIRT reports in XML format
The plug-in extends the functionality defined by the org.eclipse.birt.report.engine.emitter extension point defined in the org.eclipse.birt.report.engine plug-in.

■ Allows the user to specify an XML schema for formatting output
During the rendering process, the sample plug-in processes all the elements in the report design, exporting XML properties and related data to the output file. Optionally, the plug-in supports mapping the report elements to an XML schema to provide additional formatting for output.

The plug-in defines these mappings in the property file, <report_name>.xmlemitter. The plug-in reads the property file at run time and loads the custom tags.

Creating an XML report rendering plug-in project

Create a new plug-in project for the XML report rendering extension using the Eclipse PDE.

How to create the XML report rendering plug-in project

- 1 From the Eclipse PDE menu, choose File→New→Project. New Project appears.
- 2 On New Project, select Plug-in Project. Choose Next. New Plug-in Project appears.
- 3 In Plug-in Project, modify the settings, as shown in Table 19-5.

Table 19-5 Values for Plug-in Project fields

Section	Option	Value
Plug-in Project	Project name	org.eclipse.birt.report.engine.emitter.xml
	Use default location	Selected
Project Settings	Location	Not available when you select Use default location
	Create a Java project	Selected
Target Platform	Source folder	src
	Output folder	bin
Target Platform	Eclipse version	3.3
	an OSGi framework	Not selected

Choose Next. Plug-in Content appears.

- 4 In Plug-in Content, modify the settings, as shown in Table 19-6.

Table 19-6 Values for Plug-in Content fields

Section	Option	Value
Plug-in Properties	Plug-in ID	org.eclipse.birt.report.engine.emitter.xml
	Plug-in Version	1.0.0
	Plug-in Name	BIRT XML Emitter
	Plug-in Provider	yourCompany.com or leave blank
	Classpath	xmlEmitter.jar or leave blank

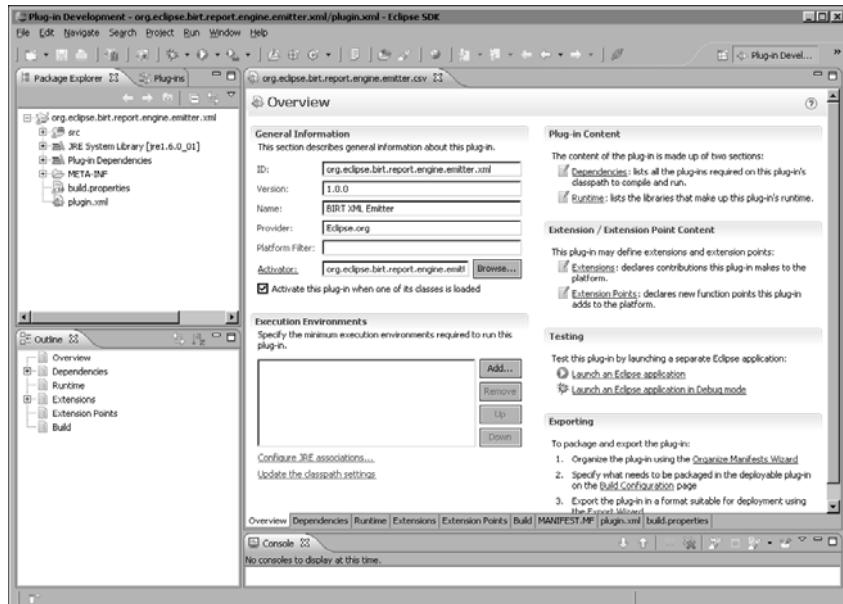
(continues)

Table 19-6 Values for Plug-in Content fields (*continued*)

Section	Option	Value
Plug-in Options	Generate an activator, a Java class that controls the plug-in's life cycle	Selected
	Activator	org.eclipse.birt.report.engine.emitter.xml.XmlPlugin
	This plug-in will make contributions to the UI	Not selected
Rich Client Application	Would you like to create a rich client application?	No

Choose Finish.

The XML report rendering extension project appears in the Eclipse PDE workbench, as shown in Figure 19-13.

**Figure 19-13** XML report rendering extension project

Defining the dependencies for the XML report rendering extension

To compile and run the XML report rendering example, you specify the org.eclipse.birt.report.engine plug-in, which must be available on the classpath for the XML rendering extension.

Declaring the emitters extension point

To implement the XML report rendering extension, specify the org.eclipse.birt.report.engine.emitters extension point and add the extension element details.

How to specify the extension point

- 1 On PDE Manifest Editor, choose Extensions.
- 2 In All Extensions, choose Add. New Extension—Extension Point Selection appears.
- 3 In Available extension points, select the following plug-in:
`org.eclipse.birt.report.engine.emitters`
Choose Finish.
- 4 In All Extensions, right-click the extension point, org.eclipse.birt.report.engine.emitters, and choose the extension element, emitter.
- 5 In Extension Element Details, specify the properties for the emitter extension element, emitter, as shown in Table 19-7.

Table 19-7 Property values for the emitter extension element

Property	Value
class	org.eclipse.birt.report.engine.emitter.xml.XMLReportEmitter
format	xml
contentType	xml
id	org.eclipse.birt.report.engine.emitter.xml

Understanding the sample XML report rendering extension

The XML report rendering extension extends the report emitter interfaces and XML writer in org.eclipse.birt.report.engine.emitter. The extension example provides access to the report container, pages, tables, rows, cells text, labels, data, images, hyperlinks, and other contents at different phases of the report generation process.

The example writes the contents of the report to an XML output file. The example creates the XML file in the same folder as the exported report. The output file name is the name of the report with a .xml extension. The example provides only limited error checking.

The following section provides a general description of the code-based extensions a developer must make to develop an XML report rendering extension after defining the plug-in framework in the Eclipse PDE.

Understanding the XML report rendering extension package

The implementation package for the XML report rendering extension example, org.eclipse.birt.report.engine.emitter.xml, contains the following classes:

- **XMLPlugin**
The plug-in run-time class for the report item extension example.
- **XMLReportEmitter**
Handles the start and end processing that renders the report container.
- **XMLRenderOption**
Integrates the plug-in with BIRT Report Engine, specifying configuration information, including the output format as XML.
- **XMLETags.java**
Defines the controls and associated property lists used when writing to the XML file.
- **XMLFileWriter**
Writes the XML version, text, image, data, label, and report tag content of the report to the XML output file.
- **LoadExportSchema**
Loads the XML Schema file, if one exists, to replace the default values specified for the XML version, text, image, data, label, and report tags. An accessor method for each tag returns the value to XMLReportEmitter for output to the export file.

The following section contains more specific information about the implementation details for the classes in the XML report rendering extension package.

Understanding XMLReportEmitter

XMLReportEmitter writes the contents of the report to an XML file.

XMLReportEmitter instantiates the writer and emitter objects and handles the start and end processing that renders the report container.

XMLReportEmitter exports the XML version, text, image, data, label, and report tag content of the report to the XML output file.

XMLReportEmitter implements the following methods:

- **XMLReportEmitter()** instantiates the XML report emitter class as an org.eclipse.birt.report.engine.presentation.ContentEmitterVisitor object to perform emitter operations.

- `initialize()` performs the following operations required to create an output stream that writes the report contents to the XML file, similar to the CSV report rendering extension:
 - Obtains a reference to the `IEmitterServices` interface
 - Instantiates the file and output stream objects, using the specified settings
 - Instantiates the XML writer object
- `start()` performs the following operations:
 - Obtains a reference to the `IReportContent` interface, containing accessor methods that get the interfaces to the report content emitters
 - Sets the start emitter logging level and writes to the log file
 - If an optional XML Schema file exists:
 - Locates the XML Schema file for the report
 - Instantiates a `LoadExportSchema` object to read the XML Schema file
 - Opens the output file, specifying the encoding scheme as UTF-8
 - Starts the XML writer
 - Writes the start tag, which specifies the `<xml>` tag, including the version and encoding schema, to the output file
 - Writes the `<report>` tag, which specifies the report name and other properties in the report property list to the output file

Listing 19-16 shows the `start()` method.

Listing 19-16 The `start()` method

```
public void start( IReportContent report )
{
  logger.log( Level.FINE,
    "[XMLReportEmitter] Start emitter." );
  String fileName =
    report.getDesign( ).getReportDesign( ).getFileName( );
  int pos = fileName.indexOf("/");
  String fn = fileName.substring(pos+1,fileName.length( ));
  fileName = fn;
  if (fileName.length( ) > 0) {
    pos = fileName.lastIndexOf(".");
    if ( pos > 0 )
      fileName = fileName.substring(0, pos);

    fileName = fileName + ".xmlemitter";
    pos = fileName.lastIndexOf("/");
  }
}
```

```

String propFileName =
    fileName.substring( pos+1 , fileName.length() );
String resourceFolder =
    report.getDesign().getReportDesign()
        .getResourceFolder();
if ( fileExists(resourceFolder + "/"
+ propFileName) )
    exportSchema = new LoadExportSchema(
        resourceFolder + "/" + propFileName );
else
    if ( fileExists(fileName))
        exportSchema =
            new LoadExportSchema( fileName );
    else exportSchema = new LoadExportSchema( "" );
}
this.report = report;
writer.open( out, "UTF-8" );
writer.startWriter();

writer.closeTag( exportSchema.getExportStartTag() );
writer.closeTag( XMLTags.TAG_CR );

String rp = exportSchema.getExportReportTag();
for (int i = 0;i < XMLTags.rPropList.length;i++)
{
    if (exportSchema.isPropertyRequired(
        XMLTags.rPropList[i], rp))
    {
        String propName = getReportPropValue(i,report);
        rp = replaceTag( rp, "??"
            +XMLTags.rPropList[i], propName );
    }
}
writer.writeCode( rp );
writer.closeTag( XMLTags.TAG_CR );
}

```

- end() performs the following operations, similar to the CSV rendering extension:
 - Sets the end report logging level and writes to the log file
 - Ends the write process and closes the XML writer
 - Closes the output file

Understanding the other XMLReportEmitter methods

The XMLReportEmitter class defines the following additional methods, called at different phases of the report generation process, that provide access to the report container, pages, tables, rows, cells text, labels, data, images,

hyperlinks, and other contents. The following examples show the processing for a label:

- `startLabel()` performs the following operations:
 - Calls `LoadExportSchema.getExportLabelTag()` to get the pattern for the `<label>` tag specified in the `<report_name>.xmlemitter` property file. If the property file does not exist, the plug-in uses the following default pattern specified in the `LoadExportSchema` class:

```
<label>??value</label>
```
 - Iterates through the following label properties list defined in `XMLTags` to determine the properties required by the report:

```
static String[ ] lPropList = {"Bookmark","Height","Hyperlink","InlineStyle", "Name","TOC","Width","X","Y" };
```
 - Calls `getLabelPropValue()` to obtain each required property value and substitute the value in the `<label>` tag expression.
 - Calls `startText()` and `XMLFileWriter.closeTag()` to write the `<label>` tag to the output file.

Listing 19-17 shows the `startLabel()` method code.

Listing 19-17 The `startLabel()` method

```
public void startLabel( ILabelContent label )
{
    String lbl = exportSchema.getExportLabelTag( );
    int len = XMLTags.lPropList.length;
    for (int i = 0;i < XMLTags.lPropList.length;i++)
    {
        if (exportSchema.isPropertyRequired(
            XMLTags.lPropList[i], lbl))
        {
            String propValue = getLabelPropValue(i,label);
            lbl = replaceTag( lbl, "??"+XMLTags.lPropList[i],
                propValue );
        }
    }
    startText( label, lbl );
    writer.closeTag( XMLTags.TAG_CR );
}
```

- `startText()` performs the following operations:
 - Sets the start text logging level and writes to the log file
 - Uses `ITextContent.getText()` to get the label text value
 - Writes the `<label>` tag to the output file

Listing 19-18 shows the `startText()` method code.

Listing 19-18 The startText() method

```
public void startText( ITextContent text, String exportTag )
{
    logger.log( Level.FINE,
        "[XMLReportEmitter] Start text" );
    String textValue = text.getText( );
    writer.writeCode( replaceTag( exportTag,
        XMLTags.valueTag, textValue ) );
}
```

- `getLabelPropValue()` performs the following operations:
 - Calls the appropriate `IContent` accessor method to obtain the property value
 - Returns the value to `startLabel()` for substitution in the `<label>` tag and writing the tag to the XML output file

Listing 19-19 shows the `getLabelPropValue()` method code.

Listing 19-19 The getLabelPropValue() method

```
private String getLabelPropValue( int property,
    ILabelContent label )
{
    String propValue;

    switch (property) {
        case 0: // "Bookmark":
            propValue = label.getBookmark( );
            break;
        case 1: // "Height":
            if ( label.getHeight( ) != null )
                propValue = label.getHeight().toString( );
            else
                propValue = "";
            break;
        case 2: // "Hyperlink":
            if ( label.getHyperlinkAction( ) != null )
                propValue =
                    label.getHyperlinkAction( ).getHyperlink( );
            else propValue = "";
            break;
        ...
        case 8: // "Y":
            if ( label.getY( ) != null )
                propValue = label.getY().toString( );
            else
                propValue = "";
            break;
        default: propValue = "";
            break;
    }
}
```

```

        if ( propValue == null )
            propValue = "";
        return propValue;
    }
}

```

Understanding XMLTags

The XMLTags class defines the controls and associated property lists used in analyzing the report contents, as shown in Listing 19-20.

Listing 19-20 The XMLTags class

```

public class XMLTags
{
    public static final String TAG_CR = "\n" ;
    static String valueTag = "?value";
    static String labelControl = "label";
    static String textControl = "text";
    static String imageControl = "image";
    static String dataControl = "data";
    static String reportControl = "report";
    static String startControl = "start";
    static String endControl = "end";

    static String[] iPropList =
        {"Bookmark", "Height", "Hyperlink", "ImageMap",
         "InlineStyle", "MIMEType", "Name", "Style", "TOC", "URI",
         "Width", "X", "Y"};
    static String[] dPropList =
        {"Bookmark", "Height", "Hyperlink", "InlineStyle", "Name",
         "Style", "TOC", "Width", "X", "Y"};
    static String[] lPropList =
        {"Bookmark", "Height", "Hyperlink", "InlineStyle", "Name",
         "TOC", "Width", "X", "Y"};
    static String[] tPropList =
        {"Bookmark", "Height", "Hyperlink", "InlineStyle", "Name",
         "Style", "Text", "TOC", "Width", "X", "Y"};
    static String[] rPropList =
        {"TotalPages", "TOCTree", "Name"};
}

```

Understanding XMLFileWriter

The XMLFileWriter class writes the closing tag, similar to the CSV report rendering extension.

Understanding XMLRenderOption

The org.eclipse.birt.report.engine.emitter.xml.XMLRenderOption class adds the XML rendering option to the BIRT Report Engine run time, as shown in Listing 19-21.

Listing 19-21 The XMLRenderOption class

```
package org.eclipse.birt.report.engine.emitter.xml;  
import org.eclipse.birt.report.engine.api.RenderOption;  
public class XMLRenderOption extends RenderOption{  
    public static final String XML = "XML";  
    protected String configPath= "";  
    public XMLRenderOption( ) {  
    }  
    public void setExportConfigFile( String config )  
    {  
        this.configPath = config;  
    }  
    public String getExportConfigFile()  
    {  
        return configPath;  
    }  
}
```

Understanding LoadExportSchema

The `org.eclipse.birt.report.engine.emitter.xml.LoadExportSchema` class optionally loads an XML Schema by performing the following operations:

- Specifies the default substitution patterns for the XML tags
- Calls the `readSchemaFile()` method
- Specifies an accessor method for each tag that returns a value to `XMLReportEmitter` for output to the export file

Listing 19-22 shows the specification of the default substitution patterns for the XML tags and the constructor, which calls the `readSchemaFile()` method.

Listing 19-22 The LoadExportSchema class

```
package org.eclipse.birt.report.engine.emitter.xml;  
...  
public class LoadExportSchema{  
    protected String fileName = "";  
    protected String startTag = "<?xml version=\"1.0\""  
encoding="UTF-8\"?>";  
    protected String textTag = "<text>??value</text>";
```

```

protected String imageTag = "<image>??value</image>";
protected String dataTag = "<data>??value</data>";
protected String labelTag = "<label>??value</label>";
protected String endTag = "</report>";
protected String reportTag = "<report:??name>";

public LoadExportSchema(String fileName)
{
    if ( fileName.length() > 0 )
    {
        this.fileName = fileName;
        readSchemaFile();
    }
}

```

The `readSchemaFile()` method reads the XML Schema file, one line at a time, replacing the default values for the patterns of the XML version, text, image, data, label, and report tags with the values specified in the XML Schema file.

Listing 19-23 The `readSchemaFile()` method

```

private void readSchemaFile()
{
    BufferedReader input = null;
    try
    {
        input = new BufferedReader(
            new FileReader(fileName) );
        String line = null; //not declared within while loop
        while (( line = input.readLine() ) != null){
            int pos = line.indexOf("=");
            if ( pos > 0 )
            {
                String index = line.substring(0, pos );
                String indexTag = line.substring(pos + 1,
                    line.length());
                if ( index.equalsIgnoreCase(
                    XMLTags.labelControl ) )
                {
                    labelTag = indexTag;
                }
                if ( index.equalsIgnoreCase( XMLTags.imageControl ) )
                {
                    imageTag = indexTag;
                }
                if ( index.equalsIgnoreCase( XMLTags.dataControl ) )
                {
                    dataTag = indexTag;
                }
                if ( index.equalsIgnoreCase( XMLTags.startControl ) )
                {

```

```
        startTag = indexTag;
    }
    if ( index.equalsIgnoreCase( XMLTags.endControl ) )
    {
        endTag = indexTag;
    }
    if ( index.equalsIgnoreCase( XMLTags.reportControl ) )
    {
        reportTag = indexTag;
    }
}
catch (FileNotFoundException ex)
{
    ex.printStackTrace( );
}
catch (IOException ex)
{
    ex.printStackTrace( );
}
finally
{
    try
    {
        if (input!= null)
        {
            input.close( );
        }
    }
    catch (IOException ex)
    {
        ex.printStackTrace( );
    }
}
```

Listing 19-24 shows the values of the patterns for the XML version, report, label, text, image, and data tags specified in the XML Schema file.

Listing 19-24 The XML Schema file

```
start=<?xml version="1.0" encoding="UTF-8"?>
report=<report name=?name>
label=<label name=?name hyperlink=?hyperlink>??value</label>
text=<text name=?name>??value</text>
image=<image name=?name>??value</image>
data=<data>??value</data>
end=</report>
```

Testing the XML report rendering plug-in

To test the XML report rendering example, create a Java application that runs a report design in an installation of the BIRT run-time engine, similar to the application created to run the CSV report rendering example.

To test the XML report rendering plug-in, perform the following tasks:

- Build the org.eclipse.birt.report.engine.emitter.xml plug-in.
- Deploy the plug-in to the BIRT run-time engine directory.
- Launch a run-time instance of the Eclipse PDE.
- Create a Java application that runs the report design and writes the report's data to an XML file.
- Create a report design containing a table that maps to a data source and data set.
- Run the application and examine the XML in the output file.

Figure 19-14 shows the report design used in the XML report rendering example.

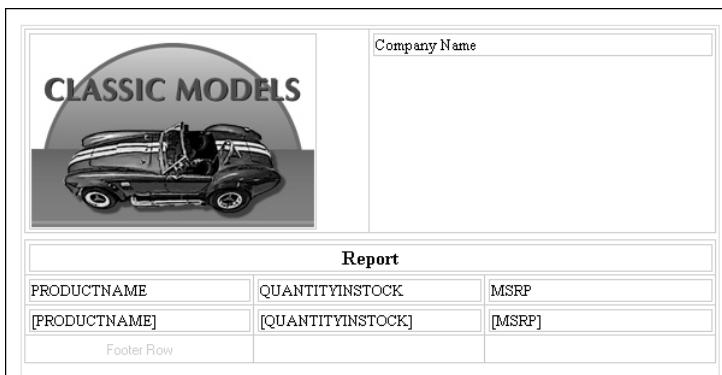


Figure 19-14 Report design for the XML report rendering example

Listing 19-25 shows the contents of the XML output file, containing XML version, report, image, label, and data tags for an executed report.

Listing 19-25 The XML output file

```
<?xml version="1.0" encoding="UTF-8"?>
<report name=
  C:/IANA/2007/runtime-XMLEmitter/ExecuteXMLReport/reports/
  xmlReport.rptdesign>
<image name=>
  /9j/4AAQSkZJRgABAgEBLAEsAAD
  /4RVaRXhpZgAATU0AKgAAAAgABwESAAMAAAABAAEAAAeAAUA
  ...
  7PMv9I9nV05cj8b7MV9zB/gh8cf/2Q==
</image>
<label
  name= hyperlink=http://www.actuate.com>Company Name
</label>
<label name= hyperlink=>Report</label>
<label name= hyperlink=>PRODUCTNAME</label>
<label name= hyperlink=>QUANTITYINSTOCK</label>
```

```
<label name= hyperlink=>MSRP</label>
<data>1969 Harley Davidson Ultimate Chopper</data>
<data>7933</data>
<data>95.7</data>
<data>1952 Alpine Renault 1300</data>
<data>7305</data>
<data>214.3</data>
<data>1996 Moto Guzzi 1100i</data>
<data>6625</data>
<data>118.94</data>
<data>2003 Harley-Davidson Eagle Drag Bike</data>
<data>5582</data>
<data>193.66</data>
...
<data>American Airlines: MD-11S</data>
<data>8820</data>
<data>74.03</data>
<data>Boeing X-32A JSF</data>
<data>4857</data>
<data>49.66</data>
<data>Pont Yacht</data>
<data>414</data>
<data>54.6</data>
</report>
```

20

Developing an ODA Extension

BIRT uses the Eclipse Data Tools Platform (DTP) open data access (ODA) API to build a driver that connects to a data source and retrieves data for a report. This API defines interfaces and classes that manage the following tasks:

- Connecting to a data source
- Preparing and executing a query
- Handling data and metadata in a result set
- Mapping between the object representation of data and the data source

Eclipse DTP also provides tools and support for SQL development, locales, logging, and other special types of processing. For more information about the Eclipse DTP project, see <http://www.eclipse.org/datatools>.

The ODA framework is a key component of the DTP. ODA presents the Java developer with a robust architecture to extend the capabilities of BIRT by being able to report on custom data sources.

The framework provides new project wizards to create plug-in projects for ODA run-time and designer extensions. The generated plug-in projects include class templates and default implementation. They help ODA extension developers to expedite the development of customized ODA data source extensions.

This chapter shows how to develop an ODA extension using examples that extend the `org.eclipse.datatools.connectivity.oda.dataSource` extension point to provide access to the following data sources:

- CSV file

Uses the new DTP ODA wizards to create a plug-in project that accesses a CSV data source. DTP ODA interfaces are similar to JDBC interfaces with

extensions that support retrieving data from relational and non-relational database sources.

- Relational database

Uses Hibernate Core for Java, an object-oriented software system for generating SQL and handling JDBC result sets. Hibernate Query Language (HQL) provides a SQL-transparent extension that makes the DTP ODA extension portable to all relational databases. Hibernate also supports developing a query in the native SQL dialect of a database.

Hibernate is free, open-source software licensed under the GNU Lesser General Public License (LGPL). For more information about Hibernate, see <http://www.hibernate.org/>.

Understanding an ODA extension

A BIRT report design specifies the type of data access and data transformations required to generate a report. All data comes from an external data source. The BIRT data engine supports the DTP ODA framework. The DTP ODA framework provides access to standard and custom data sources using an open API.

Using the DTP ODA framework makes it possible to create a plug-in driver to any external data source. BIRT uses DTP ODA extension points for the report designer and report generation environments.

A DTP ODA extension adds a new data source driver to the BIRT framework by implementing the following extension points:

- ODA data source

`org.eclipse.datatools.connectivity.oda.dataSource` supports the extension of BIRT design-time and run-time data source access. The XML schema file, `org.eclipse.datatools.connectivity.oda/schema/dataSource.exsd`, describes this extension point.

- ODA user interface

`org.eclipse.datatools.connectivity.oda.design.ui.dataSource` supports optionally adding an integrated user interface for an ODA driver to BIRT Report Designer. The plug-in can provide user interface support that allows a report designer to specify the data source and edit the data set. The XML schema file, `org.eclipse.datatools.connectivity.oda.design.ui/schema/dataSource.exsd`, describes this extension point.

- ODA connection profile

`org.eclipse.datatools.connectivity.oda.connectionProfile` supports optionally adding different types of connection profiles to an ODA driver user interface for BIRT Report Designer. A connection profile can define a category or set of configuration types such as JDBC connection profiles.

This user interface must define a corresponding newWizard element for creating the resource. The XML schema file, org.eclipse.datatools.connectivity/schema/connectionProfile.exsd, describes this extension point.

- ODA connection properties page

org.eclipse.ui.propertyPages supports optionally adding a page for editing the properties of a connection profile. The XML schema file, org.eclipse.ui.propertyPages/schema/propertyPages.exsd, describes this extension point.

For more information on the DTP ODA APIs, see the Javadoc for the org.eclipse.datatools.connectivity.oda package hierarchy. The Javadoc is in the DTP Software Development Kit (SDK) available from the Eclipse Data Tools Platform project at <http://www.eclipse.org/datatools>.

Developing the CSV ODA driver extensions

Eclipse DTP provides two ODA plug-in template wizards, one for ODA data source runtime driver, another for ODA data source designer. Each wizard automatically creates a new plug-in project, with auto generated implementations of related ODA extension points. The auto-generated Java classes implement method stubs that support a single result set and input parameters. The classes have hard-coded result set data so that an ODA extension developer can immediately verify that the generated ODA driver plug-ins work fine with an ODA consumer, for example, BIRT.

After generating the plug-ins, the developers go through all the TODO tags in the generated source code to customize as appropriate for their own custom data sources behavior.

You develop the CSV ODA extensions by performing the following tasks:

- Download the required BIRT source code from the Eclipse CVS repository.
- Create two new projects using the ODA wizards in the Eclipse PDE to implement the following plug-ins:
 - CSV ODA driver to access the data source
 - CSV ODA user interface (UI) to select the data file and available data columns in BIRT Report Designer
- Extend the source code in the CSV ODA plug-in projects by adding new functionality at the defined extension points.
- Test and deploy the extensions in the run-time environment.

You can download the source code for the CSV ODA driver extension examples at <http://www.actuate.com/birt/contributions>

About the CSV ODA plug-ins

The CSV ODA extensions require the following two plug-ins:

- `org.eclipse.birt.report.data.oda.csv`

The CSV ODA data source plug-in extends the functionality defined by the extension point, `org.eclipse.datatools.connectivity.oda.dataSource`, to create the CSV ODA driver. The first row of the CSV input file contains the column names. The remaining rows, separated by new line markers, contain the data fields, separated by commas. The `org.eclipse.birt.report.data.oda.csv` plug-in contains the database classes and data structures, such as data types, result set, metadata result set, and query used to handle data in a BIRT report.

The `org.eclipse.datatools.connectivity.oda.dataSource` extension point is in the Eclipse DTP project and is part of the `org.eclipse.datatools.connectivity.oda` plug-in. This plug-in is available from the CVS repository in `/home/datatools`.

- `org.eclipse.birt.report.data.csv.ui`

The CSV ODA UI plug-in extends the functionality defined by the `org.eclipse.datatools.connectivity.connectionProfile`, `org.eclipse.ui.propertyPages`, and `org.eclipse.datatools.connectivity.oda.design.ui.dataSource` extension points. The UI consists of the following two pages:

- The data source page specifies and validates the path and name of the CSV file.
- The data set page shows the selected data file and columns available in the file. By default, the UI selects all the columns in the data set.

Downloading BIRT source code from the CVS repository

The CSV ODA driver plug-in, `org.eclipse.birt.report.data.oda.csv`, requires the `org.eclipse.datatools.connectivity.oda` plug-in.

The CSV ODA UI extension, `org.eclipse.birt.report.data.csv.ui`, requires the following Eclipse plug-ins:

- `org.eclipse.core.runtime`
- `org.eclipse.ui`
- `org.eclipse.datatools.connectivity.oda.design.ui`
- `org.eclipse.birt.report.data.oda.csv`

For the `org.eclipse.birt.report.data.oda.csv` plug-in, you extend only the Java classes in the `org.eclipse.datatools.connectivity.oda` plug-in. For the `org.eclipse.birt.report.data.csv.ui` plug-in, you extend the Java classes in the `org.eclipse.datatools.connectivity.oda.design.ui` plug-in.

Eclipse makes source code available to the developer community in the CVS repository. To compile, you do not need the source code for the plug-ins. You can configure the system to use the JAR files in the `eclipse\plugins` folder. To debug, you may need the source code for all the required BIRT and DTP plug-ins.

Implementing the CSV ODA driver plug-in

This section describes how to implement an ODA driver plug-in, using the CSV ODA driver plug-in as an example. To create an ODA driver plug-in, perform the following tasks:

- Create the ODA driver plug-in project.
- Define the dependencies.
- Specify the run-time archive.
- Declare the ODA extension points.

You can create the CSV ODA driver plug-in project, `org.eclipse.birt.report.data.oda.csv`, in the Eclipse PDE. This section describes how to create the plug-in project using the New Plug-in Project wizard.

How to create the CSV ODA driver plug-in project

- 1 From the Eclipse PDE menu, choose `File→New→Project`.
- 2 On `New Project`—Select a wizard, open Business Intelligence and Reporting Tools and select ODA Runtime Driver Plug-in Project as shown in Figure 20-1. Choose `Next`. `New Plug-in Project` appears.

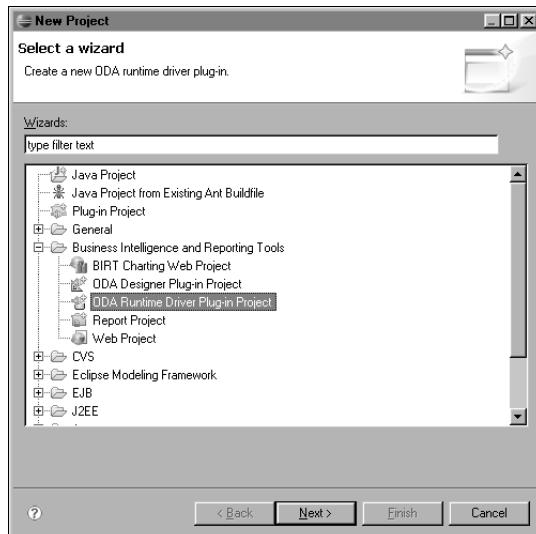


Figure 20-1 Specifying the ODA Runtime Driver Plug-in Project

- 3** In Plug-in Project, modify the settings as shown in Table 20-1. Choose Next. Plug-in Content appears.

Table 20-1 Settings for Plug-in Project options

Section	Option	Value
Plug-in Project	Project name	org.eclipse.birt.report.data.oda.csv
	Use default location	Selected
	Location	Not available when you select Use default location
Project Settings	Create a Java project	Selected
	Source folder	src
	Output folder	bin
Target Platform	Eclipse version	3.3
	OSGi framework	Deselected

- 4** In Plug-in Content, modify the settings as shown in Table 20-2. Choose Next. Templates appears.

Table 20-2 Settings for Plug-in Content options

Section	Option	Value
Plug-in Properties	Plug-in ID	org.eclipse.birt.report.data.oda.csv
	Plug-in Version	1.0.0
	Plug-in Name	CSV ODA Driver
	Plug-in Provider	yourCompany.com or leave blank
	Classpath	
Plug-in Options	Generate an activator, a Java class that controls the plug-in's life cycle	Deselected
	Activator	Not available when you deselect Plug-in Options
	This plug-in will make contributions to the UI	Deselected
Rich Client Application	Would you like to create a rich client application?	No

Your settings should look like the one in Figure 20-2.



Figure 20-2 CSV ODA Driver plug-in content

- 5 In Templates, choose ODA Data Source Runtime Driver as shown in Figure 20-3. Choose Next. ODA Data Source Runtime Driver appears.

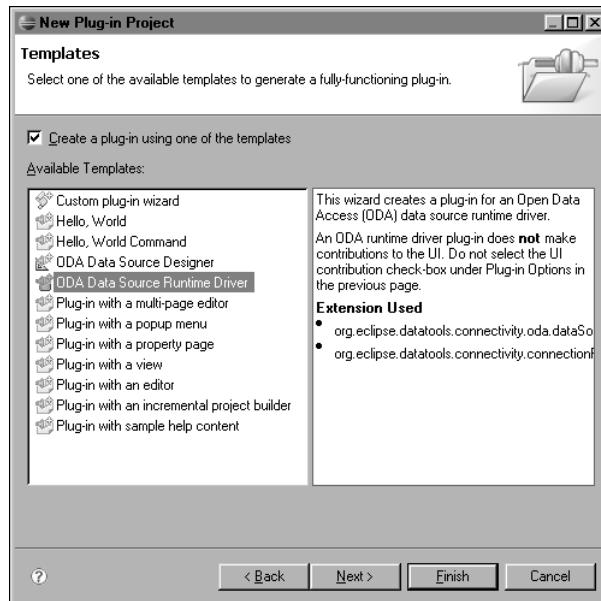


Figure 20-3 Specifying the ODA Data Source Runtime Driver template

- 6** In ODA Data Source Runtime Driver, specify values for the following options used to generate the ODA plug-in, as shown in Figure 20-4.

- 1 In Java Package Name, type:

```
org.eclipse.birt.report.data.oda.csv.impl
```

- 2 In ODA Data Source Element Id, type:

```
org.eclipse.birt.report.data.oda.csv
```

- 3 In Data Source Display Name, type:

CSV Data Source

- 4 In Number of Data Source Properties, type:

1

- 5 In Data Set Display Name, type:

CSV Data Set

- 6 In Number of Data Set Properties, type:

0

Choose Finish. The CSV ODA driver plug-in project appears in the Eclipse PDE workbench.

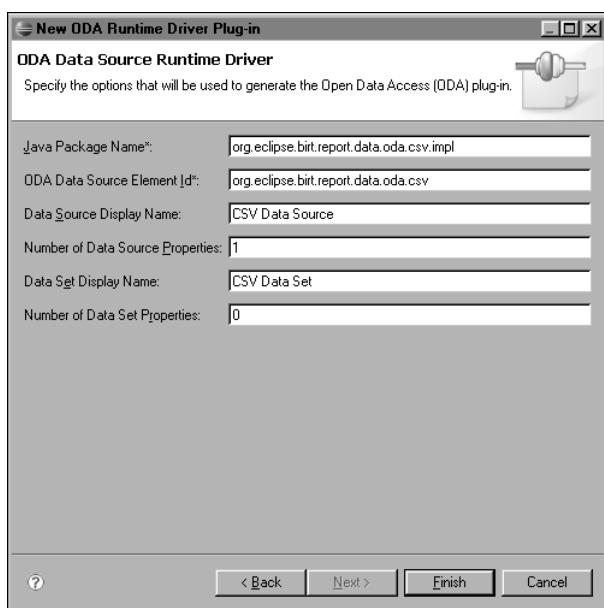


Figure 20-4 Specifying the ODA Data Source Runtime Driver options

The project created by the wizard looks like the one shown in Figure 20-5. As you see, the wizard already created all the plug-in files and the main functional Java classes. All you need to do is to customize the default

settings when needed, and add code to the stubs in the Java classes to implement the desired functionality.

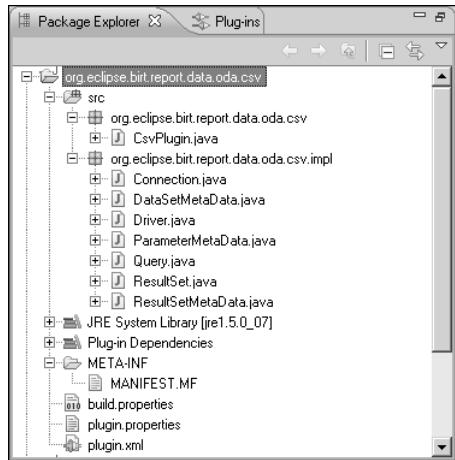


Figure 20-5 Project structure

- 7 Choose plugin.xml in Package Explorer and double-click to open PDE Manifest Editor. Using PDE Manifest Editor you can review and edit all the plug-in settings.

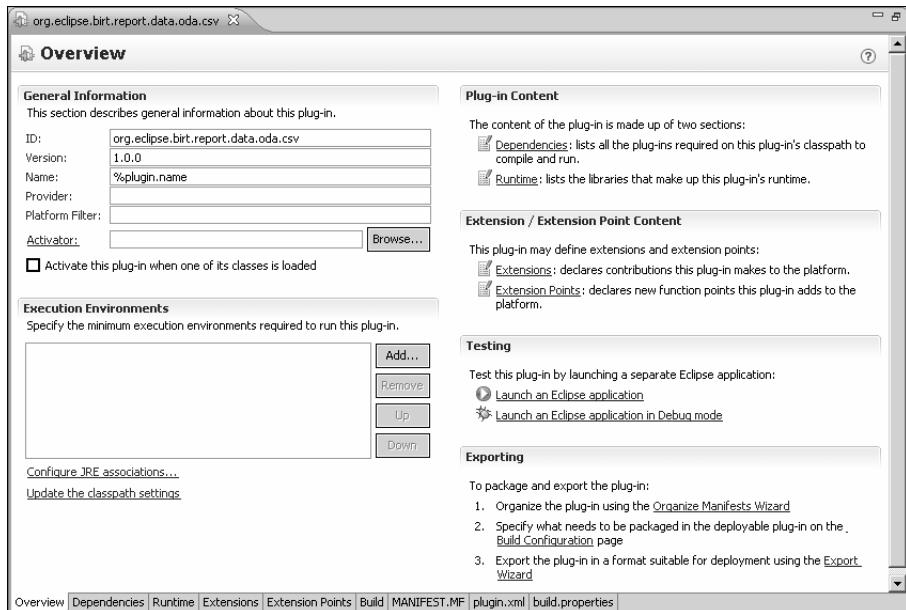


Figure 20-6 PDE Manifest Editor—Overview

Understanding the ODA data source extension points

In this step you review the extension points added by the wizard. Click the Extensions tab in PDE Manifest Editor to open the Extensions pane in the editor. The PDE Manifest Editor automatically adds the following two extension points:

- org.eclipse.datatools.connectivity.oda.dataSource
- org.eclipse.datatools.connectivity.oda.ConnectionProfile

Understanding dataSource extension point properties

The ODA data source extension point supports extending design-time and run-time data source access for an application. The extension must implement the ODA Java run-time interfaces defined in the org.eclipse.datatools.connectivity.oda plug-in. Figure 20-7 shows the ODA data source extension points used in the CSV ODA plug-in example.

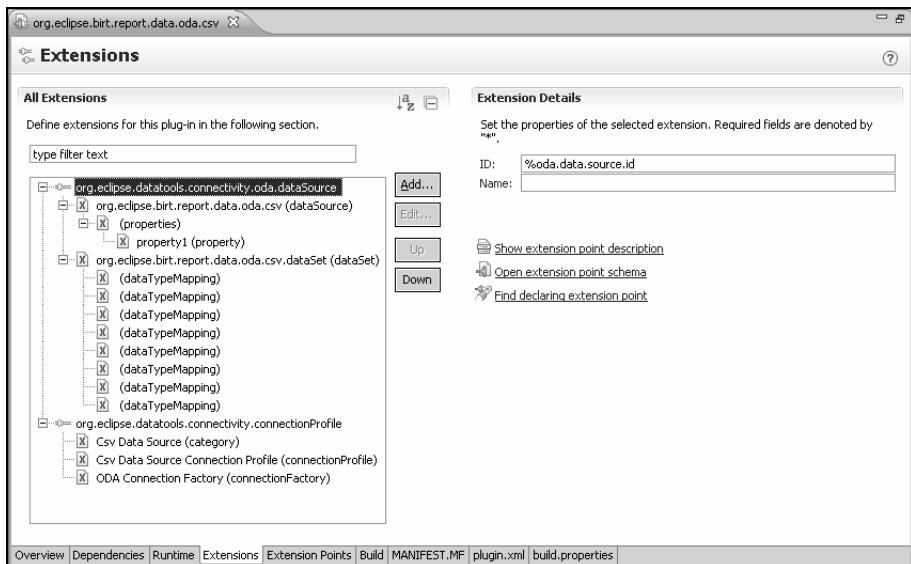


Figure 20-7 PDE Manifest Editor—Extensions

The extension point, org.eclipse.datatools.connectivity.oda.dataSource, specifies the following properties that identify the extension in the run-time environment:

- ID

Optional identifier of the extension instance. The wizard added a reference, %oda.data.source.id, to the extension point ID you specified in

the wizard. All the localized variables are defined in the plugin.properties file. The value for the ID shown in Listing 20-1 is as follows:

```
org.eclipse.birt.report.data.oda.csv
```

Listing 20-1 plugin.properties

```
#####
# Copyright (c) 2007 <>Your Company Name here>>
#
#####
# Plug-in Configuration
#
oda.data.source.id=org.eclipse.birt.report.data.oda.csv
#
#####
# NLS String
#####
#
plugin.name=Csv Data Source ODA Runtime Driver
data.source.name=Csv Data Source
data.set.name=Csv Data Set
connection.profile.name=Csv Data Source Connection Profile
```

- **Name**

Optional name of the extension instance. Fully qualified identifier of the extension

The extension point defines the extension elements and extension element details for the CSV ODA driver.

The dataSource extension element defines the ODA data source extension type to use at design time and run time. It contains the following properties:

- **id**

Fully qualified identifier of an ODA data source extension. The wizard references the externalized id with the notation %oda.data.source.id.

- **driverClass**

Java class that implements the org.eclipse.datatools.connectivity.oda.IDriver interface. This interface provides the entry point for the ODA run-time driver extension.

- **odaVersion**

Version of the ODA interfaces. Specify version 3.0 for an ODA driver developed for BIRT release 2.2.1.

- **defaultDisplayName**

Display name of the ODA data source extension. The value is externalized using the plugin.properties mechanism. The default display name is the extension id.

- **setThreadContextClassLoader**

Indicates whether the consumer of the ODA run-time extension plug-in must set the thread context class loader before calling an ODA interface method.

The OSGi class loader that loads the ODA run-time plug-in is not designed to load additional classes. To load additional classes, an ODA run-time plug-in must provide its own java.net.URLClassLoader object and switch the thread context class loader as required.

The dataSource element also specifies a property, containing the following extension element details:

- **name**

Unique name of a property group. Type HOME for a property name.

- **defaultDisplayName**

Default display name of a property group. The value can be localized using the plugin.properties mechanism.

Type “CSV File Full Path” for the default display name.

- **type**

Data type of the property. The default is String.

- **canInherit**

Flag indicating whether the property extension element can inherit properties.

- **defaultValue**

Default value of the property extension element.

- **isEncryptable**

Flag indicating whether the property is encrypted. Select false.

The dataSet extension element describes the following properties:

- **id**

Required identifier of the ODA data set extension.

- **defaultDisplayName**

Display name of the ODA data set extension. The value is localized using the plugin.properties file. The default display name is Csv Data Set, as you can see from Listing 20-1.

The `dataSet` element also specifies a complex data type, `dataTypeMapping`, which defines a sequence of data type mappings containing the following properties:

- `nativeDataTypeCode`

Integer value that must match one of the data type codes returned in the implementation for the ODA driver interface.

- `nativeDataType`

String value specifying the data source native data type.

- `odaScalarDataType`

ODA scalar data type that maps to the native type. Supported ODA data types include Date, Double, Integer, String, Time, Timestamp, Decimal, and Boolean.

The default supported types set by the wizard are shown in Table 20-3.

Table 20-3 Settings for `dataTypeMapping` elements

<code>nativeDataTypeCode</code>	<code>nativeDataType</code>	<code>odaScalarDataType</code>
1	String	String
4	Integer	Integer
8	Double	Double
2	BigDecimal	Decimal
91	Date	Date
92	Time	Time
93	TimeStamp	Timestamp
16	Boolean	Boolean

Understanding ConnectionProfile properties

This extension point supports creating database connections using connections profiles. The CSV ODA plug-in uses the settings created by the plug-in wizard.

Understanding the dependencies for the CSV ODA driver extension

In Figure 20-8, the Dependencies page shows a list of plug-ins that must be available on the classpath of the CSV ODA driver extension to compile and run.

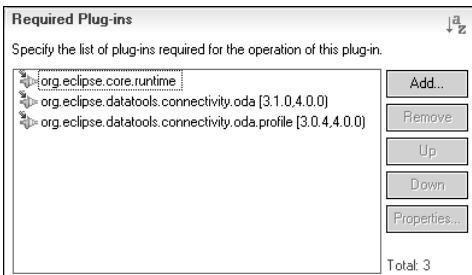


Figure 20-8 The Dependencies page

The ODA Runtime driver wizard adds the following dependencies to your plug-in:

- org.eclipse.datatools.connectivity.oda
- org.eclipse.datatools.connectivity.oda.profile

You can run Organize Manifest wizard at the end of your work to optimize the dependencies settings. The link to this wizard is on the Overview page of the Manifest Editor. Figure 20-9 shows an example of what settings to use when running the wizard.

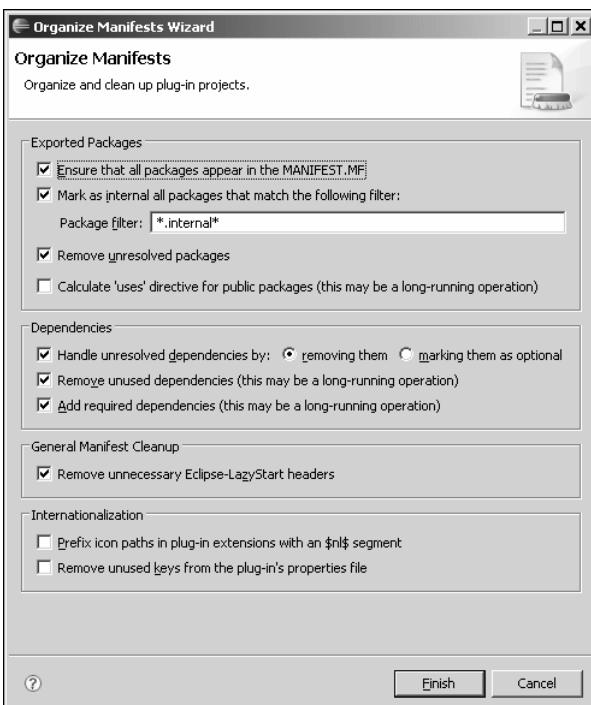


Figure 20-9 Organize Manifests Wizard

Understanding the sample CSV ODA driver extension

BIRT Data Engine supports the Eclipse DTP ODA framework. The DTP ODA framework supports creating an extension that can plug any external data source into BIRT Report Engine.

The DTP ODA API specifies the interfaces for a run-time driver. BIRT Data Engine uses the data source and data set definitions in a report design to access the ODA run-time driver to execute a query and retrieve data.

The DTP ODA interfaces are similar to JDBC interfaces with extensions that support retrieving data from non-RDBMS sources. An extended ODA driver can implement these interfaces to wrap the API for another data source, such as a CSV file, to retrieve a result set containing data rows.

The CSV ODA driver extension described in this chapter is a simplified example that illustrates how to create an ODA plug-in using the Eclipse PDE. The following section describes the code-based extensions a developer must make to complete the development of the CSV ODA driver extension after defining the plug-in framework in the Eclipse PDE.

Implementing the DTP ODA interfaces

The ODA plug-in, `org.eclipse.datatools.connectivity.oda`, defines the run-time interfaces used to retrieve data from a data source. The CSV ODA driver extension implements the following interfaces in the ODA plug-in:

- **IDriver**

The entry point to an ODA run-time driver. **IDriver** is the connection factory that generates the connection to an ODA run-time driver. An **IDriver** implementation provides the **IConnection** object used to establish a connection to a data source.

- **IConnection**

The interface that establishes a connection to a data source. An **IConnection** implementation opens and closes the connection, returns an **IQuery** object for a data set, and optionally commits or rolls back all changes made since the last commit or rollback operation.

- **IQuery**

The base interface for handling a query. The **IQuery** implementation prepares the query text, sets parameters and sorting specifications, executes the query, returns metadata for the result set, and closes the query.

- **IResultSet**

The interface used to access the result set retrieved by an IQuery object. An IResultSet implementation opens and closes a cursor that points to the current data row, and moves the cursor forward to the next row, until there are no more rows or MaxRows limit is reached. Accessor methods get the value for specified columns in the current row as specific data types. A query can retrieve one or more IResultSet instances.

- **IResultSetMetaData**

The interface that contains the metadata for an IResultSet object. An IResultSetMetaData implementation contains metadata describing each column in a result set, including the following information:

- Column count in the result set
- Display length
- Label
- Name
- Data type
- Precision
- Scale
- Permits null

- **IDataSetMetaData**

An interface that describes the features and capabilities of a data set type, including the following attributes:

- Indicates whether the data set type supports the following features:
 - Input, output, or named parameters
 - Named or multiple result sets
 - Multiple open result sets
- Provides the following information:
 - Version of the data source provider
 - Sort mode for columns, such as none, single, or multiple sort order
- Returns references to the following components:
 - Data source connection
 - Collection of objects in the data source provider catalog

- **IParameterMetaData**

An interface that provides information on the parameters defined in a prepared statement, including count, data type, precision, scale, or whether a parameter allows null.

Understanding the CSV ODA extension package

The package for the CSV ODA extension example, org.eclipse.birt.report.data.oda.csv, uses the following classes to implement the ODA plug-in interfaces:

- **Driver**

Implements the IDriver interface. Instantiates the connection object for the CSV ODA driver and sets up the log configuration and application context.
- **Connection**

Implements the IConnection interface. Opens and closes the connection to the CSV file and instantiates the IQuery object.
- **Query**

Implements the IQuery interface. Handles the processing that performs the following operations:

 - Sets up the java.io.File object, containing the file and path names
 - Fetches the data rows from the data file, using the internal class, CSVBufferReader
 - Trims the column data, removing extraneous characters such as commas and quotes
 - Prepares the result set metadata, containing the table and column names
- **ResultSet**

Implements the IResultSet interface. Handles the processing that transforms the String value for a column to the specified data type.
- **ResultSetMetaData**

Implements the IResultSetMetaData interface. Describes the metadata for each column in the result set.
- **DataSetMetaData**

Implements the IDataSetMetaData interface. Describes the features and capabilities of the data set.
- **Messages**

Defines the exception messages for the CSV ODA driver.
- **CommonConstant**

Defines the constants used in the package, such as the driver name, ODA version, query keywords, and delimiters.

Understanding Driver

The Driver class instantiates the connection object for the CSV ODA driver by calling the getConnection() method as shown in Listing 20-2.

Listing 20-2 The getConnection() method

```
public IConnection getConnection( String dataSourceType )
    throws OdaException
{
    return new Connection();
}
```

Understanding Connection

The Connection class opens and closes the connection to the CSV file and calls the newQuery() method to instantiate the Query object. Listing 20-3 shows the newQuery() method.

Listing 20-3 The newQuery() method

```
public IQuery newQuery( String dataSetType )
    throws OdaException
{
    if( !isOpen( ) )
        throw new OdaException(
            Messages.getString(
                "common_CONNECTION_HAS_NOT_OPENED" ) );
    return new Query( this.homeDir, this );
}
```

Understanding Query

The Query class constructor sets up a java.io.File object, containing the file and path names. The constructor allows the application to submit the home directory parameter, homeDir, as a file name as well as a path. Query() configures the data source property based on the value of the HOME property specified in the report design, as shown in Listing 20-4.

Listing 20-4 The Query class constructor

```
Query ( String homeDir, IConnection host )
    throws OdaException
{
    if ( homeDir == null || host == null )
        throw new OdaException(Messages.getString(
            "Common.ARGUMENT_CANNOT_BE_NULL"));
    File file = new File(homeDir);
    if (file.isDirectory( ) )
        this.homeDirectory = homeDir;
    else if (file.isFile( ) )
        this.homeDirectory = file.getParent( );
```

```
        this.connection = host;
    }
```

The Query class prepares and executes a query, then retrieves the data. Query implements the following additional methods:

- `prepare()` performs the following operations:
 - Generates query and column information by calling `splitQueryText()`
 - Validates the connection by calling `validateOpenConnection()`
 - Formats the query String, eliminating redundant spaces and converting all keywords to uppercase, by calling `formatQueryText()`
 - Validates the query by calling `validateQueryText()`
 - Prepares the metadata required for the execution of the query and retrieval of the query results by calling `prepareMetaData()`

Listing 20-5 shows the `prepare()` method.

Listing 20-5 The `prepare()` method

```
public void prepare( String queryText )
    throws OdaException
{
    if ( queryText != null )
    {
        String query = splitQueryText(queryText)[0] ;
        String colInfo = splitQueryText( queryText )[1];
        validateOpenConnection( );
        String formattedQuery = formatQueryText( query );
        validateQueryText( formattedQuery );
        prepareMetaData( formattedQuery, colInfo );
    }
    else
        throw new OdaException( Messages.getString(
            "common_NULL_QUERY_TEXT" ) );
}
```

- `prepareMetaData()` acquires the following metadata:
 - Table name
 - Actual column names read from data file
 - Query column names
 - Query data types

`prepareMetaData()` then instantiates and configures the `ResultSetMetaData` object. Listing 20-6 shows the `prepareMetaData()` method.

Listing 20-6 The prepareMetaData() method

```
private void prepareMetaData( String query,
    String savedSelectedColInfo )
throws OdaException
{
    String[ ] queryFragments =
        parsePreparedQueryText( query );
    String tableName = queryFragments[2];
    String[ ] allColumnNames =
        discoverActualColumnMetaData( tableName,NAME_LITERAL );
    String[ ] allColumnTypes =
        createTempColumnTypes( allColumnNames.length );
    String[ ] queryColumnNames = null;
    String[ ] queryColumnTypes = null;
    String[ ] queryColumnLabels = null;

    queryColumnNames = allColumnNames;
    queryColumnTypes = allColumnTypes;
    queryColumnLabels = allColumnNames;

    this.resultSetMetaData =
        new ResultSetMetaData( queryColumnNames,
            queryColumnTypes, queryColumnLabels );
    this.currentTableName = tableName;
}
```

- executeQuery() performs the following operations:
 - Fetches the data from the file to a Vector object
 - Transfers the data from the Vector to a two-dimensional String array
 - Returns the data rows and metadata in a single ResultSet object

Listing 20-7 shows the executeQuery() method.

Listing 20-7 The executeQuery() method

```
public IResultSet executeQuery( ) throws OdaException
{
    Vector v = fetchQueriedDataFromFileToVector( );
    String[][] rowSet =
        copyDataFromVectorToTwoDimensionArray( v );
    return new ResultSet( rowSet, this.resultSetMetaData );
}
```

- The internal class, CSVBufferReader, fetches the data rows from the data file. Listing 20-8 shows the readLine() method.

Listing 20-8 The readLine() method

```
public String readLine( ) throws IOException
{
    if ( isLastCharBuff( ) && needRefillCharBuff( ) )
```

```

        return null;
    if ( needRefillCharBuff( ) )
    {
        charBuffer = newACharBuff( );
        int close = reader.read( charBuffer );
        if ( close == -1 )
            return null;
        if ( close != CHARBUFSIZE )
            this.eofInPosition = close;
        this.startingPosition = 0;
    }
    String candidate = "";
    int stopIn = CHARBUFSIZE;
    if ( isLastCharBuff( ) )
    {
        stopIn = this.eofInPosition;
    }
    for ( int i = this.startingPosition; i < stopIn; i++ )
    {
        if ( this.charBuffer[i] == '\n' )
        {
            return readALine( candidate, stopIn, i );
        }
    }
    if ( isLastCharBuff( ) )
    {
        return readLastLine( candidate );
    }
    return readExtraContentOfALine( candidate );
}

```

Understanding ResultSet

The ResultSet class performs the following operations:

- Provides the cursor processing that fetches forward into the buffered result set rows
- Transforms the String value for a column to the specified data type

ResultSet implements the following methods:

- ResultSet(), the constructor, sets up a two-dimensional array that contains the table data and metadata, as shown in Listing 20-9.

Listing 20-9 The ResultSet() constructor

```

ResultSet( String[][] sData, IResultSetMetaData rsmd )
{
    this.sourceData = sData;
    this.resultSetMetaData = rsmd;
}

```

- `getRow()` returns the cursor, indicating the position of the row in the result set, as shown in Listing 20-10.

Listing 20-10 The `getRow()` method

```
public int getRow() throws OdaException
{
    validateCursorState();
    return this.cursor;
}
```

- `next()` increments the cursor to point to the next row, as shown in Listing 20-11.

Listing 20-11 The `next()` method

```
public boolean next() throws OdaException
{
    if ( (this.maxRows <= 0? false:cursor >=
        this.maxRows - 1) || cursor >=
        this.sourceData.length - 1 )
    {
        cursor = CURSOR_INITIAL_VALUE;
        return false;
    }
    cursor++;
    return true;
}
```

- `getString()` returns the value for a column in the row at the column position specified in the result set, as shown in Listing 20-12.

Listing 20-12 The `getString()` method

```
public String getString( int index )
    throws OdaException
{
    validateCursorState();
    String result = sourceData[cursor][index - 1];
    if( result.length( ) == 0 )
        result = null;
    this.wasNull = result == null ? true : false;
    return result;
}
```

Understanding ResultSetMetaData

The `ResultSetMetaData` class describes the metadata for a column in the result set, including the following information:

- Column count in the result set
- Display length

- Label
- Name
- Data type
- Precision
- Scale
- Permits null

`getColumnName()` returns the column name for a column at the row, column position specified in the result set, as shown in Listing 20-13.

Listing 20-13 The `getColumnName()` method

```
public String getColumnName( int index ) throws OdaException
{
    if ( index > getColumnCount( ) || index < 1 )
        throw new OdaException( Messages.getString(
            "resultSetMetaData_INVALID_COLUMN_INDEX" ) + index );
    return this.columnNames[index - 1].trim( );
}
```

Understanding DataSetMetaData

The `DataSetMetaData` class describes the features and capabilities of the data set, including the following:

- Indicating whether the data set supports multiple result sets
- Providing information about the sort mode for columns
- Returning a reference to the data source connection

`getConnection()` returns a reference to a data source connection, as shown in Listing 20-14.

Listing 20-14 The `getConnection()` method

```
public IConnection getConnection( ) throws OdaException
{
    return m_connection;
}
```

Understanding Messages

The `Messages` class defines the exception messages for the CSV ODA driver. `getString()` returns a message from the resource bundle using the key value, as shown in Listing 20-15.

Listing 20-15 The `getString()` method

```
public static String getString(String key) {
```

```

try {
    return RESOURCE_BUNDLE.getString(key);
}
catch (MissingResourceException e) {
    return '!' + key + '!';
}
}

```

Understanding CommonConstants

The CommonConstants class defines the constants used in the package, such as the driver name, ODA version, query keywords, and delimiters. Listing 20-16 shows these definitions.

Listing 20-16 The CommonConstants class

```

final class CommonConstants
{
    public static final String DELIMITER_COMMA = ",";
    public static final String DELIMITER_SPACE = " ";
    public static final String DELIMITER_DOUBLEQUOTE = "\"";
    public static final String KEYWORD_SELECT = "SELECT";
    public static final String KEYWORD_FROM = "FROM";
    public static final String KEYWORD_AS = "AS";
    public static final String KEYWORD_ASTERISK = "*";
    public static final String DRIVER_NAME =
        "ODA CSV FILE DRIVER";
    public static final int MaxConnections = 0;
    public static final int MaxStatements = 0;
    public static final String CONN_HOME_DIR_PROP = "HOME";
    public static final String CONN_DEFAULT_CHARSET = "UTF-8";
    public static final String PRODUCT_VERSION = "3.0";
}

```

Developing the CSV ODA UI extension

The data source extension point, org.eclipse.datatools.connectivity.oda.design.ui.dataSource, supports adding a new data source to a user interface, such as BIRT Report Designer. For each data source, the extension implements the following optional components:

- A wizard for creating the data source
- A set of pages for editing the data source
- The list of data sets that the data source supports

For each data set, the extension implements the following optional components:

- A wizard for creating the data set

- A set of pages for editing the data set

The data source editor page must implement the extension point, org.eclipse.ui.propertyPages, by extending the abstract class, org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSourceEditorPage. The data set editor page must implement the extension point, org.eclipse.ui.propertyPages, by extending the abstract class, org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSourceEditorPage. The ODA data source and data set UI extensions extend these base classes to create customized property pages with page control and other behavior.

This section describes how to implement a BIRT ODA UI plug-in, using the CSV ODA driver plug-in as an example. To create an ODA driver plug-in, perform the following tasks:

- Create the CSV ODA UI plug-in project.
- Define the dependencies.
- Specify the run-time archive.
- Declare the ODA UI extension points.

Creating the CSV ODA UI plug-in project

You can create the CSV ODA UI plug-in project, org.eclipse.birt.report.data.oda.csv.ui, using the Eclipse PDE. The Eclipse PDE provides a wizard to assist you in setting up a plug-in project and creating the framework for a plug-in extension.

The New Plug-in Project wizard simplifies the process of specifying a plug-in project, automatically adds the required extension points, and sets the dependencies. The wizard also generates the plug-in manifest file, plugin.xml, and optionally, the Java plug-in run-time class.

The wizard creates all the implementation Java classes. It also puts TODO tags in the generated source code for your custom code along with guiding comments.

After using the wizard to create the plug-in, you can review the settings, make adjustments to the settings, and add the code to the Java class stubs to implement the desired functionality.

The following section describes how to create the plug-in project using the New Plug-in Project wizard.

How to create the CSV ODA UI plug-in project

- 1 From the Eclipse menu, choose File->New->Project. New Project appears.
- 2 On New Project—Select a wizard, open Business Intelligence and Reporting Tools, and select ODA Designer Plug-in Project as shown in Figure 20-10. Choose Next. New Plug-in Project appears.

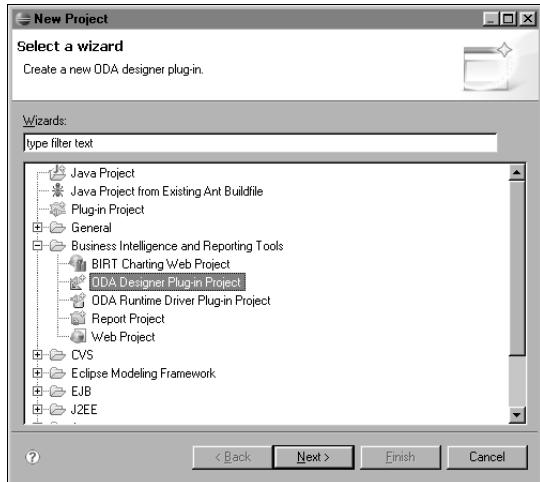


Figure 20-10 Specifying the ODA Designer Plug-in Project

- 3 In Plug-in Project, modify the settings as shown in Table 20-4. Choose Next. Plug-in Content appears.

Table 20-4 Settings for Plug-in Project options

Section	Option	Value
Plug-in Project	Project name	org.eclipse.birt.report .data.oda.csv.ui
	Use default location	Selected
	Location	Not available when you select Use default location
Project Settings	Create a Java project	Selected
	Source folder	src
	Output folder	bin
Target Platform	Eclipse version	3.3
	OSGi framework	Deselected

- 4 In Plug-in Content, modify the settings, where needed, as shown in Table 20-5. Choose Finish.

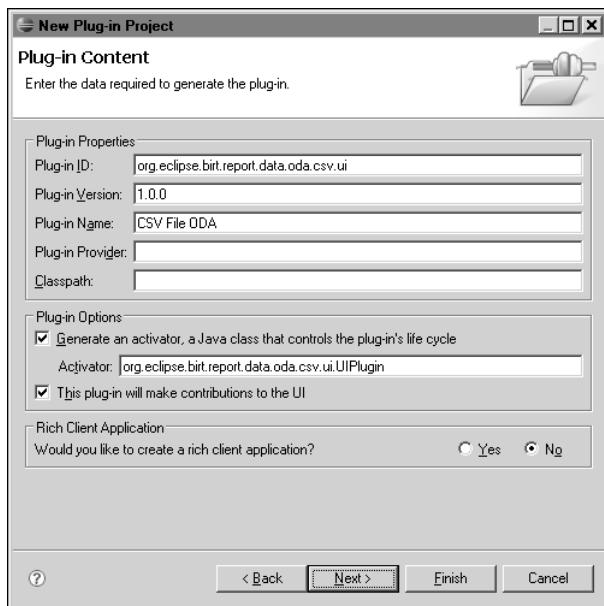
Table 20-5 Settings for Plug-in Content options

Section	Option	Value
Plug-in Properties	Plug-in ID	org.eclipse.birt.report .data.oda.csv.ui

Table 20-5 Settings for Plug-in Content options (*continued*)

Section	Option	Value
	Plug-in Version	1.0.0
	Plug-in Name	CSV File ODA
	Plug-in Provider	yourCompany.com or leave blank
	Classpath	
Plug-in Options	Generate an activator, a Java class that controls the plug-in's life cycle	Selected
	Activator	org.eclipse.birt.report.data.oda.csv.ui.UiPlugin
	This plug-in will make contributions to the UI	Selected
Rich Client Application	Would you like to create a rich client application?	No

The Plug-in Content window should look like the one in Figure 20-11.

**Figure 20-11** Plug-in Content

- 5 In Templates, choose ODA Data Source Designer as shown in Figure 20-12. Choose Next. ODA Data Source Designer appears.

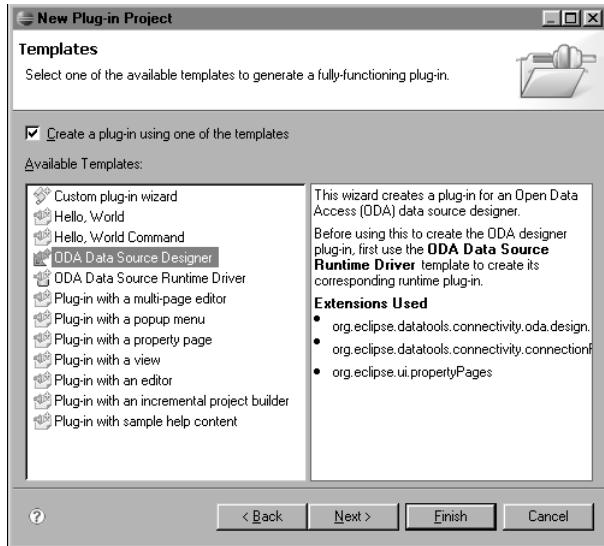


Figure 20-12 Specifying the ODA Data Source Designer template

The wizard warns you that it is required to create ODA Data Source Runtime Driver first, and then create the designer user interface. It lists the extension points that are automatically included in the default implementation.

- `org.eclipse.datatools.connectivity.oda.design.ui.dataSource`
- `org.eclipse.datatools.connectivity.connectionProfile`
- `org.eclipse.ui.propertyPages`

- 6 In ODA Data Source Designer, specify new values for the following options used to generate the ODA plug-in, as shown in Figure 20-13.

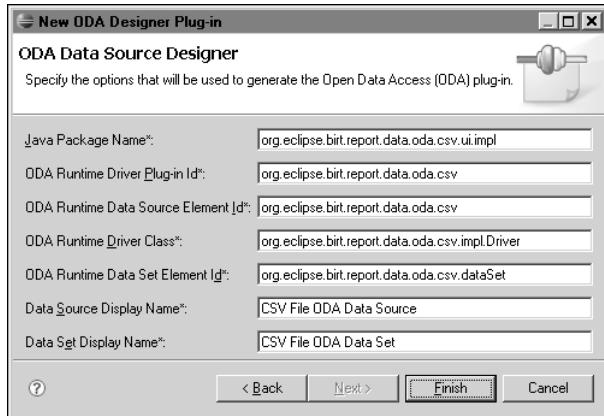


Figure 20-13 Specifying the ODA Data Source Designer options

1 In Java Package Name, type:

`org.eclipse.birt.report.data.oda.csv.ui.impl`

2 In ODA Runtime Driver Plug-in Id, type:

`org.eclipse.birt.report.data.oda.csv`

3 In ODA Runtime Data Source Element Id, type:

`org.eclipse.birt.report.data.oda.csv`

4 In ODA Runtime Driver Class, type:

`org.eclipse.birt.report.data.oda.csv.impl.Driver`

5 In ODA Runtime Data Set Element Id, type:

`org.eclipse.birt.report.data.oda.csv.dataSet`

6 In Data Source Display Name, type:

`CSV File ODA Data Source`

7 In Data Set Display Name, type:

`CSV File ODA Data Set`

Choose Finish.

Double-click plugin.xml to open it in PDE Manifest Editor. The plug-in Overview page looks like the one shown on Figure 20-14.

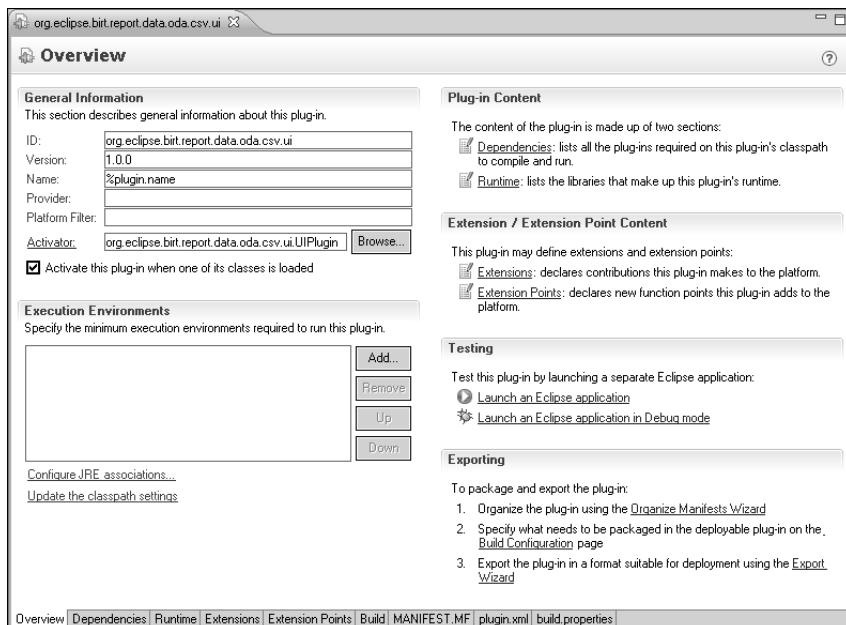


Figure 20-14 CSV ODA UI plug-in

Understanding the ODA data source UI extension points

In this next step, you review the extension points added by the wizard to implement the CSV ODA UI extension and add the extension element details. Figure 20-15 shows the list of CSV ODA UI extension points.

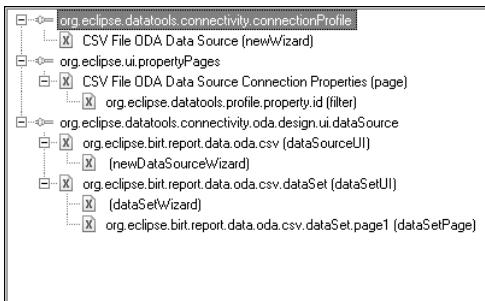


Figure 20-15 CSV ODA UI extension points

The CSV ODA UI plug-in extends the functionality defined by the following extension points:

- org.eclipse.datatools.connectivity.connectionProfile
 - Provides support for adding a connection profile
- org.eclipse.ui.propertyPages
 - Adds a property page that displays the properties of an object in a dialog box
- org.eclipse.datatools.connectivity.oda.design.ui.dataSource
 - Extends the ODA Designer UI framework to support creating a dialog page that allows a user to specify an ODA data source and a related data set

The extension points specify the following properties that identify the extensions in the run-time environment:

- ID
 - Optional identifier of the extension instance
- Name
 - Optional name of the extension instance

Understanding the ConnectionProfile extension point

The ConnectionProfile extension point specifies the following extension elements:

- category

- Identifies the category. Supports grouping of connection profile types, such as related database connection profiles.
- **connectionProfile**
Defines a connection profile type. Specifies properties such as id, category, display name, configuration type, icon, connection factory, and property persistence.
- **connectionFactory**
Defines a connection factory that creates a connection to a server using the properties stored in a connection profile.
- **newWizard**
Defines a wizard that creates a connection profile.

Understanding the **propertyPages** extension point

The **propertyPages** extension point specifies the following extension elements:

- **page**
Defines a property page. Specifies properties such as id, display name, category, icon, object class, filter, and category. The id and the display name are localized in the plugin.properties file, as shown in Listing 20-17.

Listing 20-17 plugin.properties

```
#####
# Copyright (c) 2007 <>Your Company Name here><
#
#####
# Plug-in Configuration
#
oda.data.source.id=org.eclipse.birt.report.data.oda.csv
#
#####
# NLS strings
#
plugin.name=CSV File ODA
data.source.name=CSV File Data Source
connection.profile.name=CSV File Data Source Connection
    Profile
newwizard.name=CSV File Data Source
newwizard.description>Create a CSV File Data Source
    connection profile
wizard.window.title>New CSV File Data Source Profile
wizard.data.source.page.title=CSV File Data Source
profile.property.page.name=CSV File Data Source Connection
    Properties
```

```
wizard.data.set.window.title=New CSV File Data Set  
wizard.data.set.page.title=Query
```

ODA UI framework provides a default implementation that creates a text control for each property value, and that is why there is no need for you to provide your own custom implementation. By default the wizard sets the page.class in the org.eclipse.ui.propertyPages extension point:

```
page.class=org.eclipse.datatools.connectivity.oda.design.ui.  
pages.impl.DefaultDataSourcePropertyPage
```

- filter

Specifies an action filter that evaluates the attributes of each object in a current selection. If an object has the specified attribute state, a match occurs. Each object must implement the org.eclipse.ui.IActionFilter interface.

Understanding the dataSource extension point

The dataSource extension point specifies the following extension elements:

- dataSourceUI

Adds UI support for specifying an extended data source.

- dataSetUI

The dataSetUI extension element defines the following extension elements and details:

- id

Fully qualified name of the data set, such as org.eclipse.birt.report.data.oda.csv.dataSet. This name must be the same as the name for the ODA extension driver data set.

- dataSetWizard

Wizard class that allows a report developer to specify a data set in the BIRT Report Designer UI. This class must use or extend org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSetWizard. The window title, %wizard.data.set.window.title, is localized in the plug-in properties file.

- dataSetPage

Specifies an editor page to add to the editor dialog for a data set. The data set UI adds editor pages to a dialog in the order the pages are defined. This class must use or extend org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSetWizardPage. The implementation code is in org.eclipse.birt.report.data.oda.csv.ui.impl.CustomDataSetWizardPage. The wizard provides a default implementation code.

The page display name, %wizard.data.set.page.title, is externalized in the plugin.properties file.

Understanding the sample CSV ODA UI extension

The CSV ODA UI extension described in this chapter illustrates how to create an ODA UI plug-in using the Eclipse PDE. The following section describes the code-based extensions a developer must make to complete the development of the CSV ODA UI extension, after defining the plug-in framework in the Eclipse PDE.

The CSV ODA UI plug-in contains the following packages:

- org.eclipse.birt.report.data.oda.csv.ui

Contains the following files:

- UiPlugin class is automatically generated by the PDE Manifest Editor when you create the plug-in project.
- Messages class and the properties file, messages.properties, generate the messages displayed in the UI. The localized versions for these messages are in files, using the following naming syntax:

`messages_<locale>.msg`

- org.eclipse.birt.report.data.oda.csv.ui.impl

Contains the CustomDataSetWizardPage class is an automatically generated implementation of an ODA data set designer page, which allows a user to create or edit an ODA data set.

- org.eclipse.birt.report.data.oda.csv.ui.wizards

The wizards package contains the classes that create the user interface pages used to choose a data source and data set in BIRT Report Designer.

Implementing the ODA data source and data set wizards

In BIRT release 2.1, BIRT Report Designer adopted the Eclipse Data Tools Platform (DTP) ODA design-time framework. BIRT release 2.2.1 further extends the DTP ODA design-time framework by adding new wizards to automatically generate customizable implementations. The DTP ODA framework defines two of the three extension points used in the CSV ODA UI plug-in:

- Connection profile

Defined in `org.eclipse.datatools.connectivity.connectionProfile`

- Data source and data set wizards

Defined in org.eclipse.datatools.connectivity.oda.design.ui.dataSource

The CSV ODA UI plug-in also must implement the extension point for property pages defined in org.eclipse.ui.propertyPages.

The CSV ODA UI plug-in uses the following abstract base classes in the org.eclipse.datatools.connectivity.oda.design.ui.wizards package to create the wizards that specify the data source and data set pages. An ODA UI plug-in must extend these classes to provide the wizard pages with page control and related behavior:

- **DataSourceEditorPage**

Provides the framework for implementing an ODA data source property page

- **DataSourceWizardPage**

Provides the framework for implementing an ODA data source wizard page

- **DataSetWizardPage**

Provides the framework for implementing an ODA data set wizard page

Understanding the org.eclipse.birt.report.data.oda.csv.ui.impl package

The org.eclipse.birt.report.data.oda.csv.ui.impl package in the CSV ODA UI extension example implements the CustomDataSetWizardPage class. This automatically generated ODA data set designer page allows a user to create or edit an ODA data set.

This customizable page provides a simple Query Text control for user input. The page extends org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSetWizardPage in the DTP ODA design-time framework to allow updating an ODA data set design instance using query metadata.

Understanding the org.eclipse.birt.report.data.oda.csv.ui.wizards package

The org.eclipse.birt.report.data.oda.csv.ui.wizards package in the CSV ODA UI extension example implements the following classes:

- **Constants**

Defines the constants for the data source connection properties defined in the run-time drive implementation, org.eclipse.birt.report.data.oda.csv.

- **CSVFilePropertyPage**

Extends DataSourceEditorPage. This class creates and initializes the editor controls for the property page used to specify the ODA data source. The

class updates the connection profile properties with the values collected from the page.

- CSVFileSelectionPageHelper

Specifies the page layout and sets up the control that listens for user input and verifies the location of the CSV data source file.

- CSVFileSelectionWizardPage

Extends DataSourceWizardPage. This class creates and initializes the controls for the data source wizard page. The class sets the select file message and collects the property values.

- FileSelectionWizardPage

Extends DataSetWizardPage. This class creates and initializes the controls for the data set wizard page and specifies the page layout. The class connects to the data source, executes a query, retrieves the metadata and result set, and updates the date-set design.

Understanding Constants

The Constants class defines the following variables for the data source connection properties defined in org.eclipse.birt.report.data.oda.csv:

- ODAHOME specifies the CSV ODA file path constant, HOME
- ODA_DEFAULT_CHARSET specifies the default character set as 8-bit Unicode Transformation Format (UTF-8)
- DEFAULT_MAX_ROWS sets the default maximum number of rows that can be retrieved from the data source

Listing 20-18 shows the code for the Constants class.

Listing 20-18 The Constants class

```
public class Constants {  
    public static String ODAHOME="HOME";  
    public static String ODA_DEFAULT_CHARSET = "UTF-8";  
    public static int DEFAULT_MAX_ROWS = 1000;  
}
```

Understanding CSVFilePropertyPage

CSVFilePropertyPage extends the DataSourceEditorPage class, implementing the following methods to provide page editing functionality for the CSV ODA data source property page:

- createAndInitCustomControl() method performs the following tasks:
 - Instantiates a CSVFileSelectionPageHelper object

- Specifies the page layout and sets up the editing control by calling CSVFileSelectionPageHelper.createCustomControl() and initCustomControl() methods

Listing 20-19 shows the code for the createAndInitCustomControl() method.

Listing 20-19 The createAndInitCustomControl() method

```
protected void createAndInitCustomControl
    ( Composite parent, Properties profileProps )
{
    if( m_pageHelper == null )
        m_pageHelper =
            new CSVFileSelectionPageHelper( this );
    m_pageHelper.createCustomControl( parent );
    m_pageHelper.initCustomControl( profileProps );
    if( ! isSessionEditable( ) )
        getControl( ).setEnabled( false );
}
```

- collectCustomProperties() updates the connection profile properties with the values collected from the page by calling CSVFileSelectionPageHelper.collectCustomProperties() method, as shown in Listing 20-20.

Listing 20-20 The collectCustomProperties() method

```
public Properties collectCustomProperties
    ( Properties profileProps )
{
    if( m_pageHelper == null )
        return profileProps;
    return m_pageHelper.collectCustomProperties
        ( profileProps );
}
```

Understanding CSVFileSelectionPageHelper

CSVFileSelectionPageHelper provides auxiliary processing for the CSVFilePropertyPage and CSVFileSelectionWizardPage classes.

CSVFileSelectionPageHelper implements the following methods:

- createCustomControl() performs the following tasks:
 - Sets up the composite page layout
 - Calls the setupFileLocation() method that sets up a control to listen for user input and verify the location of the CSV data source file

Listing 20-21 shows the code for the createCustomControl() method.

Listing 20-21 The createCustomControl() method

```
void createCustomControl( Composite parent )
{
    Composite content = new Composite( parent, SWT.NULL );
    GridLayout layout = new GridLayout( 2, false );
    content.setLayout(layout);
    setupFileLocation( content );
}
```

- setupFileLocation() performs the following tasks:
 - Sets up the label and the grid data object in the page layout
 - Sets up the control that listens for user input and verifies the location of the CSV data source file

Listing 20-22 shows the code for the setupFileLocation() method.

Listing 20-22 The setupFileLocation() method

```
private void setupFileLocation( Composite composite )
{
    Label label = new Label( composite, SWT.NONE );
    label.setText( Messages.getString
        ( "label.selectFile" ) );
    GridData data = new GridData( GridData.FILL_HORIZONTAL );
    fileName = new Text( composite, SWT.BORDER );
    fileName.setLayoutData( data );
    setPageComplete( false );
    fileName.addModifyListener(
        new ModifyListener( )
    {
        public void modifyText( ModifyEvent e )
        {
            verifyFileLocation();
        }
    } );
}
```

- collectCustomProperties() sets the data source directory property in the connection profile, as shown in Listing 20-23.

Listing 20-23 The collectCustomProperties() method

```
Properties collectCustomProperties( Properties props )
{
    if( props == null )
        props = new Properties();
    props.setProperty( CommonConstants.CONN_HOME_DIR_PROP,
        getFolderLocation() );
    return props;
}
```

- `initCustomControl()` initializes the data source wizard control to the location of the data source file, as shown in Listing 20-24.

Listing 20-24 The `initCustomControl()` method

```
void initCustomControl( Properties profileProps )
{
    if( profileProps == null || profileProps.isEmpty( ) || 
        fileName == null )
        return;
    String folderPath = profileProps.getProperty(
        CommonConstants.CONN_HOME_DIR_PROP );
    if( folderPath == null )
        folderPath = EMPTY_STRING;
    fileName.setText( folderPath );
    verifyFileLocation( );
}
```

Understanding CSVFileSelectionWizardPage

The `CSVFileSelectionWizardPage` class extends the `DataSourceWizardPage` class, implementing the following methods to provide the functionality for the CSV ODA data source wizard page:

- The `createPageCustomControl()` method performs the following tasks:
 - Instantiates a `CSVFileSelectionPageHelper` object
 - Specifies the page layout and sets up the wizard page control by calling `CSVFileSelectionPageHelper.createCustomControl()` method
 - Calls `CSVFileSelectionPageHelper.initCustomControl()` to initialize the control to the location of the data source file

Listing 20-25 shows the code for the `createPageCustomControl()` method.

Listing 20-25 The `createPageCustomControl()` method

```
public void createPageCustomControl( Composite parent )
{
    if( m_pageHelper == null )
        m_pageHelper =
            new CSVFileSelectionPageHelper( this );
    m_pageHelper.createCustomControl( parent );
    m_pageHelper.initCustomControl( m_csvFileProperties );
}
```

- The `collectCustomProperties()` method instantiates a `Properties` object to contain the CSV data source properties information, as shown in Listing 20-26.

Listing 20-26 The `collectCustomProperties()` method

```
public Properties collectCustomProperties( )
```

```

    {
        if( m_pageHelper != null )
            return m_pageHelper.collectCustomProperties(
                m_csvFileProperties );
        return ( m_csvFileProperties != null ) ?
            m_csvFileProperties : new Properties( );
    }

```

Understanding FileSelectionWizardPage

The FileSelectionWizardPage class extends the DataSetWizardPage class, implementing the following methods to provide the functionality for the CSV ODA data set wizard page:

- The createPageControl() method performs the following tasks:
 - Specifies the page layout and sets up the wizard page control
 - Gets the data source properties
 - Calls populateAvailableList() to update the data set design

Listing 20-27 shows the code for the createPageControl() method.

Listing 20-27 The createPageControl() method

```

private Control createPageControl( Composite parent )
{
    Composite composite = new Composite( parent, SWT.NULL );
    FormLayout layout = new FormLayout();
    composite.setLayout( layout );
    FormData data = new FormData();
    data.left = new FormAttachment( 0, 5 );
    data.top = new FormAttachment( 0, 5 );
    fileName = new Text( composite, SWT.BORDER );
    fileName.setLayoutData( data );
    Properties dataSourceProps =
        getInitializationDesign().getDataSourceDesign()
            .getPublicProperties();
    fileName.setText( ( String )dataSourceProps.getProperty( Constants.ODAHOME ) );
    data = new FormData();
    data.top = new FormAttachment(
        fileName, 10, SWT.BOTTOM );
    data.left = new FormAttachment( 0, 5 );
    data.right = new FormAttachment( 47, -5 );
    data.bottom = new FormAttachment( 100, -5 );
    data.width = DEFAULT_WIDTH;
    data.height = DEFAULT_HEIGHT;
    m_availableList = new List( composite,
        SWT.MULTI | SWT.BORDER |
        SWT.H_SCROLL | SWT.V_SCROLL );
    m_availableList.setLayoutData( data );
    m_selectedFile =

```

```

        new File(( String )(  

            dataSourceProps.getProperty( Constants.ODAHOME ) ));  

        populateAvailableList( );  

        return composite;  

    }
}

```

- `getQuery()` method builds the query for the data set by performing the following tasks:
 - Gets the table name from the file object
 - Appends the table name to a query that selects all the columns using a wildcard
 - Appends the column list then the table name to a query that selects specific columns
 - Returns the query text

Listing 20-28 shows the code for the `getQuery()` method.

Listing 20-28 The `getQuery()` method

```

private String getQuery( )  

{  

    String tableName = null;  

    StringBuffer buf = new StringBuffer( );  

    File file = m_selectedFile;  

    if(file != null)  

    {  

        tableName = file.getName( );  

    }  

    if(tableName != null)  

    {  

        if(m_availableList.getItemCount( ) == 0)  

        {  

            buf.append("select * from ").append(tableName);  

        }  

        else  

        {  

            buf.append("select ");  

            String[ ] columns = m_availableList.getItems( );  

            for(int n = 0; n < columns.length; n++)  

            {  

                buf.append(columns[n]);  

                if(n < columns.length - 1)  

                {  

                    buf.append(", ");  

                }  

            }  

            buf.append(" from ").append(tableName);  

        }  

    }  

    return buf.toString( );
}

```

- `getQueryColumnNames()` method performs the following tasks:
 - Instantiates the CSVFileDriver
 - Prepares the query and gets the results set metadata using the CSV ODA run-time driver and data source connection properties settings
 - Gets the column count
 - Iterates through the metadata results to get the column names and return the results

Listing 20-29 shows the code for the `getQueryColumnNames()` method.

Listing 20-29 The `getQueryColumnNames()` method

```

private String[ ] getQueryColumnNames(
    String queryText, File file )
{
    IDriver ffDriver = new CSVFileDriver( );
    IConnection conn = null;
    try
    {
        conn = ffDriver.getConnection( null );
        IResultSetMetaData metaData =
            getResultSetMetaData( queryText, file, conn );
        int columnCount = metaData.getColumnCount( );
        if( columnCount == 0 )
            return null;
        String[ ] result = new String[columnCount];
        for( int i = 0; i < columnCount; i++ )
            result[i] = metaData.getColumnName( i + 1 );
        return result;
    }
    catch( OdaException e )
    {
        setMessage( e.getLocalizedMessage( ), ERROR );
        disableAll( );
        return null;
    }
    finally
    {
        closeConnection( conn );
    }
}

```

- `getResultSetMetaData()` method performs the following tasks:
 - Sets up the Properties object with the location of the data source file
 - Opens the connection to the data source
 - Sets up a Query object and prepares the query text
 - Executes the query
 - Returns the metadata

Listing 20-30 shows the code for the getResultSetMetaData() method.

Listing 20-30 The getResultSetMetaData() method

```
private IResultSetMetaData getResultSetMetaData(
    String queryText, File file, IConnection conn )
throws OdaException
{
    java.util.Properties prop = new java.util.Properties( );
    prop.put( CommonConstants.CONN_HOME_DIR_PROP,
        file.getParent( ) );
    conn.open( prop );
    IQuery query = conn.newQuery( null );
    query.setMaxRows( 1 );
    query.prepare( queryText );
    query.executeQuery( );
    return query.getMetaData( );
}
```

- setResultSetMetaData() method updates the data set page design with metadata returned by the query by performing the following tasks:
 - Calls the DesignSessionUtil.toResultSetColumnsDesign() method to convert the run-time metadata to a design-time ResultSetColumns object
 - Obtains a ResultSetDefinition object from the design factory to use in populating the data set page design with the metadata definitions
 - Calls the resultSetDefn.setResultSetColumns() method to set the reference to ResultSetColumns object, containing the metadata content
 - Assigns the result set definition to the data set design

Listing 20-31 shows the code for the setResultSetMetaData() method.

Listing 20-31 The setResultSetMetaData() method

```
private void setResultSetMetaData(
    DataSetDesign dataSetDesign, IResultSetMetaData md )
throws OdaException
{
    ResultSetColumns columns =
        DesignSessionUtil.toResultSetColumnsDesign( md );
    ResultSetDefinition resultSetDefn =
        DesignFactory.eINSTANCE.createResultSetDefinition( );
    resultSetDefn.setResultSetColumns( columns );
    dataSetDesign.setPrimaryResultSet( resultSetDefn );
    dataSetDesign.getResultSets().setDerivedMetaData( true );
}
```

- savePage() method performs the following tasks:
 - Instantiates the CSVFileDriver

- Gets the result set metadata
- Updates the data set design with the metadata
- Closes the connection

Listing 20-32 shows the code for the savePage() method.

Listing 20-32 The savePage() method

```
private void savePage( DataSetDesign dataSetDesign )
{
    String queryText = getQuery( );
    dataSetDesign.setQueryText( queryText );
    IConnection conn = null;
    try
    {
        IDriver ffDriver = new CSVFileDriver( );
        conn = ffDriver.getConnection( null );
        IResultSetMetaData metaData =
            getResultSetMetaData( queryText, m_selectedFile,
                                  conn );
        setResultSetMetaData( dataSetDesign, metaData );
    }
    catch( OdaException e )
    {
        dataSetDesign.setResultSets( null );
    }
    finally
    {
        closeConnection( conn );
    }
}
```

Testing the CSV ODA UI plug-in

On PDE Manifest Editor, in Overview, the Testing section contains links to launch a plug-in as a separate Eclipse application in either Run or Debug mode.

How to launch the CSV report rendering plug-in

- 1 From the Eclipse SDK menu, choose Run—Open Run Dialog. On Run, right-click Eclipse Application. Choose New.
- 2 Create a configuration to launch an Eclipse application by performing the following tasks:
 - 1 In Name, type:
CSV ODA Test

2 On Main, in Location, type:

C:\BIRT2.2.1_Test\runTime-TestODA\testCSVODA

Run appears as shown in Figure 20-16.

3 Choose Run to launch the run-time workbench.

4 In the run-time workbench, choose the Report Design perspective.

5 In Report Design, create a new report project and create a new blank report.

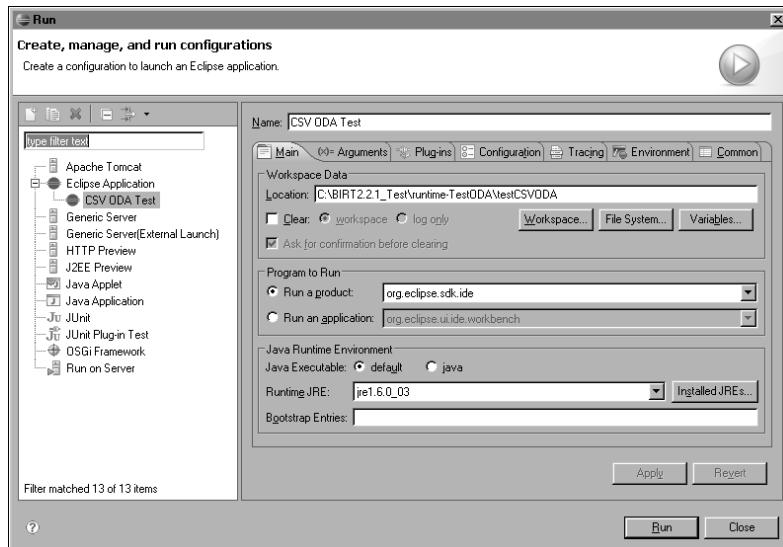


Figure 20-16 Creating a configuration to launch an Eclipse application

How to create a report design

1 In Report Design, choose File->New->Project.

2 Expand Business Intelligence and Reporting Tools and choose Report Project. Choose Next. New Report Project appears.

3 In Report Project, perform the following tasks:

1 In Project name, type:

testCSVODA

2 Select Use default location. Choose Finish. In Navigator, testCSVODA appears.

4 In Navigator, right-click testCSVODA and choose File->New->Report. New Report appears.

5 On New Report, perform the following tasks:

- 1 In Enter or select the parent folder, select testCSVODA.
 - 2 In file name, type:
`new_report.rptdesign`
- Choose Next.
- 3 In Report Templates, select Blank Report. Choose Finish. In Navigator, `new_report_1.rptdesign` appears in the testCSVODA project folder.
 - 6 Right-click `new_report.rptdesign` and choose Open. `new_report.rptdesign` appears in Report Design, as shown in Figure 20-17.

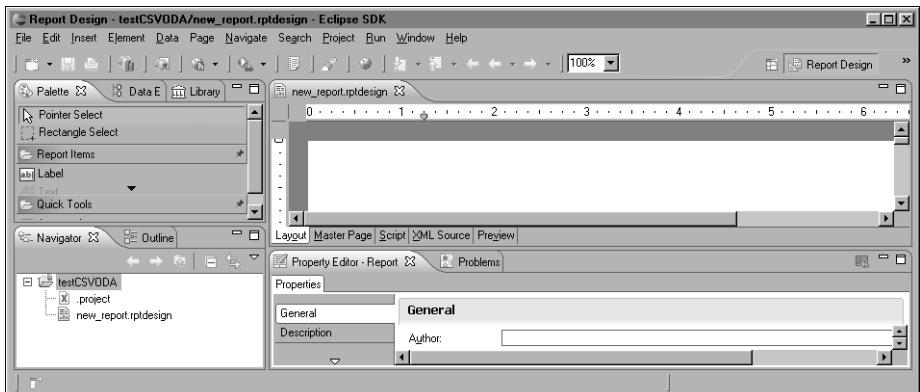


Figure 20-17 new_report.rptdesign in the report design environment

How to specify a data source

- 1 In Report Design, choose Data->New Data Source. On New Data Source, choose CSV Data Source, as shown in Figure 20-18. Choose Next. Select File appears.

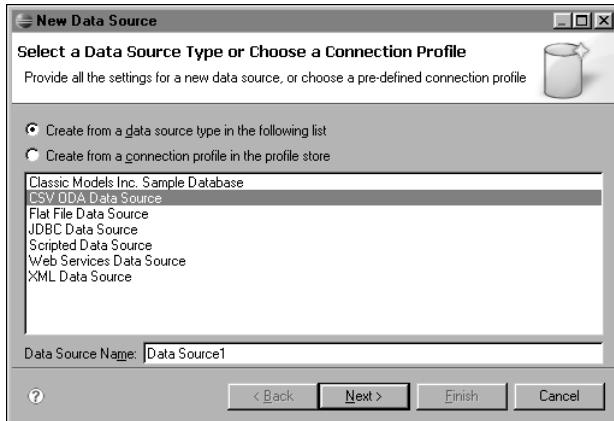


Figure 20-18 Choosing CSV Data Source

- 2** In Select File, enter the path and file name of the directory that contains the CSV data source file, as shown in Figure 20-19. Choose Finish. Report Design appears.



Figure 20-19 Path to the CSV data source file directory

How to select a new data set

- 1** In Report Design, choose Data->New Data Set. New Data Set appears, as shown in Figure 20-20. Choose Next. Select Columns appears.

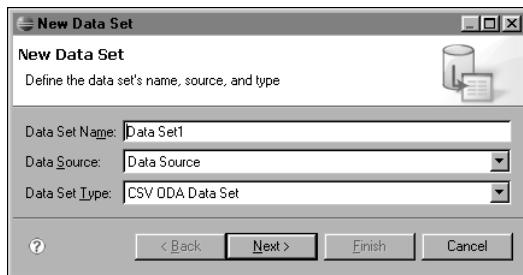


Figure 20-20 New Data Set

- 2** On Select Columns, select all the columns, as shown in Figure 20-21. Choose Finish.

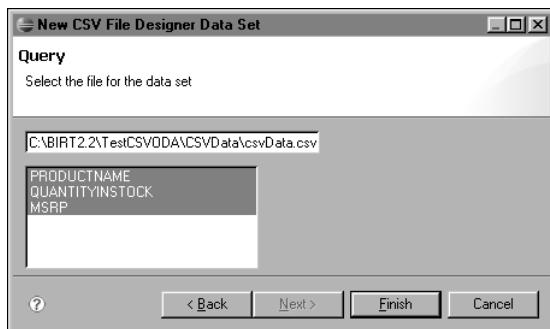


Figure 20-21 Selecting columns

Edit Data Set appears, as shown in Figure 20-22.

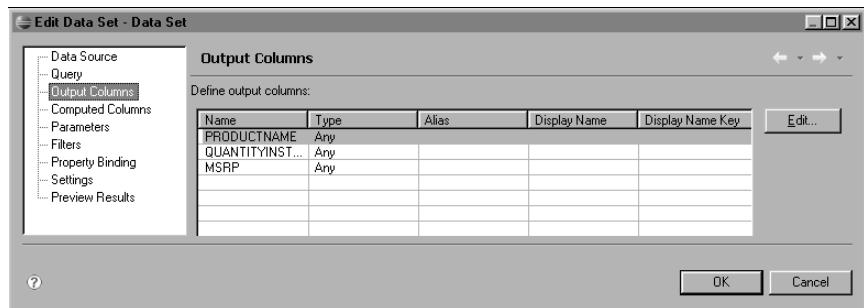


Figure 20-22 Edit Data Set

- 3 Choose Preview Results. Preview Results appears as shown in Figure 20-23. Choose OK.

Preview Results		
PRODUCTNAME	QUANTITYINSTOCK	MSRP
1969 Harley-Davidson...	7933	95.7
1992 Alpine Renault ...	7305	214.3
1996 Moto Guzzi 1100i	6625	118.94
2003 Harley-Davidso...	5582	193.66
1972 Alfa Romeo GTA 1...	3252	136
1962 Lancia A Delta 1...	6791	147.74
1968 Ford Mustang	58	194.57
2001 Ferrari Enzo	3619	207.8
1958 Selma Bus	1579	136.67

Figure 20-23 Data preview

- 4 On Data Explorer, expand Data Source and Data Sets. Data Explorer appears as shown in Figure 20-24.

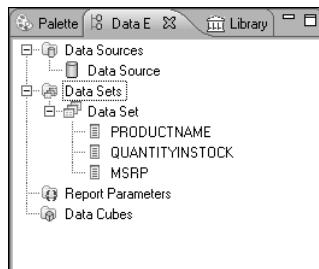


Figure 20-24 Data Set in Data Explorer

How to run a report design using CSV ODA UI and driver extensions

- 1 To build the report, drag Data Set from Data Explorer to the layout editor. Layout appears as shown in Figure 20-25.

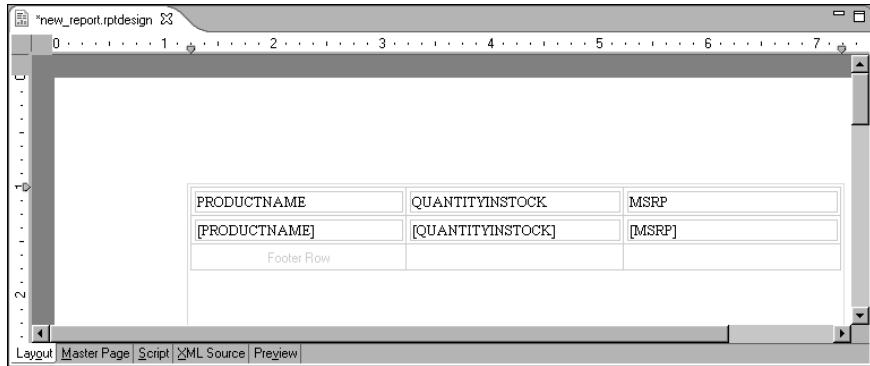


Figure 20-25 Report design in the layout editor

- 2** To run the report design, choose Preview. new_report_1.rptdesign runs, displaying the data set from the CSV data source, as shown in Figure 20-26.

PRODUCTNAME	QUANTITYINSTOCK	MSRP
1969 Harley Davidson Ultimate Chopper	7933	95.7
1952 Alpine Renault 1300	7305	214.3
1996 Moto Guzzi 1100i	6625	118.94
2003 Harley-Davidson Eagle Drag Bike	5582	193.66
1972 Alfa Romeo GTA	3252	136
1962 Lancia Delta 16V	6791	147.74
1968 Ford Mustang	68	194.57
2001 Ferrari Enzo	3619	207.8

Figure 20-26 Preview of the data set from the CSV data source

Developing a Hibernate ODA extension

You develop the Hibernate ODA extension by creating two new projects in the Eclipse PDE that implement the following plug-ins:

- org.eclipse.birt.report.data.oda.hibernate

The Hibernate ODA driver accesses a relational data source using HQL. The Hibernate ODA data source plug-in extends the functionality defined by the org.eclipse.datatools.connectivity.oda.dataSource extension point to create the Hibernate ODA driver.

- org.eclipse.birt.report.data.oda.hibernate.ui

The Hibernate ODA UI plug-in for BIRT Report Designer selects a Hibernate data source and allows the user to create an HQL statement to retrieve data from the available tables and columns. The Hibernate ODA UI plug-in extends the functionality defined by the org.eclipse.datatools

.connectivity.oda.design.ui.dataSource, org.eclipse.ui.propertyPages, and org.eclipse.datatools.connectivity.connectionProfile extension points.

The UI consists of the following pages:

- Data source page

Includes the Hibernate data source in the list of available data sources. The Hibernate ODA driver contains preconfigured Hibernate configuration and mapping files that connect to the MySQL version of the BIRT demonstration database, ClassicModels.

- Data set page

Creates an HQL statement that selects the data set and embeds the HQL statement in the report design.

In BIRT Report Designer, the Hibernate ODA data source wizard allows the report developer to select a Hibernate ODA driver containing preconfigured Hibernate configuration and mapping files. The Hibernate ODA driver searches for these configuration and mapping files in the plug-in's hibfiles directory.

The Hibernate ODA driver also searches in the hibfiles directory for JAR and ZIP files and the org.eclipse.birt.report.data.oda.jdbc plug-in for JDBC drivers to add to the classpath. This approach prevents the need to copy drivers to multiple locations. Note that changing the configuration causes the Hibernate ODA driver plug-in to rebuild the Hibernate SessionFactory, which is a machine-intensive operation.

Once the Hibernate ODA driver creates the data source configuration, you can create a data set. The Hibernate data set wizard allows the user to enter HQL statements. The Hibernate ODA UI example only supports simple queries, such as the following types of statements:

From Customer

or:

```
Select ord.orderNumber,cus.customerNumber, cus.customerName  
from Orders as ord, Customer as cus  
where ord.customerNumber = cus.customerNumber and  
cus.customerNumber = 363
```

In the Hibernate ODA plug-in, there is an exampleconfig directory. This directory contains a sample Hibernate configuration file, mapping files, and Java classes that connect to the BIRT sample MySQL database. You can test the plug-in using these files by performing the following tasks:

- Modify the hibernate.cfg.xml file to connect to your database configuration.
- Copy these files to the hibfiles directory.
- Create a JAR file containing the Java classes.

You can test and deploy the extensions in the Eclipse PDE run-time environment.

The following sections describe how to create and deploy the Hibernate ODA driver and UI plug-in projects. You can download the source code for the Hibernate ODA driver and UI extension examples at <http://www.actuate.com/birt/contributions>.

Creating the Hibernate ODA driver plug-in project

Create the Hibernate ODA driver plug-in project, org.eclipse.birt.report.data.oda.hibernate, using the New Plug-in Project wizard in the Eclipse PDE.

How to create the Hibernate ODA driver plug-in project

- 1 From the Eclipse PDE menu, choose File->New->Project.
- 2 On New Project—Select a wizard, open Business Intelligence and Reporting Tools and select ODA Runtime Driver Plug-in Project. Choose Next. New Plug-in Project appears.
- 3 In Plug-in Project, modify the settings as shown in Table 20-6.

Table 20-6 Settings for Plug-in Project options

Section	Option	Value
Plug-in Project	Project name	org.eclipse.birt.report.data.oda.hibernate
	Use default location	Selected
	Location	Not available when you select Use default location
Project Settings	Create a Java project	Selected
	Source folder name	src
	Output folder name	bin
Target Platform	Eclipse version	3.3
	OSGi framework	Deselected

- 4 On Plug-in Content, modify the settings as shown in Table 20-7.

Table 20-7 Settings for Plug-in Content options

Section	Option	Value
Plug-in Properties	Plug-in ID	org.eclipse.birt.report.data.oda.hibernate
	Plug-in Version	2.0.0
	Plug-in Name	BIRT ODA-Hibernate Driver

Table 20-7 Settings for Plug-in Content options (*continued*)

Section	Option	Value
Plug-in Options	Plug-in Provider	yourCompany.com or leave blank
	Classpath	odahibernate.jar
	Generate an activator, a Java class that controls the plug-in's life cycle	Selected
	Activator	org.eclipse.birt.report.data.oda.hibernate.Activator
Rich Client Application	This plug-in will make contributions to the UI	Deselected
	Would you like to create a rich client application?	No

Choose Next. Templates appears.

- 5 In Templates, choose ODA Data Source Runtime Driver. Choose Next. ODA Data Source Runtime Driver appears.
- 6 In ODA Data Source Runtime Driver, specify values for the following options used to generate the ODA plug-in:
 - 1 In Java Package Name, type:
`org.eclipse.birt.report.data.oda.hibernate`
 - 2 In ODA Data Source Element Id, type:
`org.eclipse.birt.report.data.oda.hibernate`
 - 3 In Data Source Display Name, type:
`Hibernate Data Source`
 - 4 In Number of Data Source Properties, type:
2
 - 5 In Data Set Display Name, type:
`Hibernate Data Set`
 - 6 In Number of Data Set Properties, type:
0

Choose Finish. The Hibernate ODA driver plug-in project appears in the Eclipse PDE workbench.

How to specify the properties of the Hibernate ODA plug-in project

- 1 Using the Eclipse PDE Manifest Editor, in Dependencies, specify the following required plug-ins in the following order:

- org.eclipse.core.runtime
 - org.eclipse.datatools.connectivity.oda
 - org.eclipse.birt.report.data.oda.jdbc
- 2** On Runtime, in Exported Packages, verify that the following packages that the plug-in exposes to clients appears in the list:
- antlr
 - antlr.actions.cpp
 - antlr.actions.csharp
 - antlr.actions.java
 - antlr.actions.python
 - antlr.ASdebug
 - antlr.build
 - antlr.collections
 - antlr.collections.impl
 - antlr.debug
 - antlr.debug.misc
 - antlr.preprocessor
 - javax.transaction
 - javax.transaction.xa
 - net.sf.cglib.beans
 - net.sf.cglib.core
 - net.sf.cglib.proxy
 - net.sf.cglib.reflect
 - net.sf.cglib.transform
 - net.sf.cglib.transform.hook
 - net.sf.cglib.transform.impl
 - net.sf.cglib.util
 - net.sf.ehcache
 - net.sf.ehcache.config
 - net.sf.ehcache.hibernate
 - net.sf.ehcache.store
 - org.apache.commons.collections

- org.apache.commons.collections.comparators
- org.apache.commons.collections.iterators
- org.apache.commons.logging
- org.apache.commons.logging.impl
- org.apache.tools.ant.taskdefs.optional
- org.dom4j
- org.dom4j.bean
- org.dom4j.datatype
- org.dom4j.dom
- org.dom4j.dtd
- org.dom4j.io
- org.dom4j.jaxb
- org.dom4j.rule
- org.dom4j.rule.pattern
- org.dom4j.swing
- org.dom4j.tree
- org.dom4j.util
- org.dom4j.xpath
- org.dom4j.xpp
- org.eclipse.birt.report.data.oda.hibernate
- org.hibernate
- org.hibernate.action
- org.hibernate.cache
- org.hibernate.cache.entry
- org.hibernate.cfg
- org.hibernate.classic
- org.hibernate.collection
- org.hibernate.connection
- org.hibernate.context
- org.hibernate.criterion
- org.hibernate.dialect
- org.hibernate.dialect.function

- org.hibernate.engine
- org.hibernate.engine.query
- org.hibernate.engine.transaction
- org.hibernate.event
- org.hibernate.event.def
- org.hibernate.exception
- org.hibernate.hql
- org.hibernate.hql.antlr
- org.hibernate.hql.ast
- org.hibernate.hql.ast.exec
- org.hibernate.hql.ast.tree
- org.hibernate.hql.ast.util
- org.hibernate.hql.classic
- org.hibernate.id
- org.hibernate.impl
- org.hibernate.intercept
- org.hibernate.jdbc
- org.hibernate.jmx
- org.hibernate.loader
- org.hibernate.loader.collection
- org.hibernate.loader.criteria
- org.hibernate.loader.custom
- org.hibernate.loader.entity
- org.hibernate.loader.hql
- org.hibernate.lob
- org.hibernate.mapping
- org.hibernate.metadata
- org.hibernate.param
- org.hibernate.persister
- org.hibernate.persister.collection
- org.hibernate.persister.entity
- org.hibernate.pretty

- org.hibernate.property
- org.hibernate.proxy
- org.hibernate.secure
- org.hibernate.sql
- org.hibernate.stat
- org.hibernate.tool.hbm2ddl
- org.hibernate.tool.instrument
- org.hibernate.transaction
- org.hibernate.transform
- org.hibernate.tuple
- org.hibernate.type
- org.hibernate.usertype
- org.hibernate.util
- org.objectweb.asm
- org.objectweb.asm.attrs

3 On Runtime, in Classpath, add the following JAR files to the plug-in classpath:

- odahibernate.jar
- lib/ant-antlr-1.6.5.jar
- lib/antlr-2.7.6rc1.jar
- lib/asm.jar
- lib/asm-attrs.jar
- lib/cglib-2.1.3.jar
- lib/commons-collections-2.1.1.jar
- lib/commons-logging-1.0.4.jar
- lib/dom4j-1.6.1.jar
- lib/ehcache-1.1.jar
- lib/hibernate3.jar
- lib/jta.jar

You must have previously imported these JAR files into the lib directory in the Hibernate ODA plug-in. You can also put these JAR files in a new plug-in that the Hibernate ODA plug-in references.

- 4** On Extensions, add the extension point, org.eclipse.datatools.connectivity.oda.dataSource, and the following elements and details for:

- **dataSource**

Add the extension element details, as shown in Table 20-8.

Table 20-8 Property settings for the dataSource extension element

Property	Value
id	org.eclipse.birt.report.data.oda.hibernate
driverClass	org.eclipse.birt.report.data.oda.hibernate.HibernateDriver
odaVersion	3.0
defaultDisplayName	Hibernate Data Source
setThreadContextClassLoader	true

The dataSource extension has an attribute named setThreadContextClassLoader, which, if set to true, sets the thread context class loader to the Hibernate ODA plug-in class loader. In this example, this attribute is set to true to avoid potential class conflicts with classes loaded with the Eclipse Tomcat plug-in.

- **dataSet**

Add the extension element details, as shown in Table 20-9.

Table 20-9 Property settings for the dataSet extension element

Property	Value
id	org.eclipse.birt.report.data.oda.hibernate.dataSet
defaultDisplayName	Hibernate Data Set

- 5** On Extensions, select dataSource and add the following properties and element details:

- HIBCONFIG, as shown in Table 20-10.

Table 20-10 HIBCONFIG property settings

Property	Value
name	HIBCONFIG
defaultDisplayName	Hibernate Configuration File
type	string

Table 20-10 HIBCONFIG property settings (*continued*)

Property	Value
canInherit	true

- MAPDIR, as shown in Table 20-11.

Table 20-11 MAPDIR property settings

Property	Value
name	MAPDIR
defaultDisplayName	Hibernate Mapping Directory
type	string
canInherit	true

- 6** On Extensions, select dataSet and add the list of dataTypeMapping elements, as shown in Table 20-12.

Table 20-12 Settings for dataTypeMapping elements

nativeDataType	nativeDataTypeCode	odaScalarDataType
BIT	-7	Integer
TINYINT	-6	Integer
SMALLINT	5	Integer
INTEGER	4	Integer
BIGINT	-5	Decimal
FLOAT	6	Double
REAL	7	Double
DOUBLE	8	Double
NUMERIC	2	Decimal
DECIMAL	3	Decimal
CHAR	1	String
VARCHAR	12	String
LONGVARCHAR	-1	String
DATE	91	Date
TIME	92	Time
TIMESTAMP	93	Timestamp
BINARY	-2	String
VARBINARY	-3	String
LONGVARBINARY	-4	String

(continues)

Table 20-12 Settings for dataTypeMapping elements (*continued*)

nativeDataType	nativeDataTypeCode	odaScalarDataType
BOOLEAN	16	Integer
BLOB	2004	Blob
CLOB	2005	Clob

Understanding the sample Hibernate ODA driver extension

The package for the Hibernate ODA extension example, org.eclipse.birt.report.data.oda.hibernate, implements the following classes using the ODA plug-in interfaces defined in the DTP plug-in, org.eclipse.datatools.connectivity.oda, and the extension points defined in the XML Schema file, datasource.exsd. The package implements the following classes:

- Activator
 - Extends org.eclipse.core.runtime.Plugin. Defines the methods for starting, managing, and stopping a plug-in instance.
- HibernateDriver
 - Implements the IDriver interface. Instantiates the connection object for the Hibernate ODA driver, which provides the entry point for the Hibernate ODA plug-in.
- Connection
 - Implements the IConnection interface. Opens and closes the connection to the Hibernate ODA data source and instantiates the IQuery object.
- Statement
 - Implements the IQuery interface. Prepares the result set metadata containing the table and column names, executes the query, and fetches the data rows from the data source.
- ResultSet
 - Implements the IResultSet interface. Provides access to the data rows in the result set, maintaining a cursor that points to the current row. Handles the processing that gets the value for a column as the specified data type.
- ResultSetMetaData
 - Implements the IResultSetMetaData interface. Describes the metadata for each column in the result set.
- DataSetMetaData
 - Implements the IDataSetMetaData interface. Describes the features and capabilities of the driver for the data set.

- **Messages**
Defines the exception messages for the Hibernate ODA driver.
- **DataTypes**
Defines, validates, and returns the data types supported by the Hibernate ODA driver.
- **CommonConstant**
Defines the constants used in the package, such as the driver name, ODA version, query keywords, and delimiters.
- **HibernateUtil**
Manages the Hibernate SessionFactory that provides the session or run-time interface between the Hibernate service and the ODA driver. This class is built based on the example HibernateUtil, available at <http://www.hibernate.org>.

The Hibernate ODA driver plug-in supports specifying the Hibernate configuration file and mapping files directory in the data source wizard. The plug-in creates the Hibernate SessionFactory from these settings. The example project has an exampleconfig directory that contains a Hibernate configuration and mapping files for use with the BIRT MySQL example database, ClassicModels.

The following sections describe the classes where there are important differences between the implementation of Hibernate ODA driver and the earlier example, the CSV ODA driver.

Understanding HibernateDriver

The HibernateDriver class instantiates the Connection object for the Hibernate ODA driver. This class implements the IDriver interface, but does not provide any processing for the methods that configure logging and set the application context. Listing 20-33 shows the getConnection() method.

Listing 20-33 The getConnection() method

```
public IConnection getConnection( String connectionClassName )
    throws OdaException
{
    return new Connection();
}
```

getMaxConnections() returns 0, imposing no limit on the number of connections to the ODA data source from the application. Listing 20-34 shows the getMaxConnections() method.

Listing 20-34 The getMaxConnections() method

```
public int getMaxConnections( ) throws OdaException
{
```

```
        return( 0 );
    }
```

Understanding Connection

The Connection class implements the following methods:

- `open()`

Opens a Hibernate session and sets the Boolean variable, `isOpen`, to true. The `open()` method uses the `HibernateUtil` class to obtain a session from a Hibernate SessionFactory, providing the run-time interface between the Hibernate service and the ODA driver.

The `open()` method retrieves the locations for the Hibernate configuration file and mapping files directory from connection properties. The `open()` method calls `HibernateUtil.constructSessionFactory()`, which attempts to build the SessionFactory with these settings. If the SessionFactory already exists, the plug-in does not recreate the SessionFactory unless the Hibernate configuration file or the mapping directory have changed.

Listing 20-35 shows the code for the `open()` method.

Listing 20-35 The `open()` method

```
public void open( Properties connProperties )
    throws OdaException
{
    try
    {
        configFile =
            connProperties.getProperty( "HIBCONFIG" );
        mapdir = connProperties.getProperty( "MAPDIR" );
        HibernateUtil
            .constructSessionFactory( configFile, mapdir );
        Session testSession = HibernateUtil.currentSession( );
        this.isOpen = true;
    }catch( Exception e )
    {
        throw new OdaException( e.getLocalizedMessage() );
    }
}
```

- `newQuery()`

Opens a new query by returning an instance of a `Statement` object, the class that implements the `IQuery` interface. The connection can handle multiple result set types, but the Hibernate ODA example uses only one and ignores the `dataSetType` parameter, as shown in Listing 20-36.

Listing 20-36 The newQuery() method

```
public IQuery newQuery( String dataSetType )
    throws OdaException
{
    if ( !isOpen( ) )
        throw new OdaException( Messages.getString(
            ( "Common.CONNECTION_IS_NOT_OPEN" ) ) );
    return new Statement( this );
}
```

- **getMetaData()**

Returns an IDatasetMetaData object of the data set type, as shown in Listing 20-37.

Listing 20-37 The getMetaData() method

```
public IDatasetMetaData getMetaData( String dataSetType )
    throws OdaException
{
    return new DataSetMetaData( this );
}
```

- **getMaxQueries()**

Indicates the maximum number of queries the driver supports. The getMaxQueries() method returns 1, indicating that the Hibernate ODA driver does not support concurrent queries, as shown in Listing 20-38.

Listing 20-38 The getMaxQueries() method

```
public int getMaxQueries( ) throws OdaException
{
    return 1;
}
```

- **commit() and rollback()**

Handle transaction processing. The Hibernate ODA driver example does not support transaction operations. In the Connection class, the commit() and rollback() methods throw UnsupportedOperationException. Listing 20-39 shows the code for the commit() method.

Listing 20-39 The commit() method

```
public void commit( ) throws OdaException
{
    throw new UnsupportedOperationException( );
}
```

- `close()`

Closes the Hibernate session, as shown in Listing 20-40.

Listing 20-40 The `close()` method

```
public void close( ) throws OdaException
{
    this.isOpen = false;
    try{
        HibernateUtil.closeSession( );
    }catch(Exception e){
        throw new OdaException( e.getLocalizedMessage( ) );
    }
}
```

Understanding DataSetMetaData

The `DataSetMetaData` class describes the features and capabilities of the data source for the specified data set. The Hibernate ODA driver example returns true or false to indicate support for a feature. The Hibernate ODA driver example does not support input or output parameters, named parameters, or multiple result sets.

The following code example indicates that the Hibernate ODA driver does not support multiple result sets, as shown in Listing 20-41.

Listing 20-41 The `supportsMultipleResultSets()` method

```
public boolean supportsMultipleResultSets( ) throws
    OdaException
{
    return false;
}
```

A method such as `getSQLStateType()`, which has no implementation, simply throws `UnsupportedOperationException`, as shown in Listing 20-42.

Listing 20-42 The `getSQLStateType()` method

```
public int getSQLStateType( ) throws OdaException
{
    throw new UnsupportedOperationException( );
}
```

Understanding Statement

The `Statement` class implements the `IQuery` interface. This class prepares and executes the query. `Statement` also handles parameters and retrieves the result set and result set metadata.

The `Statement` class implements the following methods:

- `prepare()`

The ODA framework calls the `prepare()` method before executing the query. The ODA framework uses the query saved in the report design.

The Hibernate ODA UI plug-in also calls `prepare()` to verify the columns used in the report design. The UI plug-in passes an HQL statement that gets the columns from the result set object.

`prepare()` sets up the result-set metadata and stores the query in an object variable for use by the `executeQuery()` method. The ODA run time uses the result-set metadata to retrieve the data. BIRT Report Designer also uses the result-set metadata to display the columns in the UI.

The `prepare()` method performs the following operations:

- Sets up array lists to contain the columns, column types, and column classes
- Trims the query String
- Creates a Hibernate Query object, using the HQL query
- Gets the Hibernate column names, types, and classes for the query
- Instantiates a `ResultSetMetaData` object, passing in the column names and data types
- Saves the query for execution

Listing 20-43 shows the code for the `prepare()` method.

Listing 20-43 The `prepare()` method

```
public void prepare( String query ) throws OdaException
{
    Query qry = null;
    testConnection();
    ArrayList arColsType = new ArrayList();
    ArrayList arCols = new ArrayList();
    ArrayList arColClass = new ArrayList();

    String[ ] props = null;
    try
    {
        Session hibsession = HibernateUtil.currentSession();
        query = query.replaceAll( "[\\n\\r]+", " " );
        query = query.trim();
        qry = hibsession.createQuery( query );
        Type[ ] qryReturnTypes = qry.getReturnTypes();
        if( qryReturnTypes.length > 0
            && qryReturnTypes[0].isEntityType() )
        {
            for( int j=0; j< qryReturnTypes.length; j++ )
            {
```

```

String clsName=qryReturnTypes[j].getName();
props =
HibernateUtil.getHibernateProp( clsName );
for( int x = 0; x < props.length; x++ )
{
String propType =
HibernateUtil.getHibernatePropTypes
( clsName, props[x] );
if( DataTypes.isValidType( propType ) )
{
    arColsType.add( propType );
    arCols.add( props[x] );
    arColClass.add( clsName );
}
else
{
    throw new OdaException
( Messages.getString
( "Statement.SOURCE_DATA_ERROR" ) );
}
}
}
else
{
    props = extractColumns( qry.getQueryString() );
    for( int t=0; t < qryReturnTypes.length; t++ )
    {
        if( DataTypes.isValidType
( qryReturnTypes[t].getName() ) )
        {
            arColsType.add( qryReturnTypes[t].getName() );
            arCols.add( props[t] );
        }
        else
        {
            throw new OdaException
( Messages.getString
( "Statement.SOURCE_DATA_ERROR" ) );
        }
    }
}
}
catch( Exception e )
{
    throw new OdaException( e.getLocalizedMessage() );
}
this.resultSetMetaData = new ResultSetMetaData
(( String[ ] )arCols.toArray
( new String[arCols.size( )] ),

```

```

        (String[ ])arColsType.toArray
        ( new String[arColsType.size( )] ),
        (String[ ])arCols.toArray
        ( new String[arCols.size( )] ),
        (String[ ])arColClass.toArray
        ( new String[arColClass.size( )] )));
    this.query = query;
}

```

- **getMetaData()**

The BIRT framework calls `getMetaData()` after the `prepare()` method to retrieve the metadata for a result set. The BIRT framework uses the metadata to create the data set in the report.

`Listing 20-44` shows the code for the `getMetaData()` method.

Listing 20-44 The `getMetaData()` method

```

public IResultSetMetaData getMetaData( ) throws OdaException
{
    return this.resultSetMetaData;
}

```

- **executeQuery()**

The `executeQuery()` method executes the prepared query and retrieves the results. The `executeQuery()` method returns an `IResultSet` object, which is created using the list results, result-set metadata, and Hibernate types returned from the HQL query. The ODA framework uses the `IResultSet` object to iterate over the results.

The `executeQuery()` method performs the following operations:

- Sets up an array of `org.hibernate.type.Type` to map Java types to JDBC datatypes
- Sets up a list to contain the results set
- Trims the query String
- Instantiates a Hibernate Query object, creating the HQL query
- Executes the HQL query, returning the query result set in a List
- Gets the Hibernate types for the query result set
- Instantiates a `ResultSet` object, passing in the data, metadata, and Hibernate types

`Listing 20-45` shows the code for the `executeQuery()` method.

Listing 20-45 The `executeQuery()` method

```

public IResultSet executeQuery( ) throws OdaException
{
    Type[ ] qryReturnTypes = null;

```

```

List rst = null;
try
{
    Session hibsession = HibernateUtil.currentSession( );
    String qryStr = this.query;
    qryStr = qryStr.replaceAll( "[\\n\\r]+", " " );
    qryStr.trim( );
    Query qry = hibsession.createQuery( qryStr );
    rst = qry.list( );
    qryReturnTypes = qry.getReturnTypes( );

}
catch( Exception e )
{
    throw new OdaException( e.getLocalizedMessage( ) );
}
return new ResultSet
(
    rst, getMetaData( ), qryReturnTypes
);
}

```

■ close()

The close() method clears the Connection and ResultSetMetaData objects. In the Connection object, the close() method closes the Hibernate session.

Listing 20-46 shows the code for the Statement.close() method.

Listing 20-46 The Statement.close() method

```

public void close( ) throws OdaException
{
    connection = null;
    resultSetMetaData = null;
}

```

Understanding ResultSet

The ResultSet class implements the IResultSet interface. When this class is instantiated, it stores the list.iterator() passed from the Statement object. It uses the iterator when the ODA driver framework calls the next() method.

The iterator points to the next available row of data from the HQL query results. The framework calls the accessor methods that get the data types for the columns in the current row. For example, if the first column is a String, the framework calls getString(). This method calls the getResult() method, which interprets the HQL query results.

The getResult() method parses the results in one of the following ways, depending on whether the query returns a Hibernate EntityType or just an array of values:

- If the query uses HQL and each return type is an EntityType, getResult() gets each Column class and uses the Hibernate ClassMetaData methods to retrieve the value.
- If the query returns standard data types, getResult() gets each value or values, returning an Object containing the simple value or an array of Objects containing the multiple values.

Listing 20-47 shows the code for the getResult() method.

Listing 20-47 The getResult() method

```
private Object getResult( int rstcol ) throws OdaException
{
    Object obj = this.currentRow;
    Object value = null;
    try
    {
        if( qryReturnTypes.length >
            0 && qryReturnTypes[0].isEntityType( ) )
        {
            String checkClass =
                (( ResultSetMetaData )getMetaData( ))
                    .getColumnClass( rstcol );
            Object myVal =
                HibernateUtil.getHibernatePropVal( obj,
                    checkClass,
                    getMetaData( ).getColumnName( rstcol ) );
            value = myVal;
        }
        else
        {
            if( getMetaData( ).getColumnCount( ) == 1 )
            {
                value = obj;
            }
            else
            {
                Object[ ] values = ( Object[ ] )obj;
                value = values[rstcol-1];
            }
        }
    }
    catch( Exception e )
    {
        throw new OdaException( e.getLocalizedMessage( ) );
    }
    return( value );
}
```

Understanding HibernateUtil

HibernateUtil is a utility class that provides the run-time interface between the Hibernate service and the application. The HibernateUtil class example derives from the class provided with the Hibernate documentation. HibernateUtil performs the following operations:

- Initializes the SessionFactory
- Builds the Hibernate SessionFactory
- Opens and closes a session
- Returns information on Hibernate classes and properties
- Registers the JDBC driver with the DriverManager

The Connection.open() method calls HibernateUtil.constructSessionFactory(), which creates a SessionFactory if one does not already exist. The constructSessionFactory() method closes and rebuilds the SessionFactory if the location of the configuration file or mapping files directory has changed.

The SessionFactory construction process creates the ClassLoader. The ClassLoader adds the drivers directory in the org.eclipse.birt.report.data.oda.jdbc plug-in and the hibfiles directory in the Hibernate ODA plug-in to classpath. This process also registers the JDBC driver specified in the Hibernate config file with the DriverManager.

The HibernateUtil class implements the following methods:

- initSessionFactory()

This method creates the SessionFactory object from the configuration settings in the hibernate.cfg.xml file. Listing 20-48 shows the code for the initSessionFactory() method.

Listing 20-48 The initSessionFactory() method

```
private static synchronized void initSessionFactory
( String hibfile, String mapdir)
throws HibernateException
{
    if( sessionFactory == null)
    {
        Thread thread = Thread.currentThread( );
        try
        {
            oldloader = thread.getContextClassLoader( );
            refreshURLs( );
            ClassLoader changeLoader = new URLClassLoader
                ( ( URL [ ] )URLList.toArray
                    ( new URL[0] ), thread
                        .getContextClassLoader( ) );
            thread.setContextClassLoader( changeLoader );
            Configuration cfg =

```

```

        buildConfig( hibfile,mapdir );
Class driverClass =
    changeLoader.loadClass( cfg.getProperty
        ( "connection.driver_class" ) );
Driver driver =
    ( Driver ) driverClass.newInstance( );
WrappedDriver wd =
    new WrappedDriver( driver,
        cfg.getProperty
        ( "connection.driver_class" ) );
boolean foundDriver = false;
Enumeration drivers =
    DriverManager.getDrivers( );
while ( drivers.hasMoreElements( ) )
{
    Driver nextDriver =
        ( Driver )drivers.nextElement( );
    if ( nextDriver.getClass( ) == wd.getClass( ) )
    {
        if( nextDriver.toString( )
            .equals(wd.toString( )) )
        {
            foundDriver = true;
            break;
        }
    }
}
if( !foundDriver )
{
    DriverManager.registerDriver( wd );
}
sessionFactory = cfg.buildSessionFactory( );
configuration = cfg;
HibernateMapDirectory = mapdir;
HibernateConfigFile = hibfile;
}
catch( Exception e)
{
    e.printStackTrace( );
    throw new HibernateException
        ( "No Session Factory Created " +
            e.getLocalizedMessage( ) );
}
finally
{
    thread.setContextClassLoader( oldloader );
}
}
}

```

- **constructSessionFactory**

This method checks to see if a configuration change occurred. If a change occurred, the method closes the session and SessionFactory and calls the

`initSessionFactory` to rebuild the `SessionFactory`.

Listing 20-49 shows the code for the `constructSessionFactory()` method.

Listing 20-49 The `constructSessionFactory()` method

```
public static void constructSessionFactory
( String hibfile, String mapdir)
throws HibernateException
{
    if( hibfile == null)
    {
        hibfile = "";
    }
    if( mapdir == null)
    {
        mapdir = "";
    }
    if( sessionFactory == null)
    {
        initSessionFactory( hibfile, mapdir);
        return;
    }
    if( HibernateMapDirectory.equalsIgnoreCase(
        ( mapdir ) && HibernateConfigFile
            .equalsIgnoreCase( hibfile )))
    {
        return;
    }
    synchronized( sessionFactory )
    {
        Session s = ( Session ) session.get( );
        if ( s != null )
        {
            closeSession( );
        }
        if ( sessionFactory != null &&
            !sessionFactory.isClosed( ))
        {
            closeFactory( );
        }
        sessionFactory = null;
        initSessionFactory( hibfile, mapdir);
    }
}
```

- `currentSession()`

This method opens a session when called by the `Connection.open()` method, as shown in Listing 20-50.

Listing 20-50 The `currentSession()` method

```
public static Session currentSession( )
throws HibernateException {
```

```

Session s = ( Session ) session.get( );
if ( s == null ) {
    s = sessionFactory.openSession( );
    session.set( s );
}
return s;
}

```

Other methods in this class return information on a particular class and its properties. The getHibernateProp() method returns the properties for a class. The getHibernatePropTypes() method returns the data type for a property of a class.

Building the Hibernate ODA driver plug-in

To build and deploy the org.eclipse.birt.report.data.oda.hibernate plug-in using the Eclipse PDE Manifest Editor, perform the following tasks:

- On Build, specify the Build Configuration to include the following items:
 - In Runtime Information, add the odahibernate.jar file.
 - In Binary Build, select the following files and folders:
 - META-INF
 - exampleconfig
 - hibfiles
 - lib
 - plugin.xml
- On Overview, in Exporting, choose Export Wizard and perform the following tasks:
 - In Available Plug-ins and Fragments, select org.eclipse.birt.report .data.oda.hibernate.
 - In Options, verify that Package plug-ins as individual JAR archives is not selected.
 - In Destination, choose the directory, \$INSTALL_DIR\birt-runtime- 2_2_1\Report Engine.

The Hibernate ODA example uses MySQL as the database. The BIRT sample database and the MySQL installation scripts can be downloaded from <http://www.eclipse.org/birt/db>. For information about the required Hibernate libraries, please refer to the Hibernate web site at <http://www.hibernate.org>.

Developing the Hibernate ODA UI extension

To use the data retrieved by the Hibernate ODA driver in a BIRT report design, you must extend the DTP design UI. To implement the Hibernate ODA UI, you extend the following extension points:

- `org.eclipse.datatools.connectivity.oda.design.ui.dataSource`

The `dataSource` extension point defines and implements the UI for new data source and data set wizards. These wizards use the Hibernate ODA driver plug-in to extend the functionality available in the Data Explorer of BIRT Report Designer.

- `org.eclipse.ui.propertyPages`

The `propertyPage` extension displays and manipulates the Hibernate configuration file and mapping files directory locations.

- `org.eclipse.datatools.connectivity.connectionProfile`

The `connectionProfile` extension shares a data source connection between applications.

To start developing the Hibernate ODA UI plug-in, create the plug-in project, `org.eclipse.birt.report.data.oda.hibernate.ui`.

How to create the Hibernate ODA UI plug-in project

- 1 From the Eclipse menu, choose `File->New->Project`. `New Project` appears.
- 2 On `New Project`—Select a wizard, open Business Intelligence and Reporting Tools and select ODA Designer Plug-in Project. Choose `Next`. `New Plug-in Project` appears.
- 3 In `Plug-in Project`, modify the settings as shown in Table 20-13. Choose `Next`. `Plug-in Content` appears.

Table 20-13 Settings for Plug-in Project options

Section	Option	Value
Plug-in Project	Project name	<code>org.eclipse.birt.report.data.oda.hibernate.ui</code>
	Use default location	Selected
Project Settings	Location	Not available when you select Use default location
	Create a Java project	Selected
Target Platform	Source folder name	<code>src</code>
	Output folder name	<code>bin</code>
Target Platform	Eclipse version	<code>3.2</code>
	OSGi framework	Deselected

- 4** In Plug-in Content, modify the settings as shown in Table 20-14. Choose Finish.

Table 20-14 Settings for Plug-in Content options

Section	Option	Value
Plug-in Properties	Plug-in ID	org.eclipse.birt.report.data.oda.hibernate.ui
	Plug-in Version	2.0.0
	Plug-in Name	BIRT Hibernate UI Plug-in
	Plug-in Provider	yourCompany.com or leave blank
	Classpath	hibernateodaui.jar
Plug-in Options	Generate an activator, a Java class that controls the plug-in's life cycle	Selected
	Activator	org.eclipse.birt.report.data.oda.hibernate.ui.Activator
	This plug-in will make contributions to the UI	Deselected
Rich Client Application	Would you like to create a rich client application?	No

- 5** In Templates, choose ODA Data Source Designer. Choose Next. ODA Data Source Designer appears.

- 6** In ODA Data Source Designer, specify new values for the following options used to generate the Hibernate ODA UI plug-in:

1 In Java Package Name, type:

`org.eclipse.birt.report.data.oda.hibernate.ui`

2 In ODA Runtime Driver Plug-in Id, type:

`org.eclipse.birt.report.data.oda.hibernate`

3 In ODA Runtime Data Source Element Id, type:

`org.eclipse.birt.report.data.oda.hibernate`

4 In ODA Runtime Driver Class, type:

`org.eclipse.birt.report.data.oda.hibernate.Driver`

5 In ODA Runtime Data Set Element Id, type:

`org.eclipse.birt.report.data.oda.hibernate.dataSet`

- 6** In Data Source Display Name, type:
ODA Hibernate File Designer Data Source
- 7** In Data Set Display Name, type:
ODA Hibernate File Designer Data Set

Choose Finish. The Hibernate ODA UI plug-in project appears in the Eclipse PDE workbench.

How to specify the Hibernate ODA UI dependencies

On the Eclipse PDE Manifest Editor, in Dependencies, specify the required plug-ins in the following order:

- org.eclipse.core.runtime
- org.eclipse.ui
- org.eclipse.datatools.connectivity.oda.design.ui
- org.eclipse.birt.report.data.oda.hibernate

How to specify the Hibernate ODA UI runtime

On Runtime, in Exported Packages, add
org.eclipse.birt.report.oda.hibernate.ui to the list of packages that this plug-in exposes to clients.

How to specify the Hibernate ODA UI extension points

- 1** On the PDE Manifest Editor, choose Extensions.
- 2** In All Extensions, choose Add. New Extension appears.
- 3** On New Extension—Extension Points, in the list of extension points, select the following plug-in:
`org.eclipse.datatools.connectivity.oda.design.ui.dataSource`
Choose Finish.
- 4** Repeat steps 2 and 3 to add the following extension points to the list on the Extensions page:
 - org.eclipse.ui.propertyPages
 - org.eclipse.datatools.connectivity.connectionProfile

How to add the extension details

- 1** On Extensions, select the extension point, org.eclipse.datatools.connectivity.oda.design.ui.dataSource, and add the following elements and element details:
 - dataSourceUIAdd the following id:

```
org.eclipse.birt.report.data.oda.hibernate
```

Add the following extension element to dataSourceUI:

```
newDataSourceWizard
```

Add the extension element details for the extension element, newDataSourceWizard, as shown in Table 20-15.

Table 20-15 Property settings for newDataSourceWizard

Property	Value
pageClass	org.eclipse.birt.report.data.oda.hibernate.ui.HibernateDataSourceWizard
windowTitle	Hibernate Data Source
includesProgressMonitor	false
pageTitle	Hibernate Data Source

- **dataSetUI**

Add the extension element details for the extension element, dataSetUI, as shown in Table 20-16.

Table 20-16 Property settings for the dataSetUI extension element

Property	Value
id	org.eclipse.birt.report.data.oda.hibernate.dataSet
initialPageId	org.eclipse.birt.report.data.oda.hibernate.ui.HibernatePage
supportsInParameters	true
supportsOutParameters	false

- 2** On Extensions, select dataSetUI and add the following properties and element details:

- **dataSetWizard**, as shown in Table 20-17

Table 20-17 Property settings for the dataSetWizard extension element

Property	Value
class	org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSetWizard
windowTitle	Hibernate Data Set

- `dataSetPage`, as shown in Table 20-18

Table 20-18 Property settings for the `dataSetPage` extension element

Property	Value
<code>id</code>	<code>org.eclipse.birt.report.data.oda.hibernate.ui.HibernatePage</code>
<code>wizardPageClass</code>	<code>org.eclipse.birt.report.data.oda.hibernate.ui.HibernateHqlSelectionPage</code>
<code>displayName</code>	Enter HQL
<code>path</code>	/

- 3 On Extensions, select `org.eclipse.ui.propertyPages` and add the following page property and extension element details, as shown in Table 20-19.

Table 20-19 Property settings for the page extension element

Property	Value
<code>id</code>	<code>org.eclipse.birt.report.data.oda.hibernate.</code>
<code>name</code>	ODA Hibernate Data Source Connection Properties
<code>class</code>	<code>org.eclipse.birt.report.data.oda.hibernate.ui.HibernatePropertyPage</code>
<code>objectClass</code>	<code>org.eclipse.datatools.connectivity.IConnectionProfile</code>

- 4 On Extensions, select `page` and add the following filter property and extension element details, as shown in Table 20-20.

Table 20-20 Property settings for the filter extension element

Property	Value
<code>name</code>	<code>org.eclipse.datatools.profile.property.id</code>
<code>value</code>	<code>org.eclipse.birt.report.data.oda.hibernate</code>

- 5 On Extensions, select `org.eclipse.datatools.connectivity.connectionProfile`, and add the following properties and element details:

- `category`, as shown in Table 20-21

Table 20-21 Property settings for the filter extension element

Property	Value
<code>id</code>	<code>org.eclipse.birt.report.data.oda.hibernate</code>

Table 20-21 Property settings for the filter extension element (*continued*)

Property	Value
parentCategory	org.eclipse.datatools.connectivity.oda.profileCategory
name	Hibernate Data Source

- connectionProfile, as shown in Table 20-22

Table 20-22 Property settings for the connectionProfile extension element

Property	Value
id	org.eclipse.birt.report.data.oda.hibernate
category	org.eclipse.birt.report.data.oda.hibernate
name	ODA Hibernate Data Source Connection Profile
pingFactory	org.eclipse.datatools.connectivity.oda.profile.OdaConnectionFactory

- connectionFactory, as shown in Table 20-23

Table 20-23 Property settings for the connectionFactory extension element

Property	Value
id	ogr.eclipse.datatools.connectivity.oda.IConnection
class	ogr.eclipse.datatools.connectivity.oda.profile.OdaConnectionFactory
profile	org.eclipse.birt.report.data.oda.hibernate
name	ODA Connection Factory

- newWizard, as shown in Table 20-24

Table 20-24 Property settings for the newWizard extension element

Property	Value
id	org.eclipse.birt.report.data.oda.hibernate
name	ODA Hibernate Data Source
class	org.eclipse.datatools.connectivity.oda.design.ui.wizards.NewDataSourceWizard

(continues)

Table 20-24 Property settings for the newWizard extension element (*continued*)

Property	Value
profile	org.eclipse.birt.report.data.oda.hibernate
description	Create an ODA Hibernate connection profile

Understanding the sample Hibernate ODA UI extension

The following sections describe the code-based extensions a developer must make to complete the development of the Hibernate ODA UI extension, after defining the plug-in framework in the Eclipse PDE.

The Hibernate ODA UI plug-in implements the following classes:

- **HibernatePropertyPage**

Creates and initializes the editor controls for the property page that specify the ODA data source. This class updates the connection profile properties with the values collected from the page.

HibernatePropertyPage extends org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSourceEditorPage, the abstract base class for implementing a customized ODA data source property page.

- **HibernatePageHelper**

Implements the user interface that specifies data source properties. This utility class specifies the page layout, sets up the controls that listen for user input, verifies the location of the Hibernate configuration file, and sets up the location of the mapping directory. The HibernateDataSourceWizard and HibernatePropertyPage classes use HibernatePageHelper. HibernatePageHelper also extends org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSourceEditorPage.

- **HibernateDataSourceWizard**

Creates and initializes the controls for the data source wizard page. The class sets the configuration file message and collects the property values. In the extension element settings for newDataSourceWizard, the pageClass property specifies this class as the implementation class for the dataSourceUI wizard. The HibernateDataSourceWizard class extends org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSourceWizardPage, the abstract base class for implementing a customized ODA data source wizard page.

- **HibernateHqlSelectionPage**

Creates the user interface that specifies an HQL statement. The Hibernate ODA UI plug-in calls HibernateHqlSelectionPage when creating or

modifying the data set for a data source. In the extension element settings for `dataSetPage`, the `wizardPageClass` property specifies this class as the implementation class for the `dataSetUI` page wizard. `HibernateHqlSelectionPage` also extends `org.eclipse.datatools.connectivity.oda.design.ui.wizards.DataSourceWizardPage`.

- **Messages**

This class and the related properties file, `messages.properties`, generate the messages displayed in the Hibernate ODA UI.

Understanding HibernatePageHelper

This class creates the components that select the Hibernate configuration file and a mapping files directory using the following methods:

- `createCustomControl()`
Builds the user interface for the data source
- `initCustomControl()`
Sets the initial property values
- `collectCustomProperties()`
Returns the modified properties to the ODA framework

When the data source page displays, the Finish button becomes available when the `setPageComplete()` method indicates the page is complete.

`HibernateDataSourceWizard.createPageCustomControl()` and `HibernatePropertyPage.createAndInitCustomControl()` call `HibernatePageHelper`. The `createCustomControl()` method is the entry point for this class.

Listing 20-51 shows the code for the `createCustomControl()` method.

Listing 20-51 The `createCustomControl()` method

```
void createCustomControl( Composite parent )
{
    Composite content = new Composite( parent, SWT.NULL );
    GridLayout layout = new GridLayout( 3, false );
    content.setLayout(layout);
    setupConfigLocation( content );
    setupMapLocation( content );
}
```

The `setupConfigLocation()` method sets up the configuration file location. The `setupMapLocation()` method sets up the mapping folder. These two methods perform similar tasks.

Listing 20-52 shows the code for the `setupConfigLocation()` method. This method adds a label, a text entry component, and a button. The text entry component has a `ModifyListener()` method, which verifies that the file

selected exists, and the button has a SelectionAdapter() method, which uses the FileDialog() method to access the configuration file.

Listing 20-52 The setupConfigLocation() method

```
private void setupConfigLocation( Composite composite )
{
    Label label = new Label( composite, SWT.NONE );
    label.setText("Select Hibernate Config File" );
    GridData data =
        new GridData( GridData.FILL_HORIZONTAL );
    m_configLocation = new Text( composite, SWT.BORDER );
    m_configLocation.setLayoutData( data );
    setPageComplete( true );
    m_configLocation.addModifyListener
        ( new ModifyListener()
    {
        public void modifyText( ModifyEvent e )
        {
            verifyConfigLocation( );
        }
    } );
    m_browseConfigButton = new Button( composite, SWT.NONE );
    m_browseConfigButton.setText( "..." );
    m_browseConfigButton.addSelectionListener
        ( new SelectionAdapter()
    {
        public void widgetSelected( SelectionEvent e )
        {
            FileDialog dialog = new FileDialog
                ( m_configLocation.getShell( ) );
            if( m_configLocation.getText( ) != null &&
                m_configLocation.getText( )
                .trim( ).length( ) > 0 )
            {
                dialog.setFilterPath
                    ( m_configLocation.getText() );
            }
            dialog.setText
                ( "Select Hibernate Config File" );
            String selectedLocation = dialog.open( );
            if( selectedLocation != null )
            {
                m_configLocation.setText
                    ( selectedLocation );
            }
        }
    } );
}
```

The initCustomControl() method initializes the properties settings. The plug-in passes the properties to the method from the createPageCustomControl() and setInitialProperties() methods of the

HibernateDataSourceWizard class and the createAndInitCustomControl() method of the HibernatePropertyPage class.

The initCustomControl() method retrieves the properties for the Hibernate configuration file and mapping files directory and sets text component values.

Listing 20-53 shows the code for the initCustomControl() method.

Listing 20-53 The initCustomControl() method

```
void initCustomControl( Properties profileProps )
{
    setPageComplete( true );
    setMessage( DEFAULT_MESSAGE, IMessageProvider.NONE );
    if( profileProps == null || profileProps.isEmpty() ||
        m_configLocation == null )
        return;
    String configPath =
        profileProps.getProperty( "HIBCONFIG" );
    if( configPath == null )
        configPath = EMPTY_STRING;
    m_configLocation.setText( configPath );
    String mapPath = profileProps.getProperty( "MAPDIR" );
    if( mapPath == null )
        mapPath = EMPTY_STRING;
    m_mapLocation.setText( mapPath );
    verifyConfigLocation();
}
```

When the user presses the Finish or Test Connection button, the plug-in calls the collectCustomProperties() method to retrieve the new values for the Hibernate configuration file and mapping files directory. The HibernateDataSourceWizard and HibernatePropertyPage classes call the HibernatePageHelper.collectCustomProperties() method from their collectCustomProperties() methods.

Listing 20-54 shows the code for the collectCustomProperties() method.

Listing 20-54 The collectCustomProperties() method

```
Properties collectCustomProperties( Properties props )
{
    if( props == null )
        props = new Properties();
    props.setProperty( "HIBCONFIG",
        getConfig( ) );
    props.setProperty( "MAPDIR", getMapDir( ) );
    return props;
}
```

Understanding HibernateDataSourceWizard

The HibernateDataSourceWizard class extends the DTP DataSourceWizardPage, by implementing three methods that the ODA framework calls:

- `createPageCustomControl()`
Constructs the user interface
- `setInitialProperties()`
Sets the initial values of the user interface
- `collectCustomProperties()`
Retrieve the modified values

This class creates the HibernatePageHelper class, and uses the methods described earlier to handle these three methods. The ODA framework uses this class to create a new data source.

Understanding HibernatePropertyPage

The HibernatePropertyPage class extends the DTP DataSourceEditorPage by implementing two methods that the ODA framework calls:

- `createAndInitCustomControl()`
Constructs the user interface and sets the initial values
- `collectCustomProperties()`
Retrieves the modified values

This class creates the HibernatePageHelper class and uses the methods described earlier to handle these two methods. The ODA framework uses this class to create a new data source.

Understanding HibernateHqlSelectionPage

The HibernateHqlSelectionPage class extends DataSetWizardPage to define the page controls and related functionality for the Hibernate ODA data set wizard. HibernateHqlSelectionPage allows the user to create an HQL statement that selects the data set and embeds the HQL statement in the report design. This page links to the Hibernate ODA through the `wizardPageClass` attribute of the `dataSetPage` element within the `dataSource` extension.

The HibernateHqlSelectionPage class implements the following methods:

- `createPageControl()`
This method performs the following operations:
 - Sets up a composite set of controls using a series of GridLayout and GridData objects to create the data set editor UI

- Sets the user prompt to enter an HQL statement and verify the query
- Adds a text control to allow the user to enter and modify text
- Adds a ModifyListener to the text control to detect user input
- Sets up the Verify Query button and adds a SelectionListener to detect when the user selects the button
- Returns the composite page control

Listing 20-55 shows the code for the createPageControl() method.

Listing 20-55 The createPageControl() method

```
public Control createPageControl( Composite parent )
{
    Composite composite = new Composite
        ( parent, SWT.NONE );
    GridLayout layout = new GridLayout( );
    layout.numColumns = 1;
    composite.setLayout( layout );
    Label label = new Label( composite, SWT.NONE );
    label.setText( Messages.getString
        ( "wizard.title.selectColumns" ) );
    GridData data = new GridData( GridData.FILL_BOTH );
    queryText = new Text( composite, SWT.MULTI |
        SWT.WRAP | SWT.V_SCROLL );
    queryText.setLayoutData( data );
    queryText.addModifyListener( new ModifyListener( ){
        public void modifyText( ModifyEvent e )
        {
            if( m_initialized == false )
            {
                setPageComplete(true);
                m_initialized = true;
            }
            else
            {
                setPageComplete(false);
            }
        }
    } );
    setPageComplete( false );
    Composite cBottom = new Composite
        ( composite, SWT.NONE );
    cBottom.setLayoutData
        ( new GridData( GridData.FILL_HORIZONTAL ) );
    cBottom.setLayout( new RowLayout( ) );
    queryButton = new Button( cBottom, SWT.NONE );
    queryButton.setText( Messages.getString
        ( "wizard.title.verify" ) );
    queryButton.addSelectionListener
        ( new SelectionAdapter( ) )
{
```

```

        public void widgetSelected( SelectionEvent event )
    {
        verifyQuery( );
    }
}

return composite;
}

```

- **initializeControl()**

The plug-in calls this method to retrieve the HQL query from the current design and initializes the HQL text component with this value. `initializeControl()` also reads the Hibernate configuration file and mapping files directory from the report design and stores them in member variables for use when building a query.

`Listing 20-56` shows the code for the `initializeControl()` method.

Listing 20-56 The `initializeControl()` method

```

private void initializeControl( )
{
    Properties dataSourceProps =
        getInitializationDesign( ).getDataSourceDesign( )
            .getPublicProperties( );
    m_hibconfig =
        dataSourceProps.getProperty( "HIBCONFIG" );
    m_mapdir = dataSourceProps.getProperty( "MAPDIR" );
    DataSetDesign dataSetDesign =
        getInitializationDesign( );
    if( dataSetDesign == null )
        return;
    String queryTextTmp = dataSetDesign.getQueryText( );
    if( queryTextTmp == null )
        return;
    queryText.setText( queryTextTmp );
    this.m_initialized = false;
    setMessage( "", NONE );
}

```

- **verifyQuery()**

This method is the selection event called when the user chooses the Verify Query button. `verifyQuery` performs the following operations:

- Opens a connection to the run-time environment.
- Instantiates a `Query` object and gets the query text entered by the user.
- Prepares the query.
- Checks the column to determine if the query prepare was successful. Depending on the success of the query prepare, `verifyQuery()` indicates that page processing is complete or incomplete.
- Re-enables the Verify Query button.

- Closes the connection.

Listing 20-57 shows the code for the verifyQuery() method.

Listing 20-57 The verifyQuery() method

```

boolean verifyQuery( )
{
    setMessage( "Verifying Query", INFORMATION );
    setPageComplete( false );
    queryButton.setEnabled( false );
    Connection conn = new Connection();
    try
    {
        Properties prop = new Properties();
        if( m_hibconfig == null)m_hibconfig = "";
        if( m_mapdir == null)m_mapdir = "";
        prop.put("HIBCONFIG", m_hibconfig );
        prop.put("MAPDIR", m_mapdir);
        conn.open( prop );
        IQuery query = conn.newQuery( "" );
        query.prepare( queryText.getText( ) );
        int columnCount =
            query.getMetaData().getColumnCount();
        if ( columnCount == 0 )
        {
            setPageComplete( false );
            return false;
        }
        setPageComplete( true );
        return true;
    }
    catch ( OdaException e )
    {
        System.out.println( e.getMessage() );
        showError( "ODA Verify Exception", e.getMessage() );
        setPageComplete( false );
        return false;
    }
    catch ( Exception e )
    {
        System.out.println( e.getMessage() );
        showError( "Verify Exception", e.getMessage() );
        setPageComplete( false );
        return false;
    }
    finally
    {
        try
        {
            queryButton.setEnabled( true );
            conn.close();
        }
        catch ( OdaException e )
    }
}

```

```

        {
            System.out.println( e.getMessage( ) );
            setMessage( e.getLocalizedMessage(),
                        ERROR );e.getMessage( ) );
            setPageComplete( false );
            return false;
        }
    }
}

```

- **canLeave()**

When the user chooses OK or attempts to leave the page, the plug-in calls the canLeave() method. If the HQL statement verifies or is unchanged, the plug-in permits the user to leave the page, and saves the HQL in the report. If the page is not complete the plug-in prompts the user to verify the query.

Listing 20-58 shows the code for the canLeave() method.

Listing 20-58 The canLeave() method

```

public boolean canLeave( )
{
    if ( !isPageComplete( ) )
    {
        setMessage( Messages.getString
                    ( "error.selectColumns" ), ERROR );
        return false;
    }
    return true;
}

```

- **savePage()**

The savePage() method is called when the ODA framework calls the collectDataSetDesign() method. This action occurs when the user presses the Finish button on the new data set wizard or the OK button on the data set editor is pressed. The savePage() method saves the query to the report, as shown in Listing 20-59.

Listing 20-59 The savePage() method

```

private boolean savePage( )
{
    IConnection conn = null;
    try
    {
        IDriver hqDriver = new HibernateDriver( );
        conn = hqDriver.getConnection( null );
        IResultSetMetaData metadata =
            getResultSetMetaData( dataSetDesign
                                  .getQueryText( ), conn );
        setResultSetMetaData( dataSetDesign, metadata );
    }
}

```

```

        }
        catch( OdaException e )
        {
            dataSetDesign.setResultSetSets( null );
        }
        finally
        {
            closeConnection( conn );
        }
    }
}

```

- **getResultSetMetaData()**

The savePage() method calls the getResultSetMetaData() method when saving the report design. This method retrieves the query metadata that setResultSetMetaData() uses to create the data set columns.

Listing 20-60 shows the code for the getResultSetMetaData() method.

Listing 20-60 The getResultSetMetaData() method

```

private IResultSetMetaData getResultSetMetaData
( String queryText, IConnection conn )
throws OdaException
{
    java.util.Properties prop =
        new java.util.Properties();
    if( m_hibconfig == null)m_hibconfig = "";
    if( m_mapdir == null)m_mapdir = "";
    prop.put( "HIBCONFIG", m_hibconfig );
    prop.put( "MAPDIR", m_mapdir );
    conn.open( prop );
    IQuery query = conn.newQuery( null );
    query.prepare( queryText );
    return query.getMetaData();
}

```

- **setResultSetMetaData()**

The savePage() method calls the setResultSetMetaData() method when saving the report design. This method uses the DataSetDesign and the ResultSetMetaData objects for the query to create the columns in the data set for use in the report design.

Listing 20-61 shows the code for the setResultSetMetaData() method.

Listing 20-61 The setResultSetMetaData() method

```

private void setResultMetaData
( DataSetDesign dataSetDesign,
  IResultSetMetaData md ) throws OdaException
{
    ResultSetColumns columns =
        DesignSessionUtil.toResultSetColumnsDesign( md );
    ResultSetDefinition resultSetDefn =

```

```

        DesignFactory.eINSTANCE
            .createResultSetDefinition( );
        resultSetDefn.setResultSetColumns( columns );
        dataSetDesign.setPrimaryResultSet( resultSetDefn );
        dataSetDesign getResultSets().setDerivedMetaData( true );
    }
}

```

- **collectDataSetDesign()**

The plug-in calls this method when creating or modifying the query finishes. The plug-in passes the current design to this method. `collectDataSetDesign()` then verifies that a query exists and sets the design query to the value of the query text. The `savePage()` method saves the design and creates the columns in the data set.

Listing 20-62 shows the code for the `collectDataSetDesign()` method.

Listing 20-62 The `collectDataSetDesign()` method

```

protected DataSetDesign collectDataSetDesign
    ( DataSetDesign design )
{
    if( ! hasValidData( ) )
        return design;
    design.setQueryText( queryText.getText( ) );
    savePage( design );
    return design;
}

```

Building the Hibernate ODA UI plug-in

To build and deploy the `org.eclipse.birt.report.data.oda.hibernate.ui` plug-in using the Eclipse PDE Manifest Editor, perform the following tasks:

- On Build, specify the Build Configuration to include the following items:
 - In Runtime Information, add the `hibernateodaui.jar` file.
 - In Binary Build, select the following files and folders:
 - `META-INF`
 - `plugin.xml`
- Build Configuration appears, as shown in Figure 20-27.
- On Overview, in Exporting, choose Export Wizard and perform the following tasks:
 - In Available Plug-ins and Fragments, select `org.eclipse.birt.report.data.oda.hibernate.ui`.
 - In Options, verify that Package plug-ins as individual JAR archives is not selected.

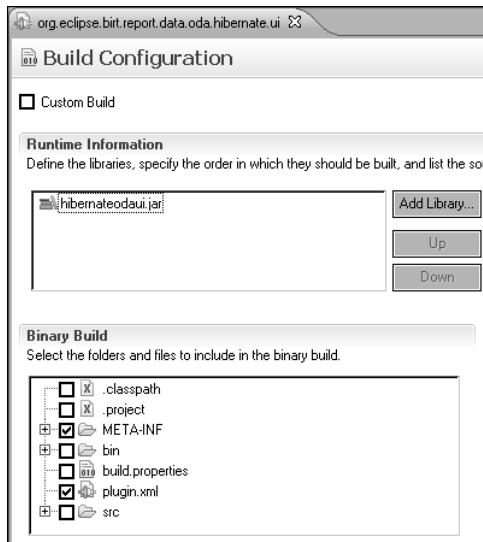


Figure 20-27 Build Configuration settings

- In Destination, choose the directory, \$INSTALL_DIR\birt-runtime-2_2_1\Report Engine, as shown in Figure 20-28.

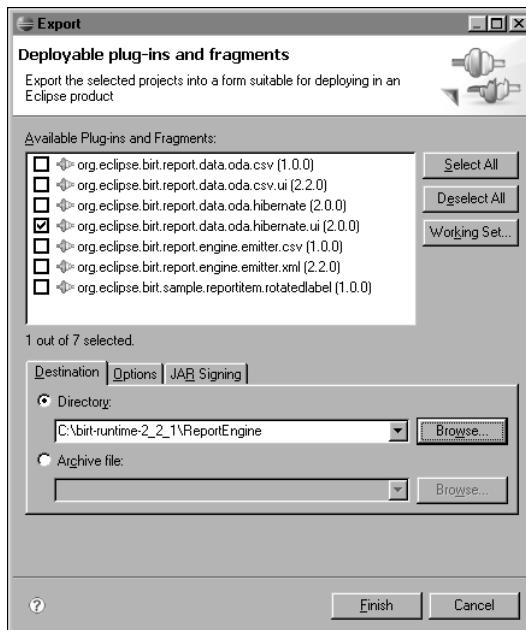


Figure 20-28 Using the Export wizard

Testing the Hibernate ODA UI plug-in

You can test the Hibernate ODA UI plug-in using a run-time instance of the Eclipse PDE workbench.

How to launch the Hibernate ODA UI plug-in

- 1 From the Eclipse SDK menu, choose Run—Open Run Dialog. On Run, right-click Eclipse Application. Choose New.
- 2 Create a configuration to launch an Eclipse application by performing the following tasks:

- 1 In Name, type:

Hibernate ODA Test

- 2 On Main, in Location, type:

C:\BIRT2.2.1_Test\runtime-TestODA\testHibernateODA

Run appears as shown in Figure 20-29.

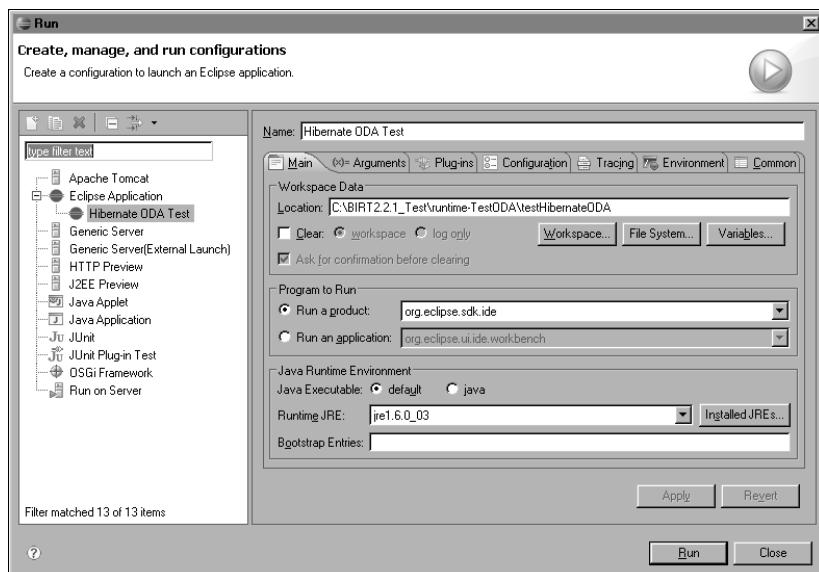


Figure 20-29 Creating a configuration to launch an Eclipse application

- 3 Choose Run to launch the run-time workbench.
- 4 In the run-time workbench, choose the Report Design perspective.
- 5 In Report Design, create a new report project and create a new blank report.

How to specify a data source and data set

- 1 In Report Design, choose Data Explorer. Data Explorer appears.

- 2** In Data Explorer, right-click Data Sources and choose New Data Source, as shown in Figure 20-30. New Data Source appears.



Figure 20-30 Choosing New Data Source

On New Data Source, choose Create from a data source type in the following list and select Hibernate Data Source as the data source type, as shown in Figure 20-31. Choose Next.

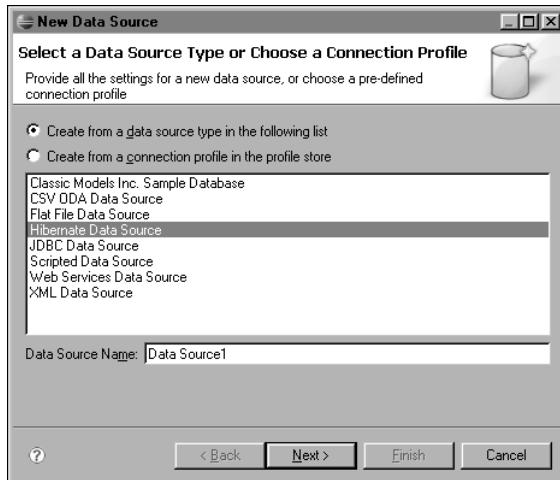


Figure 20-31 Selecting Hibernate Data Source

Hibernate Data Source appears, as shown in Figure 20-32. On Hibernate Data Source, select the Hibernate configuration file and mapping directory or leave these items blank if you use the hibfiles directory. Choose Finish.

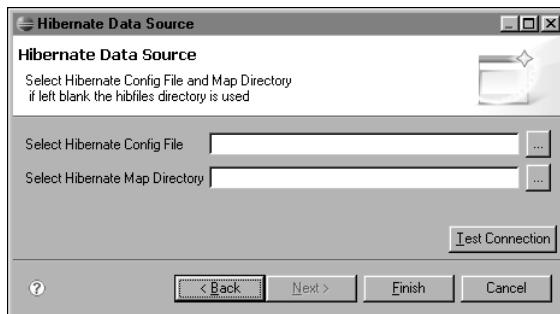


Figure 20-32 Configuring the Hibernate Data Source

Data Explorer appears with the new data source in Data Sources, as shown in Figure 20-33.



Figure 20-33 New data source in Data Explorer

- 3** In Data Explorer, right-click Data Sets and choose New Data Set, as shown in Figure 20-34.

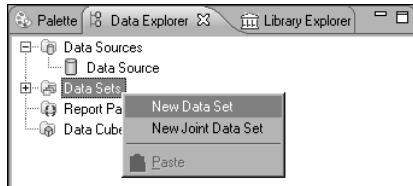


Figure 20-34 Choosing New Data Set

New Data Set appears, as shown in Figure 20-35.



Figure 20-35 New Data Set

Choose Next. Hibernate Data Set appears.

- 4** On Edit Data Set, perform the following tasks:

- 1** In Enter HQL and Verify Query, type:

```
select ord.orderNumber, cus.customerNumber,  
      cus.customerName  
  from Orders as ord, Customer as cus  
 where ord.customerNumber = cus.customerNumber  
   and cus.customerNumber = 363
```

Edit Data Set displays the query, as shown in Figure 20-36.

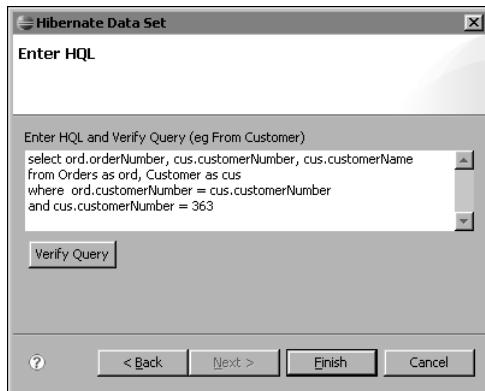


Figure 20-36 Editing the HQL query

- 2 Choose Verify Query.
- 3 Choose Finish. Edit Data Set appears. Choose Preview Results. Preview Results appears as shown in Figure 20-37.

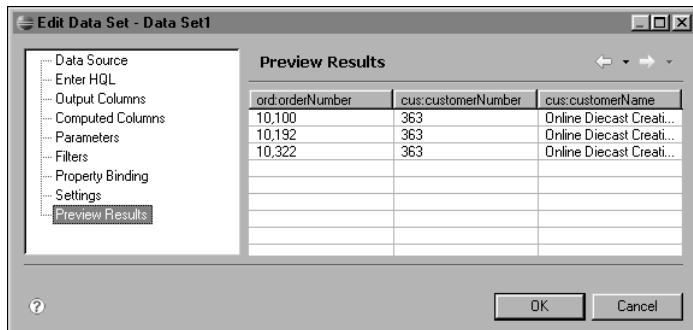


Figure 20-37 Previewing the data set

Choose OK. Data Explorer appears.

- 5 On Data Explorer, expand Data Sets. The new data set lists three columns, as shown in Figure 20-38.

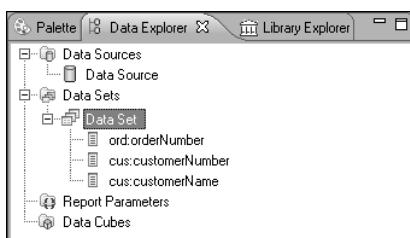


Figure 20-38 Data set in Data Explorer

- 6 To build a report that uses the data set, perform the following tasks:

- 1 On Data Explorer, drag Data Set to the layout editor. The layout appears as shown in Figure 20-39.

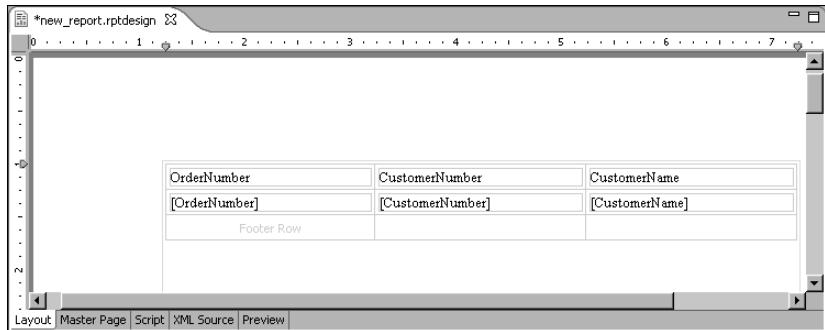


Figure 20-39 Report design in the layout editor

- 2 To view the output for new_report_1.rptdesign, choose Preview. The Preview appears as shown in Figure 20-40.

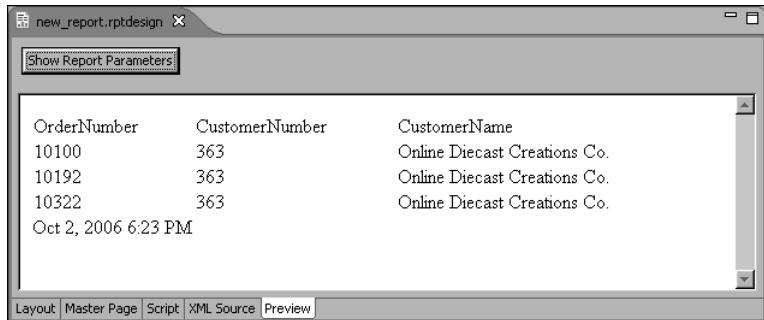


Figure 20-40 Preview of the report design

21

Developing a Fragment

The BIRT Report Engine environment supports plug-in fragments. A plug-in fragment is a separately loaded package that adds functionality to an existing plug-in, such as a specific language feature in a National Language Support (NLS) localization application. The example in this chapter creates a Java resource bundle that adds translations to the messages defined in the messages.properties files for the org.eclipse.birt.report.viewer plug-in.

Understanding a fragment

A fragment does not define its own plugin.xml file or a plug-in class. The related plug-in controls all processing. A fragment loads along the classpath of the related plug-in, providing access to all classes in the plug-in package.

A fragment inherits all the resources specified in the requires element of the plug-in manifest. A fragment can also specify additional libraries, extensions, and other resources.

The fragment's optional manifest file, fragment.xml, contains the attributes that associate the fragment with the plug-in. A fragment.xml file can specify the following tags and associated attributes:

- <fragment>

Specifies the following attributes:

- name

Display name of the extension.

- id

Unique identifier for the fragment extension.

- **plugin-id**
The plug-in associated with the fragment.
- **version**
Version of the fragment extension, such as 2.2.1.
- **type**
Whether the fragment contains code or a resource file. The default is code.
- **<runtime>**
Specifies a list of one or more libraries required by the fragment runtime. The name attribute for the <library> element can specify an archive, directory, or substitution variable.

Developing the sample fragment

The fragment example in this chapter creates an XML specification that loads additional messages.properties files that contain the translations of messages in the resource bundle for the org.eclipse.birt.report.viewer plug-in. This section describes the steps required to implement the sample org.eclipse.birt.report.viewers.nl1 project, using the Eclipse Plug-in Development Environment (PDE).

A National Language 1 (NL1) package or pack contains the translations for most major European and Asian languages. To implement the sample NL1 fragment, perform the following tasks:

- Configure the fragment project.

You can build the sample fragment plug-in project by following the instructions in this chapter.

- Add the translations contained in the message properties files to the org.eclipse.birt.report.resource bundle.

The name of each message properties file uses the following pattern:

`Messages_<lower-case language symbol>
_<upper-case language symbol>.msg`

For example, Messages_es_ES.msg indicates the message properties file for Spanish or Español.

- Build, deploy, and test the fragment.

You can build the fragment and export the fragment package from your workspace to the eclipse\plugins folder. You can test the fragment by starting Eclipse using a specific language setting and the creating a report in the BIRT report designer.

You can download the source code for the org.eclipse.birt.report.viewers.nl1 fragment example at <http://www.actuate.com/birt/contributions>.

It is not necessary to create a custom BIRT National Language Support (NLS) plug-in for most major languages. Eclipse supplies the following BIRT language packs:

- NLpack1
Contains the NL fragments and features for German, Spanish, French, Italian, Japanese, Korean, Portuguese (Brazil), Traditional Chinese and Simplified Chinese.
- NLpack2
Contains the NL fragments and features for Czech, Hungarian, Polish and Russian.
- NLpack2a
Contains the NL fragments and features for Danish, Dutch, Finnish, Greek, Norwegian, Portuguese, Swedish and Turkish.
- NLpackBidi
Contains the NL fragments and features for Arabic and Hebrew. Hebrew is only available for Eclipse run time, GEF run time and EMF run time.

You can download these packs from <http://www.eclipse.org/downloads>.

Creating a fragment project

You can create the fragment project for the NL1 fragment sample in the Eclipse PDE.

How to create the fragment project

- 1 In the Eclipse PDE, choose File->New->Project. New Project appears.
- 2 In Select a wizard, select Fragment Project. Choose Next. New Fragment Project appears.
- 3 In Fragment Project modify the settings, as shown in Table 21-1.

Table 21-1 Settings for Fragment Project fields

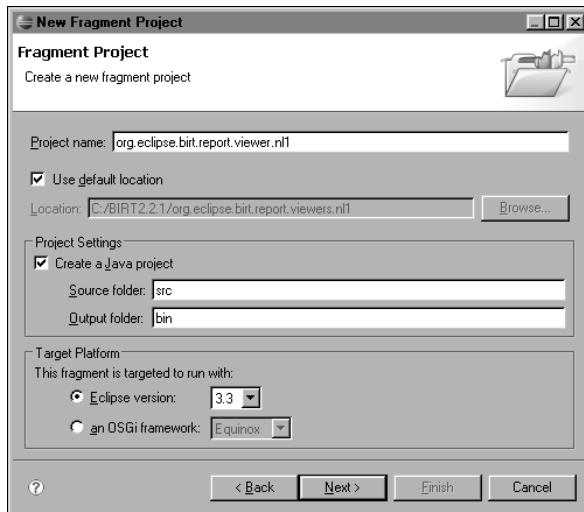
Section	Option	Value
Fragment Project	Project name	org.eclipse.birt.report.viewer.nl1
	Use default location	Selected
	Location	Not available when you select Use default location.

(continues)

Table 21-1 Settings for Fragment Project fields (*continued*)

Section	Option	Value
Project Settings	Create a Java project	Selected
	Source folder	src
	Output folder	bin
Target Platform	Eclipse version	3.3
	OSGi framework	Deselected

Fragment Project appears, as shown in Figure 21-1.

**Figure 21-1** Fragment Project settings

Choose Next. Fragment Content appears.

- 4 In Fragment Content, modify the settings as shown in Table 21-2.

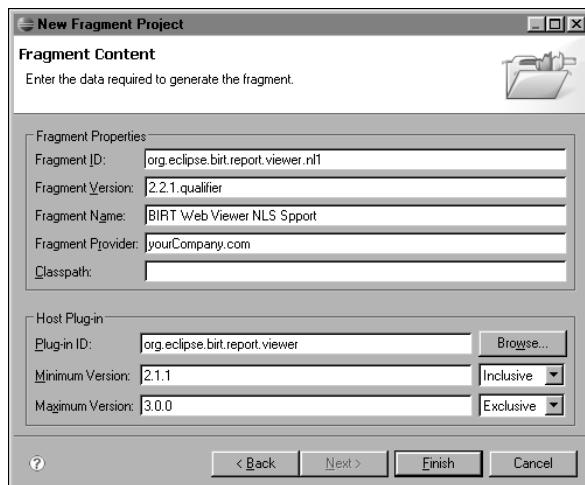
Table 21-2 Fragment Content settings

Section	Option	Value
Fragment Properties	Fragment ID	org.eclipse.birt.report.viewer.nl1
	Fragment Version	2.2.1.qualifier
	Fragment Name	BIRT Web Viewer NLS Support
	Fragment Provider	yourCompany.com or leave blank
	Classpath	Leave blank

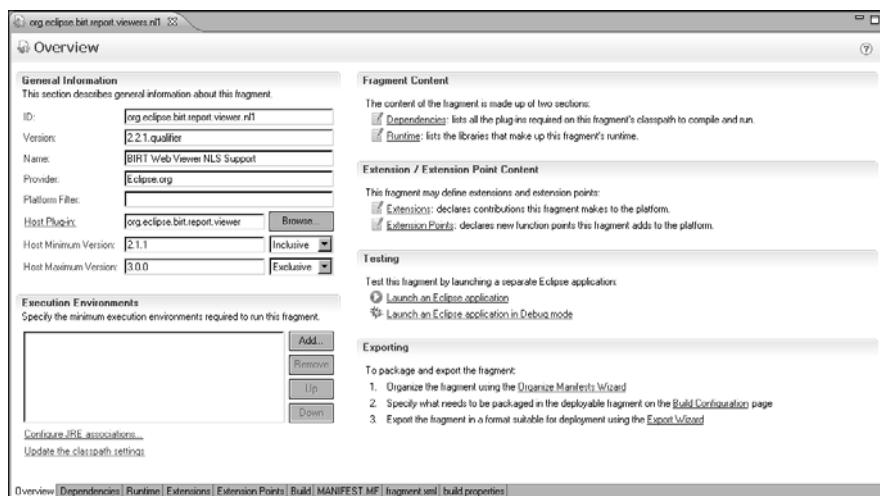
Table 21-2 Fragment Content settings (*continued*)

Section	Option	Value
Host plug-in	Plug-in ID	org.eclipse.birt.report.viewer
	Minimum Version	2.1.1 Inclusive
	Maximum Version	3.0.0 Exclusive

New Fragment Content appears, as shown in Figure 21-2. Choose Finish.

**Figure 21-2** Fragment Content settings

The fragment project appears in the Eclipse PDE Workbench, as shown in Figure 21-3.

**Figure 21-3** Fragment project in the Eclipse PDE Workbench

Understanding the sample fragment

The fragment provides the functionality required at run-time to display the messages seen in the BIRT Report Viewer in alternative languages. The fragment implements NL1 messages.properties files in org.eclipse.birt.report.resource.

Listing 21-1 shows an excerpt from the Spanish language version of BIRT Report Viewer messages from the file, Messages_es_ES.msg.

Listing 21-1 Contents of Messages_es_ES.msg

```
#####
birt.viewer.title=BIRT Report Viewer
birt.viewer.title.navigation=Navegación
birt.viewer.title.error=Error
birt.viewer.title.complete=Completado

birt.viewer.parameter=Parámetro
birt.viewer.runreport=Ejecutar informe
birt.viewer.required=Los parámetros marcados con
<FONT COLOR="red">*</FONT> son obligatorios.

birt.viewer.viewinpdf=Ver en PDF
birt.viewer.maximize=Ocultar parámetros de informe
birt.viewer.restore=Mostrar parámetros de informe

birt.viewer.error=Mensaje de error
birt.viewer.error.noparameter=No hay ningún parámetro para este
informe.
birt.viewer.error.noprinter=No se puede encontrar ninguna
impresora disponible compatible con el formato postscript.

#####
# Toolbar
#####
birt.viewer.toolbar.print=Imprimir informe como PDF
birt.viewer.toolbar.printserver=Imprimir informe en el servidor
birt.viewer.toolbar.toc=Mostrar tabla de contenido
birt.viewer.toolbar.parameter=Ejecutar informe
birt.viewer.toolbar.export=Exportar datos
birt.viewer.toolbar.font=Cambiar fuente
birt.viewer.toolbar.enableiv=Llamar a Interactive Viewer
birt.viewer.toolbar.exportreport=Exportar informe
...
#####
# General exception
#####
```

```

birt.viewer.generalException.DOCUMENT_FILE_ERROR=El archivo de
documento: {0} no existe o contiene errores.
birt.viewer.generalException.DOCUMENT_ACCESS_ERROR=No se puede
acceder al archivo de documento: {0}.
birt.viewer.generalException.REPORT_FILE_ERROR=El archivo de
informe: {0} no existe o contiene errores.
birt.viewer.generalException.REPORT_ACCESS_ERROR=No se puede
acceder al archivo de informe: {0}.
birt.viewer.generalException.DOCUMENT_FILE_PROCESSING=Procesando
el archivo de documento, inténtelo más tarde.
...
#####
# Birt Viewer JSP Taglib
#####
birt.viewer.taglib.NO_ATTR_ID=Se debe especificar el ID de
atributo.
birt.viewer.taglib.INVALID_ATTR_ID=El ID de atributo contiene
caracteres no válidos.
birt.viewer.taglib.ATTR_ID_DUPLICATE=El ID de atributo debe ser
exclusivo.
birt.viewer.taglib.NO_REPORT_SOURCE=Se debe especificar el
diseño de informe o archivo de documento.
birt.viewer.taglib.NO_REPORT_DOCUMENT=El informe necesita el
archivo de documento de informe.
birt.viewer.taglib.NO_REQUESTER_NAME=Se debe especificar un
nombre de solicitante

```

Building, deploying, and testing a fragment

Build the fragment after generating and modifying the build.xml file to specify the conversion of .msg files from native to ASCII format as .properties files. Listing 21-2 shows the contents of the build.xml file with the NativeToAscii specification.

Listing 21-2 Contents of build.xml file

```

<?xml version="1.0" encoding="UTF-8"?>
<project
    name="org.eclipse.birt.report.viewer.nl" default="Jar"
    basedir=".">
    <description>
        NL Fragment for org.eclipse.birt.report.viewer
    </description>
    <property file="META-INF/MANIFEST.MF" />
    <property name="dir.src" value="src" />
    <property name="dir.bin" value="bin" />
    <property name="nl.group" value="1" />
    <property name="module.name"
        value="org.eclipse.birt.report.viewer.nl" />
    <property name="jar.name"

```

```

    value=
        "${module.name}${nl.group}_${Bundle-Version}.jar" />
<target name="Clean">
    <delete>
        <fileset dir="${dir.src}"
            includes="**/*_??_?.properties" />
        <fileset dir=". " includes="${jar.name}" />
    </delete>
</target>
<target name="NativeToAscii"
    description="Execute native2ascii for *.msg files">
    <native2ascii encoding="Cp1252"
        src="${dir.src}"
        dest="${dir.src}"
        ext=".properties"
        includes="**/*_de_DE.msg,
        **/*_fr_FR.msg, **/*_es_ES.msg"/>
    <native2ascii encoding="GBK"
        src="${dir.src}"
        dest="${dir.src}"
        ext=".properties"
        includes="**/*_zh_CN.msg"/>
    <native2ascii encoding="SJIS"
        src="${dir.src}"
        dest="${dir.src}"
        ext=".properties"
        includes="**/*_ja_JP.msg"/>
    <native2ascii encoding="MS949"
        src="${dir.src}"
        dest="${dir.src}"
        ext=".properties"
        includes="**/*_ko_KR.msg"/>
</target>
<target name="nl-jar">
    <jar destfile="${jar.name}"
        manifest=".META-INF/MANIFEST.MF">
        <zippedfileset dir="${dir.src}"
            includes="**/*.properties"/>
        <fileset dir=". "
            includes="plugin_??_?.properties"/>
    </jar>
</target>
<target
    name="Jar" depends="NativeToAscii, nl-jar" >
</target>
<target name="Export">
    <copy todir="${export.dir}">
        <fileset dir=". " includes="${jar.name}"/>
    </copy>
</target>
</project>

```

If you set the fragment version number to 2.2.1.qualifier, the Eclipse PDE generates a JAR file with the following name:

`org.eclipse.birt.report.viewer.nl1_2.2.1.qualifier.jar`

Before building the fragment, change qualifier to the BIRT Report Viewer plug-in build number, such as v20070920. The Eclipse PDE generates a JAR file with the following name:

`org.eclipse.birt.report.viewer.nl1_2.2.1.v20070920.jar`

The Eclipse PDE provides support for deploying the plug-in in a run-time environment. To deploy the fragment to the BIRT Report Viewer example, use the Export wizard as shown in Figure 21-4 or manually copy the `org.eclipse.birt.report.viewer.nl1` JAR file from your workspace to the `eclipse/plugins` folder.

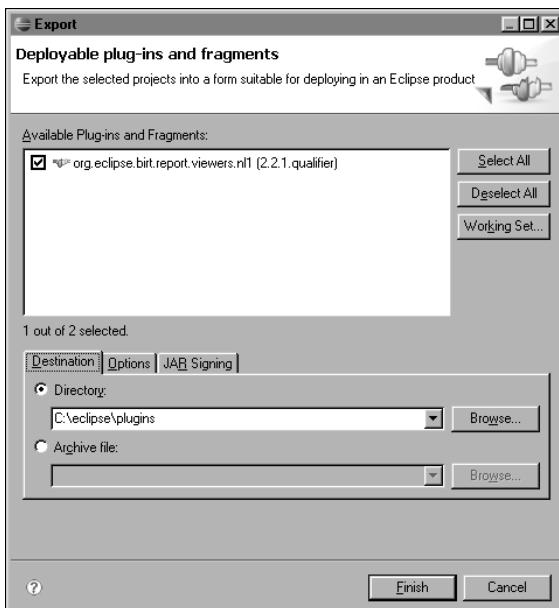


Figure 21-4 Deploying a fragment using the Export wizard

Test the fragment after deploying it by starting Eclipse using the `-nl` argument with the desired language setting. Listing 21-3 shows the command to start Eclipse using the `-nl` argument with the lower-case and upper case symbols for Spanish, as specified in the name of the Spanish messages.properties file, `Messages_es_ES.msg`.

Listing 21-3 Starting Eclipse using the `-nl` setting for Spanish

```
eclispe -nl es_ES
```

In Eclipse open BIRT Report Designer or in Spanish, Diseño de informe perspective, as shown in Figure 21-5.

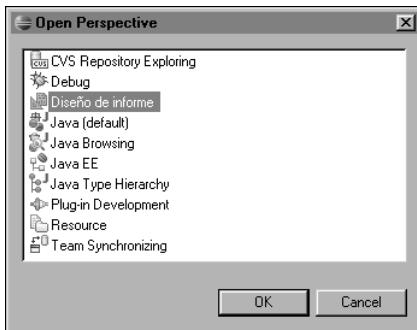


Figure 21-5 Opening the BIRT Report Designer perspective using Spanish NL1 settings

The BIRT Report Designer appears with the Spanish language settings specified in the NL1 language pack. Figure 21-6 shows the Palette with the names of the report items appearing in Spanish.

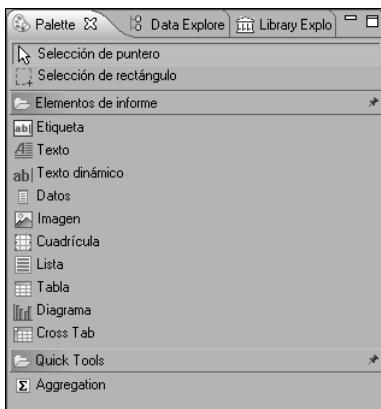


Figure 21-6 The Palette using Spanish NL1 settings

Figure 21-7 shows the Property Editor with the names of the Properties categories and the settings for General appearing in Spanish.

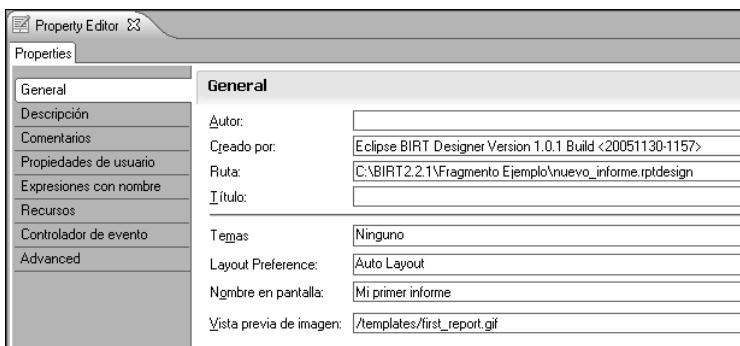


Figure 21-7 The Property Editor using Spanish NL1 settings

Figure 21-8 shows the Editor with the names of the Editor tabs appearing in Spanish.

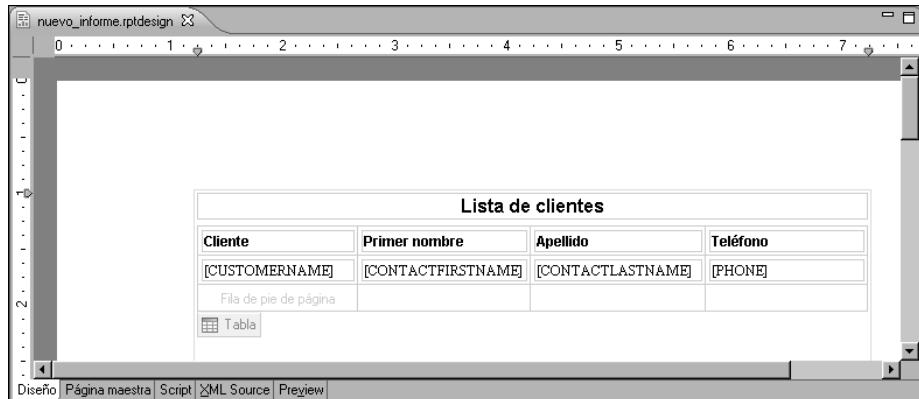


Figure 21-8 The Editor using Spanish NL1 settings

Run the report using BIRT Web Viewer by choosing View Report→View report in Web Viewer or, in Spanish, Ver informe en Web Viewer, as shown in Figure 21-9.

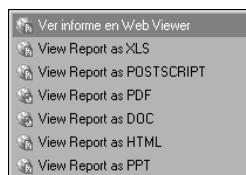


Figure 21-9 Running a report in Web Viewer using Spanish NL1 settings

Figure 21-10 shows the report with the BIRT Web Viewer page prompts, such as Showing page appearing in Spanish as Mostrando página.



Cliente	Primer nombre	Apellido	Teléfono
Atelier graphique	Carine	Schmitt	40.32.2555
Signal Gift Stores	Jean	King	7025551838
Australian Collectors, Co.	Peter	Ferguson	03 9520 4555
La Rochelle Gifts	Janine	Labrunie	40.67.8555
Baane Mini Imports	Jonas	Bergulfsen	07-98 9555
Mini Gifts Distributors Ltd.	Susan	Nelson	4155551450

Figure 21-10 The Web Viewer using Spanish NL1 settings

The NL1 language pack configures only elements in the BIRT Report Designer and Viewer user interfaces. For example, in Figure 21-8 and

Figure 21-10, the elements in the report design and output, such as the table title and column names, are user-defined and not part of the NL1 language pack configuration.

abstract base class

A class that organizes a class hierarchy or defines methods and variables that apply to descendant classes. An abstract base class does not support the creation of instances.

Related terms

class, class hierarchy, descendant class, method, variable

aggregate function

A function that performs a calculation over a set of data rows. For example, SUM calculates the sum of values of a specified numeric field over a set of data rows. Examples of aggregate functions include AVERAGE, COUNT, MAX, MIN, and SUM.

Related terms

data row, field, function, value

Contrast with

aggregate row, aggregate value

aggregate row

A single row that summarizes data from a group of rows returned by a query. A SQL query that includes an aggregate expression and a Group By clause returns aggregate rows. For example, a row that totals all orders made by one customer is an aggregate row.

Related terms

data, query, row, SQL (Structured Query Language), value

Contrast with

aggregate value, data row, SQL SELECT statement

aggregate value

The result of applying an aggregate function to a set of data rows. For example, consider a set of data rows that have a field, SPEED, which has values: 20, 10, 30, 15, 40. Applying the aggregate function MAX to dataSetRow("SPEED"), produces the aggregate value, 40, which is the maximum value for the field.

Related terms

aggregate function, data row, field, value

alias

- 1 In a SQL SELECT statement, an alternative name given to a database table or column.
- 2 An alternative name that is given to a table column for use in an expression or in code in a script method. This name must be a valid variable name that begins with a letter and contains only alphanumeric characters.

Related terms

column, expression, method, SQL SELECT statement, table, variable

Contrast with

display name

ancestor class

A class in the inheritance hierarchy from which a particular class directly or indirectly derives.

Related terms

class, inheritance

Contrast with

class hierarchy, descendant class, subclass, superclass

applet

A small desktop application that usually performs a simple task, for example, a Java program that runs directly from the web browser.

Related terms

application, Java

application

A complete, self-contained program that performs a specific set of related tasks.

Contrast with

applet

application programming interface (API)

A set of routines, including functions, methods, and procedures, that exposes application functionality to support integration and extend applications.

Related terms

application, function, method, procedure

argument

A constant, expression, or variable that supplies data to a function or method.

Related terms

constant, data, expression, function, method, variable

Contrast with

parameter

array A data variable that consists of sequentially indexed elements that have the same data type. Each element has a common name, a common data type, and a unique index number identifier. Changes to an element of an array do not affect other elements.

Related terms

data, data type, variable

assignment statement

A statement that assigns a value to a variable. For example:

```
StringToDisplay = "My Name"
```

Related terms

statement, value, variable

BIRT See Business Intelligence and Reporting Tools (BIRT).

BIRT extension

See Business Intelligence and Reporting Tools (BIRT) extension.

BIRT Report Designer

See Business Intelligence and Reporting Tools (BIRT) Report Designer.

BIRT technology

See Business Intelligence and Reporting Tools (BIRT) technology.

bookmark An expression that identifies a report element. A bookmark is used in a hyperlink expression.

Related terms

expression, hyperlink, report element

Boolean expression

An expression that evaluates to True or False. For example, Total > 3000 is a Boolean expression. If the condition is met, the condition evaluates to True. If the condition is not met, the condition evaluates to False.

Related term

expression

Contrast with

conditional expression, numeric expression

breakpoint

In BIRT Report Designer, a place marker in a program that is being debugged. At a breakpoint, execution pauses so that the report developer can examine and edit data values as necessary.

Related terms

Business Intelligence and Reporting Tools (BIRT) Report Designer, data, debug, value

bridge class

A class that maps the functionality of one class to the similar behavior of another class. For example, the JDBC-ODBC bridge class enables applications that use standard JDBC protocol to access a database that uses the ODBC protocol. BIRT Report Designer and BIRT RCP Report Designer use this type of class.

Related terms

application, Business Intelligence and Reporting Tools (BIRT) Report Designer, Business Intelligence and Reporting Tools (BIRT) Rich Client Platform (RCP) Report Designer, class, Java Database Connectivity (JDBC), open database connectivity (ODBC), protocol

Business Intelligence and Reporting Tools (BIRT)

A reporting platform that is built on the Eclipse platform, the industry standard for open source software development. BIRT provides a complete solution for extracting data, processing data to answer business questions, and presenting the results in a formatted document that is meaningful to end users.

Related terms

data, Eclipse, report

Contrast with

Business Intelligence and Reporting Tools (BIRT) extension

Business Intelligence and Reporting Tools (BIRT) Chart Engine

A tool that supports designing and deploying charts outside a report design. Using this engine, Java developers embed charting capabilities into an application. BIRT Chart Engine is a set of Eclipse plug-ins and Java archive (.jar) files. The chart engine is also known as the charting library.

Related terms

application, Business Intelligence and Reporting Tools (BIRT), chart, design, Eclipse platform, Java, Java archive (.jar) file, plug-in, report

Contrast with

Business Intelligence and Reporting Tools (BIRT) Report Engine

Business Intelligence and Reporting Tools (BIRT) Demo Database

A sample database that is used in tutorials in online help for BIRT Report Designer and BIRT RCP Report Designer. This package provides this demo database in Derby, Microsoft Access, and MySQL formats.

Related terms

Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) Report Designer, Business Intelligence and Reporting Tools (BIRT) Rich Client Platform (RCP) Report Designer

Business Intelligence and Reporting Tools (BIRT) extension

A related set of extension points that adds custom functionality to the BIRT platform. BIRT extensions include

- Charting extension
- Open data access (ODA) extension
- Rendering extension
- Report item extension

Related terms

Business Intelligence and Reporting Tools (BIRT), charting extension, extension, extension point, rendering extension, report, report item extension

Business Intelligence and Reporting Tools (BIRT) Report Designer

A tool that builds BIRT report designs and previews reports that are generated from the designs. BIRT Report Designer is a set of plug-ins to the Eclipse platform and includes BIRT Chart Engine, BIRT Demo Database, and BIRT Report Engine. A report developer who uses this tool can access the full capabilities of the Eclipse platform.

Related terms

Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) Chart Engine, Business Intelligence and Reporting Tools (BIRT) Demo Database, Business Intelligence and Reporting Tools (BIRT) Report Engine, design, Eclipse platform, plug-in, report

Contrast with

Business Intelligence and Reporting Tools (BIRT) Rich Client Platform (RCP) Report Designer

Business Intelligence and Reporting Tools (BIRT) Report Engine

A component that supports deploying BIRT charting, reporting, and viewing capabilities as a stand-alone application or on an application server. BIRT Report Engine consists of a set of Eclipse plug-ins, Java archive (.jar) files, web archive (.war) files, and web applications.

Related terms

application, Business Intelligence and Reporting Tools (BIRT), Java archive (.jar) file, plug-in, report, web archive (.war) file

Contrast with

Business Intelligence and Reporting Tools (BIRT) Chart Engine

Business Intelligence and Reporting Tools (BIRT) Rich Client Platform (RCP) Report Designer

A stand-alone tool that builds BIRT report designs and previews reports that are generated from the designs. BIRT RCP Report Designer uses the Eclipse Rich Client Platform. This tool includes BIRT Report Engine, BIRT Chart Engine, and BIRT Demo Database. BIRT RCP Report Designer supports report design and preview functionality without the additional overhead of the full Eclipse platform. BIRT RCP Report Designer does not support the Java-based scripting and the report debugger functionality

the full Eclipse platform provides. BIRT RCP Report Designer can use, but not create, BIRT extensions.

Related terms

Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) extension, Business Intelligence and Reporting Tools (BIRT) Chart Engine, Business Intelligence and Reporting Tools (BIRT) Demo Database, Business Intelligence and Reporting Tools (BIRT) Report Engine, debug, design, Eclipse platform, Eclipse Rich Client Platform (RCP), extension, JavaScript, library (.rptlibrary) file, plug-in, report

Contrast with

Business Intelligence and Reporting Tools (BIRT) Report Designer

Business Intelligence and Reporting Tools (BIRT) Samples

A sample of a BIRT report item extension and examples of BIRT charting applications. The report item extension sample is an Eclipse platform plug-in. The charting applications use BIRT Chart Engine. Java developers use these examples as models of how to design custom report items and embed charting capabilities in an application.

Related terms

application, Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) Chart Engine, chart, design, Eclipse, Eclipse platform, Java, plug-in, report, report item, report item extension

Business Intelligence and Reporting Tools (BIRT) technology

A set of Java applications and APIs that support the design and deployment of a business report. BIRT applications include BIRT Report Designer, BIRT RCP Report Designer, and a report viewer web application servlet. The BIRT Java APIs provide programmatic access to BIRT functionality.

Related terms

application, application programming interface (API), Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) Report Designer, Business Intelligence and Reporting Tools (BIRT) Rich Client Platform (RCP) Report Designer, Java, report viewer servlet

cascading parameters

Report parameters that have a hierarchical relationship. For example, the following parameters have a hierarchical relationship:

Country
State
City

In a group of cascading parameters, each report parameter displays a set of values. When a report user selects a value from the top-level parameter, the selected value determines the values that the next parameter displays,

and so on. Cascading parameters display only relevant values to the user. Figure G-1 shows cascading parameters as they appear to a report user.

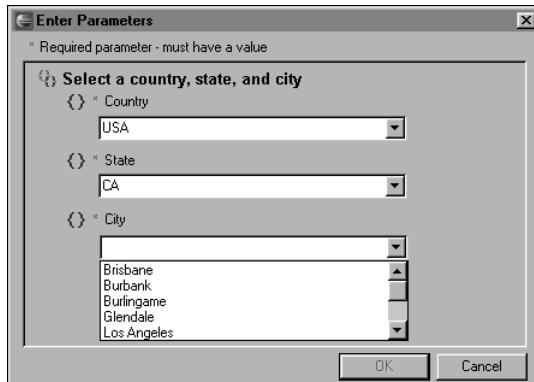


Figure G-1 Cascading parameters

Related terms

hierarchy, parameter, report, value

Contrast with

cascading style sheet (CSS)

cascading style sheet (CSS)

A file that contains a set of rules that attach formats and styles to specified HTML elements. For example, a cascading style sheet can specify the color, font, and size for an HTML heading.

Related terms

element, font, hypertext markup language (HTML), style

Contrast with

template

case sensitivity

A condition in which the letter case is significant for the purposes of comparison. For example, "McManus" does not match "MCMANUS" or "mcmanus" in a case-sensitive environment.

category

In an area, bar, line, step, or stock chart, one of the discrete values that organizes data on an axis that does not use a numerical scale. Typically, the x-axis of a chart displays category values. In a pie chart, category values are called orthogonal axis values and define which sectors appear in a pie.

Related terms

chart, data, value

Contrast with

series

cell

An intersection of a row and a column in a cross tab, grid element, or table element. Figure G-2 shows a cell.

Related terms

column, cross tab, grid element, row, table element

	Column 1	Column 2	Column 3
Row 1	Data	Data	Data
Row 2	Data	Data	Data
Row 3	Data	Data	Data
Row 4	Data	Data	Data

Figure G-2 Cell

character

An elementary mark that represents data, usually in the form of a graphic spatial arrangement of connected or adjacent strokes, such as a letter or a digit. A character is independent of font size and other display properties. For example, an uppercase C is a character.

Related term

data

Contrast with

character set, glyph

character set

A mapping of specific characters to code points. For example, in most character sets, the letter A maps to the hexadecimal value 0x21.

Related terms

character, code point

Contrast with

locale

chart

A graphic representation of data or the relationships among sets of data.

Related term

data

chart element

A report item that displays values from data rows in the form of a chart. The chart element can use data rows from the report design's data set or a different data set. A report item extension defines the chart element.

Related terms

chart, data, data row, data set, design, element, report, report item, report item extension, value

Contrast with

charting extension

chart engine

See Business Intelligence and Reporting Tools (BIRT) Chart Engine.

charting extension

A BIRT extension that adds a new type of chart, a new component to an existing chart type, or a new user interface component to the BIRT chart engine.

Related terms

Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) extension, Business Intelligence and Reporting Tools (BIRT) Chart Engine, chart, extension

charting library

See Business Intelligence and Reporting Tools (BIRT) Chart Engine.

class

A set of methods and variables that defines the attributes and behavior of an object. All objects of a given class are identical in form and behavior but can contain different data in their variables.

Related terms

data, method, object, variable

Contrast with

subclass, superclass

class hierarchy

A tree structure that represents the inheritance relationships among a set of classes.

Related terms

class, inheritance

class name

A unique name for a class that permits unambiguous references to its public methods and variables.

Related terms

class, method, variable

class variable

A variable that all instances of a class share. The run-time system creates only one copy of a class variable. The value of the class variable is the same for all instances of the class, for example, the taxRate variable in an Order class.

Related terms

class, object, value, variable

Contrast with

dynamic variable, field variable, global variable, instance variable, local variable, member variable, static variable

code point

A hexadecimal value. Every character in a character set is represented by a code point. The computer uses the code point to process the character.

Related terms

character, character set

- column** 1 A named field in a database table or query. For each data row, the column can have a different value, called the column value. The term column refers to the definition of the column, not to any particular value. Figure G-3 shows a column in a database table.

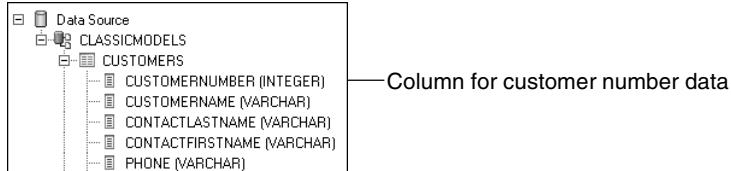


Figure G-3 Column in a database table

- 2 A vertical sequence of cells in a cross tab, grid element, or table element. Figure G-4 shows a column in a cross tab.

	Column 1	Column 2	Column 3
Row 1	Data	Data	Data
Row 2	Data	Data	Data
Row 3	Data	Data	Data
Row 4	Data	Data	Data

Figure G-4 Column in a cross tab

Related terms

cell, cross tab, data row, field, grid element, query, table, table element, value

column binding

A named column that defines an expression that specifies what data to return. For each piece of data to display in a report, there must be column binding. Column bindings form an intermediate layer between data-set data and report elements.

Related terms

column, data, data set, expression, report, report element

computed column

See computed field.

computed field

A field that displays the result of an expression rather than stored data.

Related terms

data, expression, field

Contrast with

computed value

computed value

The result of a calculation that is defined by an expression. To display a computed value in a report, use a data element.

Related terms

data element, expression, report, value

Contrast with

computed field

conditional expression

An expression that returns value A or value B depending on whether a Boolean expression evaluates to True or False.

Related term

Boolean expression, expression

configuration file

In open data access (ODA), a file that specifies the ODA interface version of the driver and defines the structure, contents, and semantics of requests and responses between the open data source and the design tool.

Related terms

data source, interface, open data access (ODA), open data access (ODA) driver, request, response

Connection

A Java object that provides access to a data source.

Related terms

data source, Java, object

constant

An unchanging, predefined value. A constant does not change while a program is running, but the value of a field or variable can change.

Related terms

field, value, variable

constructor code

Code that initializes an instance of a class.

Related terms

class, object

container

- 1 An application that acts as a master program to hold and execute a set of commands or to run other software routines. For example, application servers provide containers that support communication between applications and Enterprise JavaBeans.
- 2 A data structure that holds one or more different types of data. For example, a grid element can contain label elements and other report items.

Related terms

application, data, Enterprise JavaBean (EJB), grid element, label element, report item

containment

A relationship among instantiated objects in a report. One object, the container, incorporates other objects, the contents.

Related terms

container, instantiation, object, report

containment hierarchy

A hierarchy of objects in a report.

Related terms

hierarchy, object, report

converter

A tool that converts data from one format to another format.

Related terms

data, format

cross tab

A report that summarizes data from database table columns into a concise format for analysis. Data appears in a matrix with rows and columns. Every cell in a cross tab contains an aggregate value. A cross tab shows how one item relates to another, such as order totals by credit rating and order status. Figure G-5 shows a cross tab.

Order Totals by Credit Rating and Order Status

	A			B			C			Totals		
	Quantity	Ave Qty	Amount	Quantity	Ave Qty	Amount	Quantity	Ave Qty	Amount	Quantity	Ave Qty	Amount
Open	98,205	349	9,870,799	40,472	283	4,673,517	4,980	332	638,565	143,657	327	16,180,871
Closed	171,813	365	20,633,175	131,183	497	14,465,314	21,790	198	2,591,731	324,786	384	37,890,220
In Evaluation	90,969	469	12,135,326	22,168	411	2,596,936	0	0	0	113,137	456	14,732,262
Cancelled	23,944	855	3,029,520	0	0	0	0	0	0	23,944	855	3,029,520
Selected	10,540	527	1,375,204	0	0	0	0	0	0	10,540	527	1,375,204
Totals	395,471	398	47,044,024	193,823	420	21,735,767	26,770	214	3,228,286	616,064	390	72,008,077

Figure G-5 Cross tab

Related terms

aggregate value, cell, column, data, grid, report, row, table, value

Contrast with

aggregate function

cross-tab element

A report item that displays a cross tab. A cross tab displays aggregate values in a matrix with rows and columns.

Related terms

aggregate value, cross tab, report item

CSS

See cascading style sheet (CSS).

cube A multidimensional data structure that provides multiple measures and dimensions to access and analyze large quantities of data. BIRT uses a cube to structure data for display in a cross-tab element.

Related terms

cross-tab element, data, dimension, measure, multidimensional data

custom data source

See open data access (ODA).

data Information that is stored in databases, flat files, or other data sources that can appear in a report.

Related terms

data source, flat file, report

Contrast with

metadata

data element



A report item that displays a computed value or a value from a data set field.

Related terms

computed value, data set, element, field, report item, value

Contrast with

label element, Report Object Model (ROM) element, text element

Data Explorer

An Eclipse view that shows the data sources, data sets, report parameters, and data cubes that were created for use in a report. Use Data Explorer to create, edit, or delete these items. Figure G-6 shows Data Explorer.

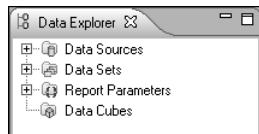


Figure G-6 Data Explorer

Related terms

cube, data set, data source, Eclipse view, parameter, report

data point A point on a chart that corresponds to a particular pair of x- and y-axis values.

Related terms

chart, value

Contrast with

data row, data set

data row One row of data that a data set returns. A data set, which specifies the data to retrieve from a data source, typically returns many data rows.

Related terms

data, data set, data source, row

Contrast with

data point, filter

data set

A description of the data to retrieve or compute from a data source.

Related terms

data, data source

Contrast with

data point, data row

data set field

See column.

data set parameter

A parameter that is associated with a data set column and passes an expression to extend dynamically the query's WHERE clause. A data set parameter restricts the number of data rows that the data set supplies to the report.

Related terms

column, data row, data set, expression, parameter, query, report

Contrast with

report parameter

data source

- 1 A SQL database or other repository of data also known as an input source. For example, a flat file can be a data source.
- 2 An object that contains the connection information for an external data source, such as a flat file, a SQL database, or another repository of data.

Related terms

data, flat file, SQL (Structured Query Language)

Contrast with

data row, data set

data type

- 1 A category for values that determines their characteristics, such as the information they can hold and the permitted operations.
- 2 The data type of a value determines the default appearance of the value in a report. This appearance depends on the locale in which a user generates the report. For example, the order in which year, month, and day appear in a date-and-time data value depends on the locale. BIRT uses three fundamental data types: date-and-time, numeric, and string. Data sources such as relational databases support more data types, which BIRT maps to the appropriate fundamental data type.

Related terms

Business Intelligence and Reporting Tools (BIRT), date-and-time data type, locale, numeric data type, report, String data type, value, variable

database connection

See data source.

database management system (DBMS)

Software that organizes simultaneous access to shared data. Database management systems store relationships among various data elements.

Related term

data

database schema

See schema.

date-and-time data type

A data type for date-and-time calculations. Report items can contain expressions or fields with a date-and-time data type. The appearance of date-and-time values in the report document is based on locale and format settings specified by your computer and the report design.

Related terms

data type, design, expression, field, format, locale, report, report item

debug

To detect, locate, and fix errors. Typically, debugging involves executing specific portions of a computer program and analyzing the operation of those portions.

declaration

The definition of a class, constant, method, or variable that specifies the name and, if appropriate, the data type.

Related terms

class, constant, data type, method, variable

derived class

See descendant class.

descendant class

A class that is based on another class.

Related term

class

Contrast with

subclass, superclass

design

- 1 To create a report specification. Designing a report includes selecting data, laying out the report visually, and saving the layout in a report design file.
- 2 A report specification. A report design (.rptdesign) file contains a report design.

Related terms

data, layout, report, report design (.rptdesign) file

DHTML (dynamic hypertext markup language)

See dynamic hypertext markup language (DHTML).

dimension In a cube, fields such as products, customers, or orders, by which measures are aggregated.

Related terms

cube, field, measure

Contrast with

multidimensional data

display name

An alternative name for a table column, report parameter, chart series, or user-defined ROM property. BIRT Report Designer displays this alternative name in the user interface, for example, as a column heading in a report. This name can contain any character, including spaces and punctuation.

Related terms

Business Intelligence and Reporting Tools (BIRT) Report Designer, character, chart, column, Data Explorer, report, report parameter, Report Object Model (ROM), table, value

Contrast with

alias

document object model (DOM)

A model that defines the structure of a document such as an HTML or XML document. The document object model defines interfaces that dynamically create, access, and manipulate the internal structure of the document. The URL to the W3C document object model is

www.w3.org/DOM/

Related terms

extensible markup language (XML), hypertext markup language (HTML), interface, Uniform Resource Locator (URL), World Wide Web Consortium (W3C)

Contrast with

document type definition (DTD), structured content

document type definition (DTD)

A set of markup tags and the interpretation of those tags that together define the structure of an XML document.

Related terms

extensible markup language (XML), tag

Contrast with

document object model (DOM), schema, structured content

domain name

A name that defines a node on the internet. For example, the Eclipse Foundation's domain name is eclipse. The URL is

www.eclipse.org

Related terms

node, Uniform Resource Locator (URL)

driver

An interface that supports communication between an application and another application or a peripheral device such as a printer.

Related term

interface

dynamic hypertext markup language (DHTML)

An HTML extension that provides enhanced viewing capabilities and interactivity in a web page without the necessity for communication with a web server. The Document Object Model Group of the W3C develops DHTML standards.

Related terms

document object model (DOM), hypertext markup language (HTML), web page, web server, World Wide Web Consortium (W3C)

dynamic text element

ab

A data element that displays text data that contains multiple style formats and a variable amount of text. A dynamic text element adjusts its size to accommodate varying amounts of data. Use a dynamic text element to display a data source field that contains formatted text. A dynamic text element supports plain or HTML text.

Related terms

data, data source, field, format, hypertext markup language (HTML)

Contrast with

text element

dynamic variable

A variable that changes during program execution. The program requests the memory allocation for a dynamic variable at run time.

Related term

variable

Contrast with

class variable, field variable, global variable, instance variable, local variable, member variable, static variable

Eclipse

An open platform for tool integration that is built by an open community of tool providers. The Eclipse platform is written in Java and includes extensive plug-in construction toolkits and examples.

Related terms

Eclipse platform, Java, plug-in

Contrast with

Business Intelligence and Reporting Tools (BIRT) extension

Eclipse Modeling Framework (EMF)

A Java framework and code generation facility for building tools and other applications that are based on a structured model. EMF uses XML schemas to generate the EMF model of a plug-in. For example, a BIRT chart type uses EMF to represent the chart structure and properties.

Related terms

application, Business Intelligence and Reporting Tools (BIRT) technology, chart, Eclipse, extensible markup language (XML), framework, Java, plug-in, property, schema

Eclipse perspective

A predefined layout of the Eclipse Workbench, including which Eclipse views are visible and where they appear. A perspective also controls what appears in certain menus and toolbars. A user can switch the perspective to work on a different task and can rearrange and customize a perspective to better suit a particular task. Figure G-7 shows the Eclipse Java perspective.

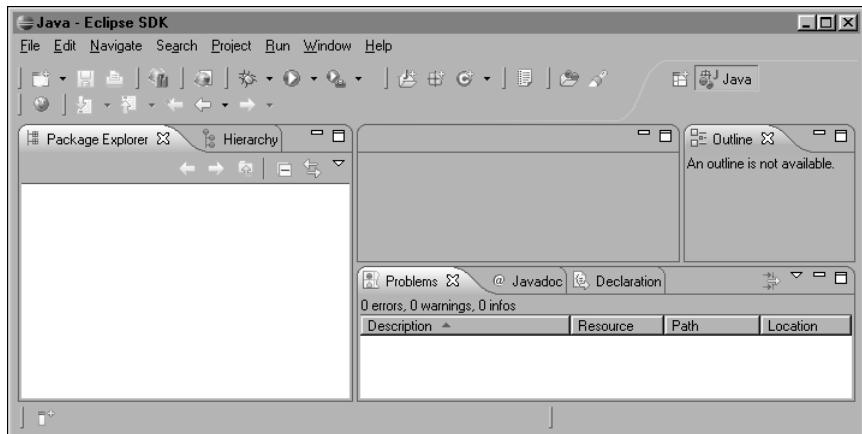


Figure G-7 Eclipse perspective

Related terms

Eclipse, Eclipse view, Eclipse Workbench

Eclipse platform

The core framework and services in which Eclipse plug-in extensions exist. The Eclipse platform provides the run-time environment in which plug-ins load and run. The Eclipse platform consists of a core component and a user interface component. The user interface component is known as the Eclipse Workbench. The core portion of the Eclipse platform is called the platform core or the core.

Related terms

Eclipse, Eclipse Workbench, extension, framework, plug-in

Eclipse Plug-in Development Environment (PDE)

An integrated design tool for creating, developing, testing, and deploying a plug-in. The Eclipse PDE provides wizards, editors, views, and launchers that support plug-in development. The Eclipse PDE supports host and run-time instances of a workbench project. The host instance provides the development environment. The run-time instance enables the launching of a plug-in for testing purposes.

Related terms

design, Eclipse, Eclipse project, Eclipse Workbench, object, plug-in

Eclipse project

A user-specified directory within an Eclipse workspace. An Eclipse project contains folders and files that are used for builds, version management, sharing, and resource organization.

Related terms

Eclipse, Eclipse workspace

Eclipse Rich Client Platform (RCP)

The Eclipse framework for building a client application that uses a minimal set of plug-ins. Eclipse Rich Client Platform (RCP) uses a subset of the components that are available in the Eclipse platform. An Eclipse rich client application is typically a specialized user interface that supports a specific function, such as the report development tools in the BIRT Rich Client Platform.

Related terms

application, Business Intelligence and Reporting Tools (BIRT), Eclipse, Eclipse platform, framework, plug-in, report

Eclipse view

A dockable window on the Eclipse Workbench, similar to a pane in Windows. An Eclipse view can be an editor, the Navigator, a palette of report items, a graphical report designer, or any other functional component that Eclipse or an Eclipse project provides. A view can have its own menus and toolbars. Multiple views can be visible at one time.

Related terms

design, Eclipse, Eclipse perspective, Eclipse project, Eclipse Workbench, Navigator, Palette, report, report item

Eclipse Workbench

The Eclipse desktop development environment, which supports the Eclipse perspectives.

Related terms

Eclipse, Eclipse perspective

Contrast with

Eclipse Plug-in Development Environment (PDE), Eclipse workspace

Eclipse workspace

A user-specified directory that contains one or more Eclipse projects. An Eclipse workspace is a general umbrella for managing resources in the Eclipse platform. The Eclipse platform can use one or more workspaces. A user can switch between workspaces.

Related terms

Eclipse platform, Eclipse project

Contrast with

Eclipse Workbench

EJB

See Enterprise JavaBean (EJB).

element

- 1 In Report Object Model (ROM), a component that describes a piece of a report. A ROM element typically has a name and a set of properties.
- 2 A tag-delimited structure in an XML or HTML document that contains a unit of data. For example, the root element of an HTML page starts with the beginning tag, <HTML>, and ends with the closing tag, </HTML>. This root element encloses all the other elements that define the contents of a page. An XML element must be well-formed, with both a beginning and a closing tag. In HTML, some tags, such as
, the forced line break tag, do not require a closing tag.

Related terms

data, extensible markup language (XML), hypertext markup language (HTML), property, report, Report Object Model (ROM), Report Object Model (ROM) element, tag, well-formed XML

Contrast with

report item

ellipsis button



A button that opens tools that you use to perform tasks, such as navigating to a file, building an expression, or specifying localized text.

encapsulation

A technique of packaging-related functions and subroutines together. Encapsulation compartmentalizes the structure and behavior of a class, hiding the implementation details, so that parts of an object-oriented system need not depend upon or affect each other's internal details.

Related terms

class, function, object

Contrast with

object-oriented programming

enterprise

A large collection of networked computers that run on multiple platforms. Enterprise systems can include both mainframes and

workstations that are integrated in a single, managed environment. Typical software products that are used in an enterprise environment include web browsers, applications, applets, web tools, and multiple databases that support a warehouse of information.

Related terms

applet, application

Contrast with

Enterprise JavaBean (EJB), enterprise reporting

Enterprise JavaBean (EJB)

A standards-based server-side component that encapsulates the business logic of an application. An EJB can provide access to data or model the data itself. Application servers provide the deployment environment for EJBs.

Related terms

application, data, JavaBean

Contrast with

Java

enterprise reporting

The production of a high volume of simple and complex structured documents that collect data from a variety of data sources. A large number of geographically distributed users who are working in both client/server and internet environments receive, work with, and modify these reports.

Related terms

data, data source, report

Contrast with

enterprise, structured content

event

An action or occurrence recognized by an object. Each object responds to a predefined set of events that can be extended by the developer.

Related term

object

Contrast with

event handler, event listener

event handler

A Java or JavaScript method that is executed upon the firing of a BIRT event. BIRT fires events at various times in the report generation process. By writing custom code for the associated event handlers, the BIRT report developer can provide special handling at the time the events are fired. Report items, data sets, and data sources all have event handlers for which the report developer can provide custom code.

Related terms

Business Intelligence and Reporting Tools (BIRT), data set, data source, event, Java, JavaScript, method, report, report item

Contrast with
event listener

event listener

An interface that detects when a particular event occurs and runs a registered method in response to that event.

Related terms

event, method

Contrast with

event handler

exception An abnormal situation that a program encounters. In some cases, the program handles the exception and returns a message to the user or application that is running the program. In other cases, the program cannot handle the exception, and the program terminates.

Related term

application

expression

A combination of constants, functions, literal values, names of fields, and operators that evaluates to a single value.

Related terms

constant, field, function, operator, value

Contrast with

regular expression

expression builder

A tool for selecting data fields, functions, and operators to write JavaScript expressions. Figure G-8 shows the expression builder.

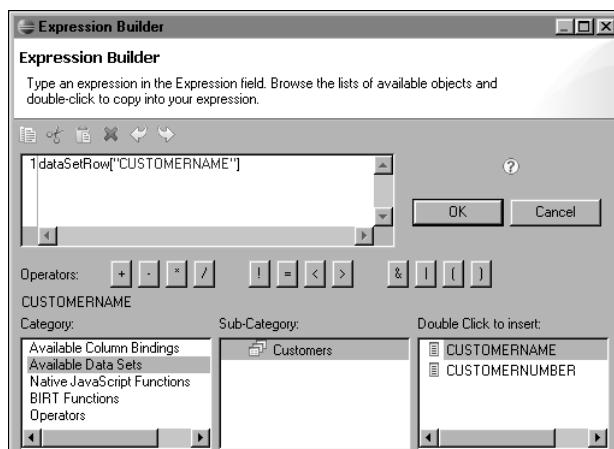


Figure G-8 Expression builder

Related terms

data, expression, field, function, JavaScript, operator

extensible markup language (XML)

A markup language that supports the interchange of data among data sources and applications. Using XML, a wide variety of applications, legacy systems, and databases can exchange information. XML is content-oriented rather than format-oriented. XML uses tags to structure data into nested elements. An XML schema that is structured according to the rules that were defined by the W3C describes the structure of the data. XML documents must be well-formed.

Related terms

application, data, data source, element, schema, tag, well-formed XML, World Wide Web Consortium (W3C)

Contrast with

hypertext markup language (HTML)

extension

A module that adds functionality to an application. BIRT consists of a set of extensions, called plug-ins, which add functionality to the Eclipse development environment.

Related terms

application, Business Intelligence and Reporting Tools (BIRT), Eclipse, plug-in

Contrast with

Business Intelligence and Reporting Tools (BIRT) extension, extension point

extension point

A defined place in an application where a developer adds custom functionality. The APIs in BIRT support adding custom functionality to the BIRT framework. In the Eclipse Plug-in Development Environment (PDE), a developer views the extension points in the PDE Manifest Editor to guide and control plug-in development tasks.

Related terms

application, application programming interface (API), Business Intelligence and Reporting Tools (BIRT), Eclipse Plug-in Development Environment (PDE), extension, framework, plug-in

Contrast with

Business Intelligence and Reporting Tools (BIRT) extension

field

The smallest identifiable part of a database table structure. In a relational database, a field is also called a column.

Related terms

column, table

field variable

In Java, a member variable with public visibility.

Related terms

Java, member, variable

Contrast with

class variable, dynamic variable, global variable, instance variable, local variable, member variable, static variable

file types

Table G-1 lists the report designer's file types.

Table G-1 File types

Display name	Glossary term	File type
BIRT Report Design	report design (.rptdesign) file	RPTDESIGN
BIRT Report Design Library	library (.rptlibrary) file	RPTLIBRARY
BIRT Report Design Template	report template (.rpttemplate) file	RPTTEMPLATE
BIRT Report Document	report document (.rptdocument) file	RPTDOCUMENT

filter

To exclude any data rows from the result set that do not meet a set of conditions. Some external data sources can filter data as specified by conditions that the query includes directly or through the use of report parameters. In addition, BIRT can filter data after retrieval from the external data source. Report developers can specify conditions for filtering in either the data set or a report item.

Related terms

Business Intelligence and Reporting Tools (BIRT), data, data row, data set, data source, query, report, report item, report parameter, result set

flat file

A file that contains data in the form of text.

Related term

data

Contrast with

data source

font

A family of characters of a given style. Fonts contain information that specifies typeface, weight, posture, and type size.

Related term

character

footer

1 A unit of information that appears at the bottom of a page.

2 A group footer is a unit of information that appears at the bottom of a group section.

Related terms

group, section

Contrast with
header

- format**
- 1 A set of standard options with which to display and print currency values, dates, numbers, and times.
 - 2 A specification that describes layout and properties of report data or other information, for example, PDF or HTML.

Related terms

data, hypertext markup language (HTML), layout, property, report, value

Contrast with
style

forms-capable browser

A web browser that handles hypertext markup language (HTML) forms. HTML tags enable interactive forms, including check boxes, drop-down lists, fill-in text areas, and option buttons.

Related terms

hypertext markup language (HTML), tag

- fragment** See plug-in fragment.

framework

A set of interrelated classes that provide an architecture for building an application.

Related terms

application, class

full outer join

See outer join.

- function** A sequence of instructions that are defined as a separate unit within a program. To invoke the function, include its name as one of the instructions anywhere in the program. BIRT provides JavaScript functions to support building expressions.

Related terms

Business Intelligence and Reporting Tools (BIRT), expression, JavaScript

Contrast with
method

global variable

A variable that is available at all levels in an application. A global variable stays in memory in the scope of all executing procedures until the application terminates.

Related terms

application, procedure, scope, variable

Contrast with

class variable, dynamic variable, field variable, instance variable, local variable, member variable, static variable

glyph

- 1 An image that is used for the visual representation of a character.
- 2 A specific letter form from a specific font. An uppercase C in Palatino font is a glyph.

Related terms

character, font

grandchild class

See descendant class.

grandparent class

See ancestor class.

grid

See grid element.

grid element

A report item that contains and displays other report elements in a static row and column format. A grid element aligns the cells horizontally and vertically.

Figure G-9 shows a report title section that consists of an image element and two text elements that are arranged in a grid element with one row and two columns.

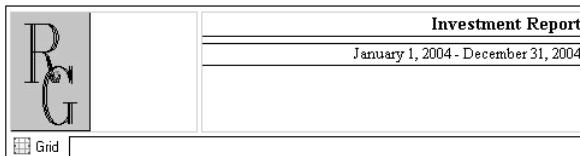


Figure G-9 Grid element

Related terms

cell, column, element, image element, report, report item, row, text element

Contrast with

list element, table element

group

A set of data rows that have one or more column values in common. For example, in a sales report, a group consists of all the orders that are placed by a single customer.

Related terms

column, data row, report, value

Contrast with

group key, grouped report

grouped report

A report that organizes data in logical groups. Figure G-10 shows a grouped report.

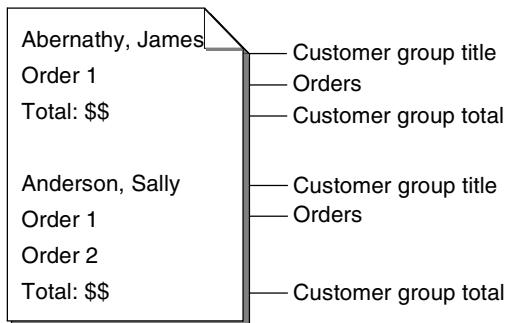


Figure G-10 Grouped report

Related terms

data, group, report

group key

A data set column that is used to group and sort data in a report. For example, a report developer can group and sort customers by credit rank.

Related terms

column, data, data set, group, report, sort

header

1 A unit of information that appears at the top of every page.

2 A group header is a unit of information that appears at the beginning of a group section.

Related terms

group, section

Contrast with

footer

hexadecimal number

A number in base 16. A hexadecimal number uses the digits 0 through 9 and the letters A through F. Each place represents a power of 16. By comparison, base 10 numbers use the digits 0 through 9. Each place represents a power of 10.

Contrast with

character set, octal number

hierarchy

Any tree structure that has a root and branches that do not converge.

HTML

See hypertext markup language (HTML).

HTML element

See element.

HTTP See hypertext transfer protocol (HTTP).

hyperlink



A connection from one part of a report to another part of the same or different report. Typically, hyperlinks support access to related information within the same report, in another report, or in another application. A change from the standard cursor shape to a cursor shaped like a hand indicates a hyperlink.

Related terms

application, report

hypertext markup language (HTML)

A specification that determines the layout of a web page. HTML is the markup language that tells a parser that the text is a certain portion of a document on the web, for example, the title, heading, or body text. A web browser parses HTML to display a web page.

Related terms

layout, web page

Contrast with

dynamic hypertext markup language (DHTML), extensible markup language (XML)

hypertext markup language page

See web page.

hypertext transfer protocol (HTTP)

An internet standard that supports the exchange of information using the web.

Contrast with

protocol

identifier The name that is assigned to an item in a program such as a class, function, or variable.

Related terms

class, function, variable

image A graphic that appears in a report. BIRT Report Designer supports .gif, .jpg, and .png file types.

Related terms

Business Intelligence and Reporting Tools (BIRT) Report Designer, report

Contrast with

image element

image element



A report item that adds an image to a report design.

Related terms

design, element, image, report, report item

inheritance

A mechanism whereby one class of objects can be defined as a special case of a more general class and includes the method and variable definitions of the general class, known as a base or superclass. The superclass serves as the baseline for the appearance and behavior of the descendant class, which is also known as a subclass. In the subclass, the appearance and behavior can be further customized without affecting the superclass. Typically, a subclass augments or redefines the behavior and structure of its superclass or superclasses. Figure G-11 shows an example of inheritance.

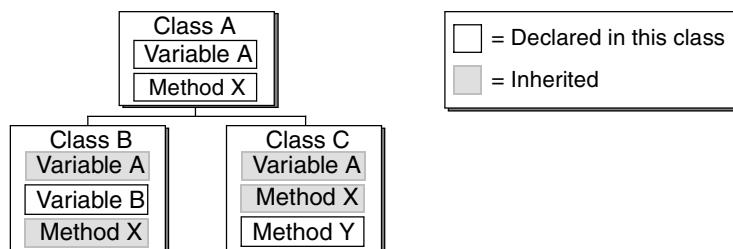


Figure G-11 Inheritance

Related terms

class, descendant class, file types, method, object, subclass, superclass, variable

Contrast with

abstract base class, hierarchy

inner join

- 1 A type of join that returns records from two tables that are based on their having specified values in the join fields. The most common type of inner join is one in which records are combined and returned when specified field values are equal. For example, if customer and order tables are joined on customer ID, the result set contains only combined customer and order records where the customer IDs are equal, excluding records for customers who have no orders.
- 2 When creating a joint data set in BIRT, a type of join that returns all rows from both data sets if the specified field values are equal. For example, if customer and order data sets are joined on customer ID, the joint data set returns only combined customer and order rows where the customer IDs are equal.

Related terms

Business Intelligence and Reporting Tools (BIRT) technology, data set, field, join, result set, row, table, value

Contrast with

outer join

input source

See data source.

instance See object.

instance variable

A variable that other instances of a class do not share. The run-time system creates a new copy of an instance variable each time the system instantiates the class. An instance variable can contain a different value in each instance of a class, for example, the customerID variable in a Customer class.

Related terms

class, value, variable

Contrast with

class variable, dynamic variable, field variable, global variable, local variable, member variable, static variable

instantiation

The action of creating an object.

Related term

object

Contrast with

class

interface

- 1 The connection and interaction among hardware, software, and the user. Hardware interfaces include plugs, sockets, wires, and electrical pulses traveling through them in a particular pattern. Hardware interfaces include electrical timing considerations such as Ethernet and Token Ring, network topologies, RS-232 transmission, and the IDE, ESDI, SCSI, ISA, EISA, and Micro Channel. Software or programming interfaces are the languages, codes, and messages that programs use to communicate with each other and to the hardware and the user. Software interfaces include applications running on specific operating systems, SMTP e-mail, and LU 6.2 communications protocols.
- 2 In Java, an interface defines a set of methods to provide a required functionality. The interface provides a mechanism for classes to communicate in order to execute particular actions.

Related terms

application, class, Connection, Java, method, protocol

internationalization

The process of designing an application to work correctly in multiple locales.

Related terms

application, locale

Contrast with

localization

IP address The unique 32-bit ID of a node on a TCP/IP network.

Related term	
node	
J2EE	See Java 2 Enterprise Edition (J2EE).
J2SE	See Java 2 Runtime Standard Edition (J2SE).
JAR	See Java archive (.jar) file.
Java	A programming language that is designed for writing client/server and networked applications, particularly for delivery on the web. Java can be used to write applets that animate a web page or create an interactive web site.
Related terms	
	applet, application, web page
Contrast with	
	JavaScript

Java 2 Enterprise Edition (J2EE)

A platform-independent environment that includes APIs, services, and transport protocols, and is used to develop and deploy web-based enterprise applications. Typically, this environment is used to develop highly scalable web-based applications. This environment builds on J2SE functionality and requires an accessible J2SE installation.

Related terms	
	application, application programming interface (API), enterprise, enterprise reporting, Java 2 Runtime Standard Edition (J2SE), protocol
Contrast with	
	Enterprise JavaBean (EJB), Java Development Kit (JDK)

Java 2 Runtime Standard Edition (J2SE)

A smaller-scale, platform-independent environment that provides supporting functionality to the capabilities of J2EE. The J2SE does not support Enterprise JavaBean or enterprise environment.

Related terms	
	enterprise, Enterprise JavaBean (EJB), Java 2 Enterprise Edition (J2EE)
Contrast with	
	Java Development Kit (JDK)

Java archive (.jar) file

A file format that is used to bundle Java applications.

Related terms	
	application, Java
Contrast with	
	web archive (.war) file

Java Database Connectivity (JDBC)

A standard protocol that Java uses to access database data sources in a platform-independent manner.

Related terms

data source, Java, protocol

Contrast with

data element, schema

Java Development Kit (JDK)

A Sun Microsystems software development kit that defines the Java API and is used to build Java programs. The kit contains software tools and other programs, examples, and documentation that enable software developers to create applications using the Java programming language.

Related terms

application, application programming interface (API), Java

Contrast with

Java 2 Enterprise Edition (J2EE), Java 2 Runtime Standard Edition (J2SE), JavaServer Page (JSP)

Java Naming and Directory Interface (JNDI)

A naming standard that provides clients with access to items such as EJBs.

Related term

Enterprise JavaBean (EJB)

Java Virtual Machine (JVM)

The Java SDK interpreter that converts Java bytecode into machine language for execution in a specified software and hardware configuration.

Related terms

Java, SDK (Software Development Kit)

JavaBean

A reusable, standards-based component that is written in Java that encapsulates the business logic of an application. A JavaBean can provide access to data or model the data itself.

Related terms

application, data, encapsulation, Java

Contrast with

Enterprise JavaBean (EJB), enterprise reporting

JavaScript

An interpreted, platform-independent language that is used to enhance web pages and provide additional functionality in web servers. For example, JavaScript can interact with the HTML of a web page to change an icon when the cursor moves across it.

Related terms

hypertext markup language (HTML), web page, web server

Contrast with

Java

JavaServer Page (JSP)

A standard Java extension that simplifies the creation and management of dynamic web pages. The code combines HTML and Java code in one document. The Java code uses tags that instruct the JSP container to generate a servlet.

Related terms

hypertext markup language (HTML), Java, servlet, tag, web page

JDBC

See Java Database Connectivity (JDBC).

JDK

See Java Development Kit (JDK).

JNDI

See Java Naming and Directory Interface (JNDI).

join

A SQL query operation that combines records from two tables and returns them in a result set that is based on the values in the join fields. Without additional qualification, join usually refers to one where field values are equal. For example, customer and order tables are joined on a common field such as customer ID. The result set contains combined customer and order records in which the customer IDs are equal.

Related terms

field, query, result set, SQL (Structured Query Language), table, value

Contrast with

inner join, join condition, outer join, SQL SELECT statement

join condition

A condition that specifies a match in the values of related fields in two tables. Typically, the values are equal. For example, if two tables have a field called customer ID, a join condition exists where the customer ID value in one table equals the customer ID value in the second table.

Related terms

field, join, table, value

joint data set

A data set that combines data from two data sets.

Related terms

data, data set

JSP

See JavaServer Page (JSP).

JVM

See Java Virtual Machine (JVM).

keyword

A reserved word that is recognized as part of a programming language.

label element



A report item that displays a short piece of static text in a report.

Related terms

report, report item

Contrast with	data element, text element
layout	The designed appearance of a report. Designing the layout of a report entails placing report items on a page and arranging them in a way that helps the report user analyze the information easily. A report displays information in a tabular list, a series of paragraphs, a chart, or a series of subreports.
Related terms	chart, listing report, report, report item, subreport
layout editor	A window in a report designer in which a report developer arranges, formats, and sizes report elements.
Related terms	design, report
Contrast with	previewer, Property Editor, report editor, script editor
lazy load	The capability in a run-time environment to load a code segment to memory only if it is needed. By lazily loading a code segment, the run-time environment minimizes start-up time and conserves memory resources. For example, BIRT Report Engine builds a registry at startup that contains the list of available plug-ins, then loads a plug-in only if the processing requires it.
Related terms	Business Intelligence and Reporting Tools (BIRT) Report Engine, plug-in
left outer join	See outer join.
library	A collection of reusable and shareable report elements. A library can contain embedded images, styles, visual report items, JavaScript code, data sources, and data sets. A report developer uses a report designer to develop a library and to retrieve report elements from a library for use in a report design.
Related terms	Business Intelligence and Reporting Tools (BIRT) Report Designer, data set, data source, design, JavaScript, report element, report item, style
Contrast with	file types
library (.rptlibrary) file	In BIRT Report Designer, an XML file that contains reusable and shareable report elements. A report developer uses a report designer to create a library file directly or from a report design (.rptdesign) file.

Related terms

Business Intelligence and Reporting Tools (BIRT) Report Designer, design, extensible markup language (XML), report design (.rptdesign) file, report element

Contrast with

file types, report design (.rptdesign) file, report document (.rptdocument) file, report template (.rpttemplate) file

link

See hyperlink.

listener

See event listener.

list element

A report item that iterates through the data rows in a data set. The list element contains and displays other report items in a variety of layouts.

Related terms

data, data row, data set, element, layout, report item

Contrast with

grid element, table element

listing report

A report that provides a simple view of data. Figure G-12 shows a listing report.

Customer List		
Customer	Phone	Contact
ANG Resellers	(91) 745 6555	Alejandra Camino
AV Stores, Co.	(171) 555-1555	Rachel Ashworth
Alpha Cognac	61.77.6555	Annette Roulet

Figure G-12 Listing report

Related terms

data, report

local variable

A variable that is available only at the current level in an application. A local variable stays in memory in the scope of an executing procedure until the procedure terminates. When the procedure ends, the run-time system destroys the variable and returns the memory to the system.

Related terms

application, procedure, scope, variable

Contrast with

field variable, global variable

locale

A location and the language, date format, currency, sorting sequence, time format, and other such characteristics that are associated with that location. The location is not always identical to the country. There can be multiple languages and locales within one country. For example, China

has two locales: Beijing and Hong Kong. Canada has two language-based locales: French and English.

Contrast with
localization

localization

The process of translating database content, printed documents, and software programs into another language. Report developers localize static text in a report so that the report displays text in another language that is appropriate to the locale configured on the user's machine.

Related terms
locale, report

Contrast with
internationalization

manifest A text file in a Java archive (.jar) file that describes the contents of the archive.

Related term
Java archive (.jar) file

master page

 A predefined layout that specifies a consistent appearance for all pages of a report. A master page typically includes standard headers and footers that display information such as page numbers, a date, or a copyright statement.

The master page can contain report elements in the header and footer areas only, as shown in Figure G-13.

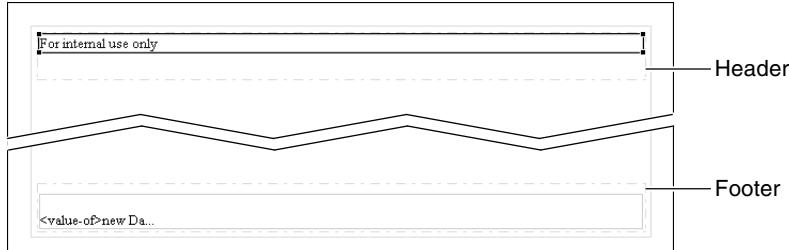


Figure G-13 Master page layout

The master page's header and footer content appears on every page of the report in paginated formats, as shown in Figure G-14.

Related terms

Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) Report Designer, footer, grid element, header, hypertext markup language (HTML), layout, previewer, report, template

measure In a cube, values that are aggregated, such as average cost or total units of products.

Related terms

Business Intelligence and Reporting Tools (BIRT) technology, cube, value

Contrast with

dimension

Customer List		
Report generated on Aug 23, 2005 12:14 PM		
Customer	Phone	Contact
ANG Resellers	(91) 745 6555	Alejandra Camino
AV Stores, Co.	(171) 555-1555	Rachel Ashworth
Alpha Cognac	61.77.6555	Annette Roulet
American Souvenirs Inc	2035557845	Keith Franco
Amica Models & Co.	011-4988555	Paolo Accoti
Baane Mini Imports	07-98 9555	Jonas Bergulsen
Bavarian Collectables Imports, Co.	+49 89 61 08 9555	Michael Donnermeyer
Blauer See Auto, Co.	+49 69 66 90 2555	Roland Keitel

Aug 23, 2005 12:14 PM

Header content

Footer content

Figure G-14 Master page content

member

A method or variable that is defined in a class and provides or uses information about the state of a single object.

Related terms

class, method, object, variable

Contrast with

global variable, instance variable, static variable

member variable

A declared variable within a class. A set of member variables in a class contains the data or state for every object of that class.

Related terms

class, data, declaration, object, variable

Contrast with

class variable, dynamic variable, field variable, global variable, instance variable, local variable, static variable

metadata

Information about the structure of data that enables a program to process information. For example, a relational database stores metadata that describes the name, size, and data type of objects in a database, such as tables and columns.

Related terms

column, data, data type, table

method

A routine that provides functionality to an object or a class.

Related terms

class, object

Contrast with

data, function

modal window

A window that retains focus until explicitly closed by the user. Typically, dialog boxes and message windows are modal. For example, an error message dialog box remains on the screen until the user responds.

Contrast with
modeless window

mode An operational state of a system. Mode implies that there are at least two possible states. Typically, there are many modes for both hardware and software.

modeless window

A window that solicits input but permits users to continue using the current application without closing the modeless window, for example, an Eclipse view.

Related terms
application, Eclipse view

Contrast with
modal window

multidimensional data

Any set of records that you can break down or filter according to the contents of individual fields or dimensions, such as product, location, or time. This data organization supports presenting and analyzing complex relationships.

Related terms
data, dimension, field, filter

multithreaded application

An application that handles multiple simultaneous users and sessions.

Related term
application

Navigator In BIRT Report Designer, an Eclipse view that shows all projects and reports within each project. Each project is a directory in the file system. Use Navigator to manage report files, for example, deleting files, renaming files, or moving files from one project to another. Figure G-15 shows Navigator.

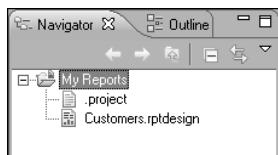


Figure G-15 Navigator

Related terms
Eclipse project, Eclipse view, report

node	A computer that is accessible on the internet.
	Contrast with domain name
null	A value that indicates that a variable or field contains no data.
	Related terms data, field, value, variable
numeric data type	A data type that is used for calculations that result in a value that is a number. Report items that contain expressions or fields with a numeric data type display numbers, based on the formats and locale settings that are specified by your computer and the report design.
	Related terms data type, design, expression, field, format, locale, report, report item, value
numeric expression	A numeric constant, a simple numeric variable, a scalar reference to a numeric array, a numeric-valued function reference, or a sequence of these items, that are separated by numeric operators. For example:
	<pre>dataSetRow["PRICEEACH"] * dataSetRow["QUANTITYORDERED"]</pre>
	Related terms array, constant, function, operator, variable
	Contrast with Boolean expression
object	An instance of a particular class, including its characteristics, such as instance variables and methods.
	Related terms class, instance variable, method, variable
object-oriented programming	A technique for writing applications using classes, not algorithms, as the fundamental building blocks. The design methodology uses three main concepts: encapsulation, inheritance, and polymorphism.
	Related terms application, class, encapsulation, inheritance, polymorphism
	Contrast with object
octal number	A number in base 8. An octal number uses only the digits 0 through 7. Each place represents a power of 8. By comparison, base 10 numbers use the digits 0 through 9. Each place represents a power of 10.
	Contrast with hexadecimal number

ODA See open data access (ODA).

online analytical processing (OLAP)

The process of analyzing, collecting, managing, and presenting multidimensional data.

Related terms

data, multidimensional data

online help

Information that appears on the computer screen to help the user understand an application.

Related term

application

open data access (ODA)

A technology that enables accessing data from standard and custom data sources. ODA uses XML data structures and Java interfaces to handle communication between the data source and the application that needs the data. Using ODA to access data from a data source requires an ODA driver and typically also includes an associated tool for designing queries on the data source. ODA provides interfaces for creating data drivers to establish connections, access metadata, and execute queries to retrieve data. ODA also provides interfaces to integrate query builder tools within an application designer tool. In BIRT, ODA is implemented using plug-ins to the Eclipse Data Tools Project.

Related terms

application, Business Intelligence and Reporting Tools (BIRT), Connection, data, data source, driver, extensible markup language (XML), interface, Java, metadata, open data access (ODA) driver, plug-in, query

open data access (ODA) driver

An ODA driver communicates between an arbitrary data source and an application during report execution. An ODA driver establishes a connection with a data source, accesses metadata about the data, and executes queries on the data source. Each ODA driver consists of a configuration file and classes that implement the ODA run-time Java interfaces that conform to ODA. In BIRT, ODA drivers are implemented as an Eclipse plug-in to the Data Tools Platform project.

Related terms

application, BIRT technology, class, Connection, data, data source, driver, Eclipse, interface, Java, metadata, open data access (ODA), plug-in, query

open database connectivity (ODBC)

A standard protocol that is used by software products as one of the database management system (DBMS) interfaces to connect applications and reports to databases that comply with this specification.

Related terms

application, database management system (DBMS), interface, protocol, report

Contrast with

Connection, data source, Java Database Connectivity (JDBC)

operator

A symbol or keyword that performs an operation on expressions.

Related terms

expression, keyword

outer join

A type of join that returns records from one table even when no matching values exist in the other table. The kinds of outer join are the left outer join, the right outer join, and the full outer join. The left outer join returns all records from the table on the left in the join operation, even when no matching values exist in the other table. The right outer join returns all records from the table on the right in the join operation. For example, if customers and orders tables are left outer joined on customer ID, the result set will contain all customer records, including records for customers who have no orders. The full outer join returns the results of both left and right outer joins. The result set contains all records from both tables.

A joint data set supports all outer join types between two data sets.

Related terms

Business Intelligence and Reporting Tools (BIRT) technology, data set, join, joint data set, query, result set, row, table, value

Contrast with

inner join

Outline

An Eclipse view that shows all report elements that comprise a report design, report library, or report template. Outline shows the report elements' containment hierarchy in a tree-structured diagram.

Figure G-16 shows Outline.

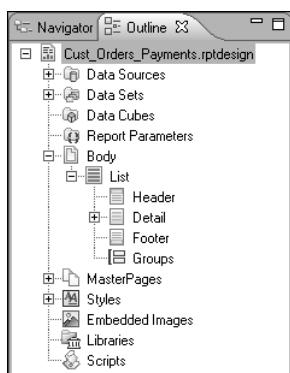


Figure G-16 Outline

Related terms

design, Eclipse view, hierarchy, library, report, report element, template

package

A set of functionally related Java classes that are organized in one directory.

Related terms

class, Java

Palette

An Eclipse view that shows the visual report elements for organizing and displaying data in a report. Figure G-17 shows Palette.

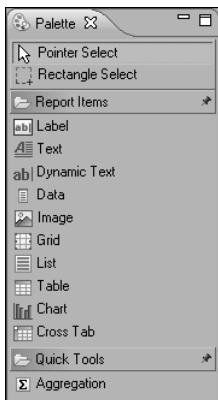


Figure G-17 Palette

Related terms

data, Eclipse view, report, report element

parameter

- 1 A report element that provides input to the execution of the report. Parameters provide control over report data selection, processing, and formatting.
- 2 The definition of an argument to a procedure.

Related terms

argument, data, format, procedure, report, report element

Contrast with

cascading parameters, data set parameter, report parameter

parent class

See superclass.

password

An optional code that restricts user name access to a resource on a computer system.

pattern

A template or model for implementing a solution to a common problem in object-oriented programming or design. For example, the singleton design pattern restricts the instantiation of a class to only one object. The use of the singleton pattern prevents the proliferation of identical objects

in a run-time environment and requires a programmer to manage access to the object in a multithreaded application.

Related terms

class, design, instantiation, multithreaded application, object, object-oriented programming

perspective

See Eclipse perspective.

platform

The software and hardware environment in which a program runs. Linux, MacOS, Microsoft Windows, Solaris OS, and UNIX are examples of software systems that run on hardware processors made by vendors such as AMD, Apple, Intel, IBM, Motorola, Sun, and Hewlett-Packard.

plug-in

- 1 An extension that is used by the Eclipse development environment. At run time, Eclipse scans its plug-in subdirectory to discover any extensions to the platform. Eclipse places the information about each extension in a registry, using lazy load to access the extension.
- 2 A software program that extends the capabilities of a web browser. For example, a plug-in gives you the ability to play audio samples or video movies.

Related terms

application, Eclipse, extension, lazy load

Contrast with

Eclipse Plug-in Development Environment (PDE)

plugin fragment

A separately loaded plug-in that adds functionality to an existing plug-in, such as support for a new language in a localized application. The plugin manifest contains attributes that associate the fragment with the existing plug-in.

Related terms

application, localization, manifest, plug-in

polymorphism

The ability to provide different implementations with a common interface, simplifying the communication among objects. For example, defining a unique print method for each kind of document in a system supports printing any document by sending the instruction to print without concern for how that method is actually carried out for a given document.

Related terms

interface, method, object

portal

A web page that serves as a starting point for accessing information and applications on the internet or an intranet. The basic function of a portal is to aggregate information from different sources.

Related terms
application, web page
Contrast with
portlet
portlet
A window in a browser that provides a view of specific information that is available from a portal.
Related term
portal
previewer
A design tool that supports displaying a report or data.
Related terms
data, design, report
Contrast with
layout editor, script editor, Standard Viewer
procedure
A set of commands, input data, and statements that perform a specific set of operations. For example, methods are procedures.
Related terms
data, method, statement
process
A computer program that has no user interface. For example, the servlet that generates a BIRT report is a process.
Related terms
Business Intelligence and Reporting Tools (BIRT), interface, report, servlet
project
See Eclipse project.
Properties
A grouped alphabetical list of all properties of report elements in a report design. Experienced report developers use this Eclipse view to modify any property of a report element. Properties displays the same content as the Advanced page of Property Editor. Figure G-18 shows Properties.

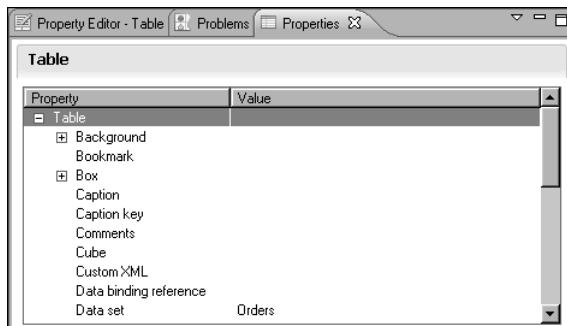


Figure G-18 Properties

Related terms

design, Eclipse view, property, Property Editor, report, report element

property A characteristic of a report item that controls its appearance and behavior. For example, a report developer can specify a font size for a label element.

Related terms

font, label element, report item, value

Contrast with

method

Property Editor

An Eclipse view that displays sets of key properties of report elements in a report design. The report developer uses Property Editor to modify the properties of report elements. Figure G-19 shows Property Editor.

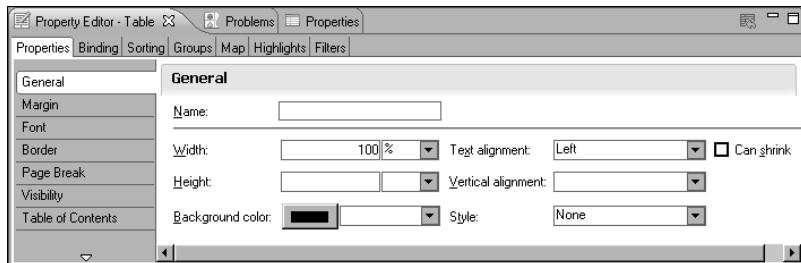


Figure G-19 Property Editor

Related terms

design, Eclipse view, property, report, report element

Contrast with

Properties

protocol A communication standard for the exchange of information. For example, in TCP/IP, the internet protocol (IP) is the syntax and order in which messages are received and sent.

Related term

syntax

publish To copy files to a shared folder to make them available to report users and developers. Libraries and resource files are published to the resources folder. Templates are published to the templates folder.

Related terms

library, report executable file, resource file, template

query A statement that specifies which data rows to retrieve from a data source. For example, a query that retrieves data from a database typically is a SQL SELECT statement.

Related terms

data row, data source, SQL SELECT statement

range A continuous set of values of any data type. For example, 1–31 is a numeric range.

Related terms
data type, value

regular expression

A JavaScript mechanism that matches patterns in text. The regular expression syntax can validate text data, find simple and complex strings of text within larger blocks of text, and substitute new text for old text.

Related terms
data, expression, JavaScript, syntax

rendering extension

A BIRT extension that produces a report in a specific format. For example, BIRT provides rendering extensions for HTML and PDF.

Related terms
Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) extension, extension, hypertext markup language (HTML), report

report A category of documents that presents formatted and structured content from a data source, such as a database or text file.

Related terms
data source, format, structured content

report design (.rptdesign) file

An XML file that contains the complete description of a report. The report design file describes the structure and organization of the report, its constituent report items and their style attributes, its data sets, its data sources, and its Java and JavaScript event handler code. BIRT Report Designer creates the report design file and the BIRT Report Engine processes it to create a formatted report.

Related terms
Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) Report Designer, Business Intelligence and Reporting Tools (BIRT) Report Engine, data set, data source, design, event handler, extensible markup language (XML), format, Java, JavaScript, report, report item, style

Contrast with
file types, library (.rptlibrary) file, report document (.rptdocument) file, report template (.rpttemplate) file

report document (.rptdocument) file

A binary file that encapsulates the report item identifier and additional information, such as data rows, pagination information, and table of contents information.

Related terms
Business Intelligence and Reporting Tools (BIRT) Report Engine, data row, report item

Contrast with

file types, library (.rptlibrary) file, report design (.rptdesign) file, report template (.rpttemplate) file

report editor

In BIRT Report Designer, the main window where a report developer designs and previews a report. The report editor supports opening multiple report designs. For each report design, the report editor displays these five pages: layout editor, master page editor, previewer, script editor, and XML source editor.

Related terms

Business Intelligence and Reporting Tools (BIRT) Report Designer, design, extensible markup language (XML), master page, layout editor, previewer, report, script editor

Contrast with

report design (.rptdesign) file

report element

A visual or non-visual component of a report design. A visual report element, such as a table or a label, is a report item. A non-visual report element, such as a report parameter or a data source is a logical component.

Related terms

data source, design, element, label element, report, report item, report parameter, table element

report executable file

A file that contains instructions for generating a report document.

Related terms

file types, report

report item

A report element that you add to a report design to display content in the report output. For example, a data element displays data from a data set when you run a report.

Related terms

data, data element, data set, design, report, report element, run

report item extension

A BIRT extension that implements a custom report item.

Related terms

Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) extension, extension, report, report item

report library file

See library (.rptlibrary) file.

Report Object Model (ROM)

The set of XML report elements that BIRT technology uses to build a report design file. ROM defines report elements for both the visual and non-visual components of a report. The complete ROM specification is at:

<http://www.eclipse.org/birt/ref>

Related terms

Business Intelligence and Reporting Tools (BIRT) technology, design, element, extensible markup language (XML), report, report element

Contrast with

Report Object Model (ROM) element

Report Object Model definition file (rom.def)

The file that BIRT technology uses to generate and validate a report design. rom.def contains property definitions for the ROM elements. rom.def does not include definitions for report items that are defined by report item extensions, such as the chart element.

Related terms

Business Intelligence and Reporting Tools (BIRT) technology, chart element, design, property, report, report item, report item extension

Contrast with

Report Object Model (ROM), Report Object Model (ROM) element, Report Object Model (ROM) schema

Report Object Model (ROM) element

An XML element in rom.def that defines a report element.

Related terms

element, extensible markup language (XML), report element, Report Object Model definition file (rom.def)

Contrast with

report item, Report Object Model (ROM) schema

Report Object Model (ROM) schema

The XML schema that defines the rules for the structure of report design files. All BIRT report design files must conform to this schema. To validate a report design, open the file in a schema-aware XML viewer such as XML Spy. The ROM schema is at:

<http://www.eclipse.org/birt/2005/design>

Related terms

Business Intelligence and Reporting Tools (BIRT), design, extensible markup language (XML), report, Report Object Model (ROM), schema

Contrast with

Report Object Model definition file (rom.def)

report parameter

1 See parameter.

- 2** A report element that contains a value. Report parameters provide an opportunity for the user to type a value as input to the execution of the report.

Related terms

parameter, report, report element, value

Contrast with

cascading parameters, data set parameter

report template

See template.

report template (.rpttemplate) file

An XML file that contains a reusable design that a report developer can employ when developing a new report.

Related terms

design, extensible markup language (XML), file types, report, style, template

Contrast with

file types, library (.rptlibrary) file, report design (.rptdesign) file, report document (.rptdocument) file

report viewer servlet

A J2EE web application servlet that produces a report from a report design (.rptdesign) file or a report document (.rptdocument) file. When deployed to a J2EE application server, the report viewer servlet makes reports available for viewing over the web. The report viewer servlet is also the active component of the report previewer of BIRT Report Designer.

Related terms

application, Business Intelligence and Reporting Tools (BIRT) Report Designer, Java 2 Enterprise Edition (J2EE), previewer, report, report design (.rptdesign) file, report document (.rptdocument) file, servlet, web server

request

A message that an application sends to a server to specify an operation for the server to perform.

Related term

application

Contrast with

response

reserved word

See keyword.

response

A message that a server sends to an application. The response message contains the results of a requested operation.

Related term

application

Contrast with

request

resource file

A text file that contains the mapping from resource keys to string values for a particular locale. Resource files support producing a report with localized values for label and text elements.

Related terms

label element, locale, localization, resource key, text element, value

resource key

A unique value that maps to a string in a resource file. For example, the resource key, greeting, can map to Hello, Bonjour, and Hola in the resource files for English, French, and Spanish, respectively.

Related terms

label element, locale, localization, resource file, text element, value

result set

Data rows from an external data source. For example, the data rows that are returned by a SQL SELECT statement performed on a relational database are a result set.

Related terms

data, data row, data source, SQL SELECT statement

Rich Client Platform (RCP)

See Eclipse Rich Client Platform (RCP).

right outer join

See outer join.

ROM

See Report Object Model (ROM).

row

1 A record in a table.

2 A horizontal sequence of cells in a grid element or table element.

Related terms

cell, grid element, table, table element

Contrast with

data row

RPTDESIGN

See report design (.rptdesign) file.

RPTDOCUMENT

See report document (.rptdocument) file.

RPTLIBRARY

See library (.rptlibrary) file.

RPTTEMPLATE

See report template (.rpttemplate) file.

run

To execute a program, utility, or other machine function.

schema

- 1 A database schema specifies the structure of database objects and the relationships between the data. The database objects are items such as tables.
- 2 An XML schema defines the structure of an XML document. An XML schema consists of element declarations and type definitions that describe a model for the information that a well-formed XML document must contain. The XML schema provides a common vocabulary and grammar for XML documents that support exchanging data among applications.

Related terms

application, data, element, extensible markup language (XML), object, report, table, well-formed XML

scope

The parts of a program in which a symbol or object exists or is visible. Where the element is declared determines the scope of a program element. Scopes can be nested. A method introduces a new scope for its parameters and local variables. A class introduces a scope for its member variables, member functions, and nested classes. Code in a method in one scope has visibility to other symbols in that same scope and, with certain exceptions, to symbols in outer scopes.

Related terms

class, function, member, method, object, parameter, variable

script editor

In the report editor in BIRT Report Designer, the page where a report developer adds or modifies JavaScript for a report element.

Related terms

Business Intelligence and Reporting Tools (BIRT) Report Designer, JavaScript, report, report editor, report element

Contrast with

layout editor, previewer

scripting language

See JavaScript.

SDK (Software Development Kit)

A collection of programming tools, utilities, compilers, debuggers, interpreters, and APIs that a developer uses to build an application to run on a specified technology platform. For example, the Java SDK supports

developers in building an application that users can download across a network to run on any operating system. The Java Virtual Machine (JVM), the Java SDK interpreter, executes the application in the specified software and hardware configuration.

Related terms

application, application programming interface (API), Java, Java Virtual Machine (JVM), platform

section

A horizontal band in a report design. A section structures and formats related report items. A section uses a grid element, list element, or table element to contain data values, text, and images.

Related terms

data, design, grid element, image, list element, report, report item, table element, value

select

- 1 To highlight one or more items, for example, in a report design. A user-driven operation then affects the selected items. Figure G-20 shows selected items.

Customer	Phone	Contact
row["CUSTOMERNAME..."]	row["PHONE"]	row["CONTACTFIRS..."]

Figure G-20 Selected items

- 2 To highlight a check box or a list item in a dialog box.

Related terms

design, report

SELECT

See SQL SELECT statement.

series

A sequence of related values. In a chart, for example, a series is a set of related points. Figure G-21 shows a bar chart that displays a series of quarterly sales revenue figures over four years.

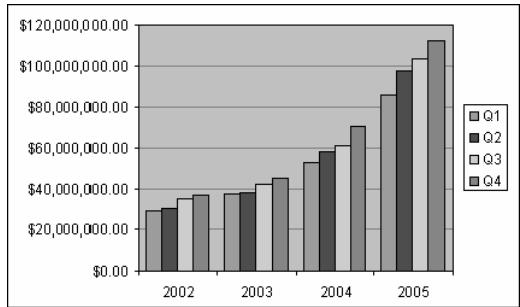


Figure G-21 Series in a chart

Related terms

chart, value

Contrast with
category

servlet A small Java application that runs on a web server to extend the server's functionality.

Related terms
application, Java, web server

simple object access protocol (SOAP)

A message-based protocol based on extensible markup language (XML). Use SOAP to access applications and their services on the web. SOAP employs XML syntax to send text commands across the internet using HTTP. SOAP supports implementing a messaging system.

Related terms

application, extensible markup language (XML), hypertext transfer protocol (HTTP), protocol, syntax

slot A construct that represents a set of ROM elements that are contained within another ROM element. For example, the body slot of the report design element can contain one or more of any type of report item. Figure G-22 shows a body slot.

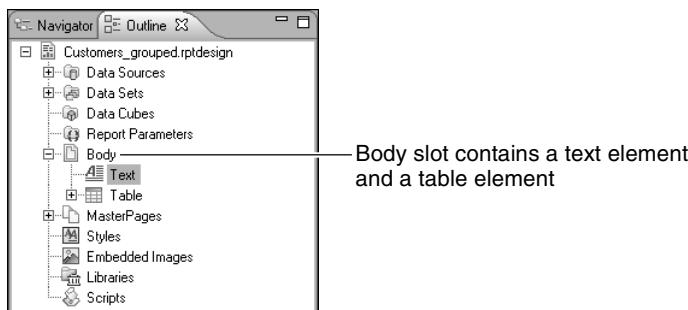


Figure G-22 Body slot

Related terms

design, element, report, report element, report item, Report Object Model (ROM) element

sort To specify the order in which data is processed or displayed. For example, customer names can be sorted in alphabetical order.

Related term

data

Contrast with

sort key

sort key A field that is used to sort data. For example, if you sort data by customer name, then the customer name field is a sort key. You can sort data using one or more sort keys.

Related terms
data, field, sort

SQL (Structured Query Language)

A language that is used to access and process data in a relational database. For example, the following SQL query accesses a database's customers table and retrieves the customer name and credit limit values where the credit limit is less than or equal to 100000. The SQL query then sorts the values by customer name:

```
SELECT customers.customerName,  
       customers.creditLimit  
    FROM customers  
   WHERE customers.creditLimit >= 100000  
 ORDER BY customers.customerName
```

Related terms
data, query, sort, table, value

Contrast with
SQL SELECT statement

SQL SELECT statement

A statement in SQL (Structured Query Language) that provides instructions about which data to retrieve for a report.

Related terms
data, report, SQL (Structured Query Language), statement

Standard Viewer

A viewer that appears after the user runs a report. In the Standard Viewer, the user can perform basic viewing tasks, such as navigating the report, viewing parameter information, exporting data, and using a table of contents.

Related terms
data, parameter, report

Contrast with
previewer, report viewer servlet

state See instance variable.

statement A syntactically complete unit in a programming language that expresses one action, declaration, or definition.

static variable

A variable that is shared by all instances of a class and its descendant classes. In Java, a static variable is known as a class variable. The compiler specifies the memory allocation for a static variable. The program receives the memory allocation for a static variable as the program loads.

Related terms
class, class variable, descendant class, Java, variable

Contrast with

dynamic variable, field variable, global variable, instance variable, local variable, member variable

String data type

A data type that consists of a sequence of contiguous characters including letters, numerals, spaces, and punctuation marks.

Related terms

character, data type

Contrast with

string expression

string expression

An expression that evaluates to a series of contiguous characters.

Elements of the expression can include a function that returns a string, a string constant, a string literal, a string operator, or a string variable.

Related terms

character, constant, expression, function, operator, variable

Contrast with

String data type

structured content

A formatted document that displays information from one or more data sources.

Related terms

data source, format

Contrast with

report

Structured Query Language (SQL)

See SQL (Structured Query Language).

style

A named set of formatting characteristics, such as font, color, alignment, and borders, that report developers apply to a report item to control its appearance.

Related terms

design, font, format, report, report item

Contrast with

cascading style sheet (CSS)

style sheet

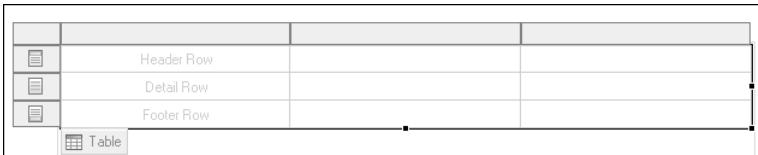
See cascading style sheet (CSS).

subclass

The immediate descendant class.

Related terms

class, descendant class

	Contrast with superclass
subreport	A report that appears inside another report. Typically, the subreport uses data values from the outer report.
	Related terms data, report, value
superclass	The immediate ancestor class.
	Related terms ancestor class, class
	Contrast with descendant class, subclass
syntax	The rules that govern the structure of a language.
tab	The label above a page in a dialog box that contains multiple pages.
	Contrast with label element
table	A named set of columns in a relational database.
	Related term column
	Contrast with table element
table element	<p> A report item that contains and displays data in a row and column format. The table element iterates through the data rows in a data set. Figure G-23 shows a table element.</p> 
	Figure G-23 Table element
	Related terms column, data, data row, data set, element, report item, row
	Contrast with grid element, list element, table
tag	An element in a markup language that identifies how to process a part of a document.
	Related term element

Contrast with
extensible markup language (XML)

template In BIRT Report Designer, a predefined structure for a report design. A report developer uses a report template to maintain a consistent style across a set of report designs and for streamlining the report design process. A report template can describe a complete report or a component of a report. BIRT Report Designer also supports custom templates.

In Figure G-24, New Report displays the available templates and Preview displays a representation of the report layout for the selected My First Report, a customer-listing-report template.

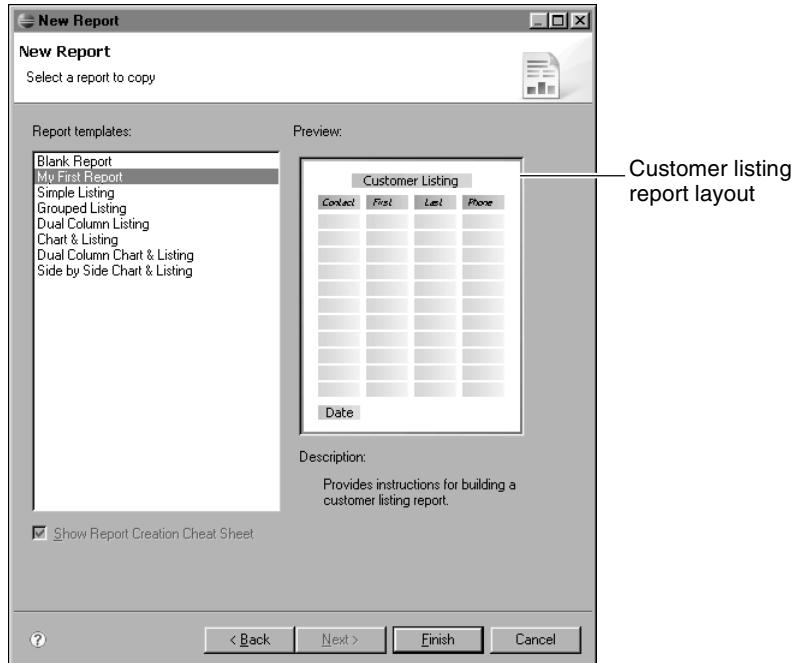


Figure G-24 Template

Related terms

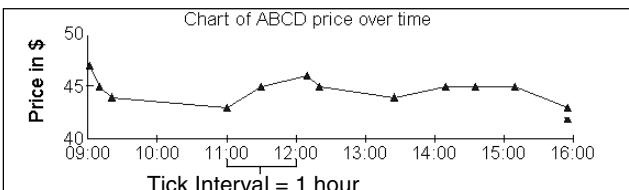
Business Intelligence and Reporting Tools (BIRT), Business Intelligence and Reporting Tools (BIRT) Report Designer, design, layout, listing report, report, report design (.rptdesign) file

Contrast with

report template (.rpttemplate) file

text element

-  A report item that displays user-specified text. The text can span multiple lines and can contain HTML formatting and dynamic values that are derived from data set fields or expressions.

Related terms	data set, expression, field, format, hypertext markup language (HTML), report item, value
Contrast with	data element, dynamic text element, label element
text file	See flat file.
theme	A set of related styles that are stored in a library (.rptlibrary) file. A theme provides a preferred appearance for the report items in a report design. A library file can store multiple themes. A report design can use styles from a single theme as well as styles defined in the report design itself.
Related terms	design, library (.rptlibrary) file, report, report item, style
Contrast with	cascading style sheet (CSS)
tick	A marker that occurs at regular intervals along the x- or y-axis of a chart. Typically, the value of each tick appears on the axis.
Related term	chart
Contrast with	tick interval
tick interval	The distance between ticks on an axis. Figure G-25 shows a tick interval in a chart.
Related terms	chart, tick
 <p>Figure G-25 illustrates a line chart titled "Chart of ABCD price over time". The Y-axis is labeled "Price in \$" and ranges from 40 to 50. The X-axis shows time in hours from 09:00 to 16:00. The chart displays 7 data points connected by lines, representing price fluctuations over the period. A label "Tick Interval = 1 hour" is present at the bottom of the chart area.</p>	
Figure G-25	Tick interval
toolbar	A bar that contains various buttons that provide access to common tasks. Different toolbars are available for different kinds of tasks.
translator	See converter.
type	See data type.
Unicode	A living language standard that is managed by the Technical Committee of the Unicode Consortium. The current Unicode standard provides code points for more than 65,000 characters. Unicode encoding has no

dependency on a platform or software program and therefore provides a basis for software internationalization.

Related terms

code point, character, internationalization

Uniform Resource Locator (URL)

A character string that identifies the location and type of a piece of information that is accessible over the web. `http://` is the familiar indicator that an item is accessible over the web. The URL typically includes the domain name, type of organization, and a precise location within the directory structure where the item is located.

Related terms

character, domain name, hypertext transfer protocol (HTTP)

Contrast with

Universal Resource Identifier (URI)

universal hyperlink

See hyperlink.

Universal Resource Identifier (URI)

A set of names and addresses in the form of short strings that identify resources on the web. Resources are documents, images, downloadable files, and so on.

Contrast with

Uniform Resource Locator (URL)

URL

See Uniform Resource Locator (URL).

value

- 1 A quantity that is assigned to a constant, variable, parameter, or symbol.
- 2 A specific occurrence of an attribute. For example, blue is a possible value for the attribute color.

Related terms

constant, parameter, variable

variable

A named storage location for data that can be modified while a program runs. Each variable has a unique name that identifies it within its scope. Each variable is capable of containing a certain type of data.

Related terms

data, data type, scope

Contrast with

class variable, dynamic variable, field variable, global variable, instance variable, local variable, member variable, static variable

view

A predefined query that retrieves data from one or more tables in a relational database. Unlike a table, a view does not store data. Users can use views to select, delete, insert, and update data. The database uses the

definition of the view to determine the appropriate action on the underlying tables. For example, a database handles a query on a view by combining the requested data from the underlying tables.

Related terms

data, query, table

Contrast with

Eclipse view

viewer See previewer and Standard Viewer.

web archive (.war) file

A file format that is used to bundle web applications.

Related terms

application, format

Contrast with

Java archive (.jar) file

web page A page that contains tags that a web browser interprets and displays.

Related term

tag

web server

A computer or a program that provides web services on the internet. A web server accepts requests that are based on the hypertext transfer protocol (HTTP). A web server also executes server-side scripts, such as ASPs and JSPs.

Related terms

hypertext transfer protocol (HTTP), JavaServer Page (JSP), request, web page

web services

A software system designed to support interoperable machine-to-machine interaction over a network. Web services refers to clients and servers that communicate using XML messages that adhere to the SOAP standard. Web services are invoked remotely using SOAP or HTTP-GET and HTTP-POST protocols. Web services are based on XML and return a response to the client in XML format. Web services are language-independent. They can be built and consumed on any operating system as long as that operating system supports the SOAP protocol and XML.

Related terms

extensible markup language (XML), hypertext markup language (HTML), protocol, simple object access protocol (SOAP)

well-formed XML

An XML document that follows syntax rules that were established in the XML 1.0 recommendation. Well-formed means that a document must contain one or more elements and that the root element must contain all

the other elements. Each element must nest inside any enclosing elements, following the syntax rules.

Related terms

element, extensible markup language (XML), syntax

workbench

See Eclipse Workbench.

workspace

See Eclipse workspace.

World Wide Web Consortium (W3C)

An international, but unofficial, standards body that provides recommendations regarding web standards. The World Wide Web Consortium publishes several levels of documents, including notes, working drafts, proposed recommendations, and recommendations about web applications that are related to topics such as HTML and XML.

Related terms

application, extensible markup language (XML), hypertext markup language (HTML)

XML (extensible markup language)

See extensible markup language (XML).

XML element

See element.

XML PATH language (XPath)

XPath supports addressing an element or elements within an XML document based on a path through the document hierarchy.

Related terms

element, extensible markup language (XML)

XML schema

See schema.

XPath

See XML PATH language (XPath).

This page intentionally left blank

Symbols

" (double quotation mark) character
 command line arguments and 22
 JavaScript code and 329
,

(comma) character 448
\ (backslash) character 329
... button 602

A

absolute paths 31, 32
abstract base class 583
AbstractBaseDialog.js 58
AbstractBaseReportDocument.js 60
AbstractBaseToc.js 57
AbstractBaseToolBar.js 58
AbstractExceptionDialog.js 58
AbstractParameterDialog.js 59
AbstractReportComponent.js 60
AbstractUIComponent.js 58
accessing
 charts 324
 component libraries 92
 custom data sources 217, 478
 data sets 315
 data structures 310
 demo database 20
 design model objects 244
 Eclipse PDE 377
 external data sources 103, 478
 formatted output 301
 Java classes 151, 152, 159, 226, 295
 Java objects 217, 226
 metadata 240
 ODA data sources 478, 486

report components 246, 247
report designs 246, 275, 302, 304
report elements 275, 302
report items 275, 303, 306, 308
report properties 246
report viewer 28, 30
reports 30, 33, 301
resource files 281, 304
script editor 127
 source code 369
Acrobat Reader 301
action handlers 299
actions 299
 See also events
adapter classes 156, 161
Add Entry dialog 390
Add External JARS button 452
Add Folder button 391
Add Library button 390
adding
 charts to designs 339, 351, 352, 355
 custom drivers 90
 data sources 500
 lists 103
 master pages 102, 131, 298
 ODA drivers 478
 ODA user interfaces 478, 479, 480
 page breaks 120
 report items 305, 315, 399, 421
 scripted data sets 217, 220
 scripted data sources 219
 tables 103
 update site projects 396
Adobe Acrobat Reader. *See* Acrobat Reader

afterClose events 113, 114, 124, 164
afterClose method
 scripted data sets and 114, 164
 scripted data sources and 112, 113, 164
afterDataSetFilled events 188, 211
afterDataSetFilled method 183
afterDrawAxisLabel method 183
afterDrawAxisTitle method 183
afterDrawBlock events 194
afterDrawBlock method 183
afterDrawDataPoint events 210
afterDrawDataPoint method 183
afterDrawDataPointLabel method 184
afterDrawFittingCurve method 184
afterDrawLegendEntry method 184
afterDrawLegendItem events 196
afterDrawMarkerLine method 184
afterDrawMarkerRange method 184
afterDrawSeries method 184
afterDrawSeriesTitle method 184
afterFactory events 111, 124, 166
afterFactory method 111, 142, 144, 166
afterGeneration events 191
afterGeneration method 184
afterOpen events 113, 114, 163, 164
afterOpen method
 scripted data sets and 114, 164
 scripted data sources and 112, 113, 163
afterRender events 112, 124, 166
afterRender method 112, 166
afterRendering method 184
_agentstyle parameter 47
aggregate data. *See* aggregate values
aggregate functions 583
aggregate package 258, 259
aggregate rows 583
aggregate values 583
aggregating data 90, 112
AJAX-based communications 55
aliases 584
All Extensions section (Extensions) 386
alternate names. *See* aliases; display
 names
ancestor classes 584
 See also superclasses
anchor property 327
annotation element 370
annotations 370, 387
Ant scripts 392

Apache Derby databases 20
Apache Tomcat manager accounts 32
Apache Tomcat servers 27, 28, 32, 35
api extension package 419
API Javadoc 236
api packages 235, 275
APIs. *See* application programming
 interfaces
appContext parameter 229
AppContextKey attribute 78
AppContextValue attribute 78
appinfo attribute 370
applets 584
application context 78, 134, 228, 280, 296
Application Context Map 134, 228
application context objects 30, 169
application programming interfaces
 (APIs)
 BIRT engines and 88
 BIRT extensions and 369
 charts and 321, 345
 custom data sources and 478
 custom report designers and 93
 custom report generators and 93
 defined 584
 report designs and 279
 report engine and 237, 238
 report items extensions and 401
 report rendering extensions and 433
 reporting applications and 235, 275,
 276, 277
application servers 7, 21, 27, 28, 33
 See also specific type
applications
 See also multithreaded applications
accessing report designs for 246, 275,
 302, 304
accessing report items for 303
accessing report viewer for 30
configuring BIRT home for 279
connecting to external sources
 and 478, 491
creating stand-alone 93, 279
customizing 93
debugging 285
defined 584
deploying 31, 277
developing 235, 275, 277, 369

applications (*continued*)
generating charts and 17, 321, 322,
323, 345
generating reports from 275, 276, 279,
300, 302
getting context for 130, 169, 296
installing plug-ins for 277, 372
integrating with Eclipse 83
rendering environments for 275
running reports and 237, 238
validating report designs for 89
application-wide scriptable objects 239
archive files
See also jar files; war files
BIRT packages in 17
chart engine and 18
downloading 9, 11
unpacking 5, 6, 11
area charts 321
See also charts
arguments 162, 584
See also functions; parameters
array properties 139
arrays
connection properties and 140
defined 585
executable expressions and 96
ODA result sets and 497
row objects and 139
scripted data sets and 227, 228
user-defined properties and 96
ASCII files. *See* text files
assignment statements 585
attribute package 258, 264, 265
attributes
complex properties and 309
plug-in extension points and 388, 399
report item extensions and 370, 386
report parameters and 290
ROM elements and 98, 99
testing chart 265
web viewer tags 64
XML schemas and 370, 385
autobinding charting example 352
AutoDataBindingViewer class 352
axes properties (charts) 199, 328, 332
axes values 198, 327, 332, 334, 335
See also charts
axis objects 198

axis titles 200
axis.jar 28
axis-ant.jar 28
AxisImpl class 198
B
backslash (\) character 329
bar charts 188, 321
See also charts
base URLs 299
baseURL attribute
parameterPage tag 72
report tag 68
viewer tag 65
BEA WebLogic servers 27
beforeClose events 113, 114, 124, 163, 164
beforeClose method
data sets and 164
data sources and 112, 113, 114, 163
beforeDataSet events 188
beforeDataSetFilled events 187
beforeDataSetFilled method 184
beforeDrawAxisLabel events 198, 209
beforeDrawAxisLabel method 184
beforeDrawAxisTitle events 198, 210
beforeDrawAxisTitle method 184
beforeDrawBlock events 194, 205
beforeDrawBlock method 184
beforeDrawDataPoint events 210
beforeDrawDataPoint method 184
beforeDrawDataPointLabel method 184
beforeDrawFittingCurve method 184
beforeDrawLegendEntry method 184
beforeDrawLegendItem events 196
beforeDrawMarkerLine method 184
beforeDrawMarkerRange method 185
beforeDrawSeries events 206, 211
beforeDrawSeries method 185
beforeDrawSeriesTitle method 185
beforeFactory events 111, 116, 166, 174
beforeFactory method 111, 116, 166
beforeGeneration events 190, 203
beforeGeneration method 185
beforeOpen events 113, 114, 163, 164
beforeOpen method
connection properties and 141
data sets and 114, 164
data sources and 112, 113, 163
queries and 140

beforeRender events 111, 116, 166
beforeRender method 111, 116, 166
beforeRendering method 185
Binary Build section 390, 392
binary files 92, 275
binding data sets to charts 186, 339
binding data sets to report items 317, 318
BIRT 83, 586
 BIRT applications 84, 87
 See also applications
 BIRT components 83
 BIRT Data Tools Platform
 See also DTP Integration package
 BIRT Demo Database 20, 586
 See also Classic Models sample database
 BIRT Demo Database package 19
 BIRT documentation xxii, 236
 BIRT DTP Integration archive 19
 BIRT engines 84, 88
 See also specific engine
 BIRT extensions 93, 586
 See also extensions
 BIRT home 238, 239
 BIRT home directory 277, 279
 BIRT model API 303
 BIRT open source projects. *See* projects
 BIRT RCP Report Designer
 See also BIRT Report Designer; Rich Client Platforms
 accessing sample database for 20
 adding charting functionality for 18
 compared to BIRT Report Designer 88
 defined 587
 installing 10–11
 removing cached pages for 12
 requirements for 3
 scripting and 107
 specifying JVM for 12
 starting 11
 testing installations for 11
 updating 14, 15, 16
 BIRT Report Designer
 See also designs
 accessing packages for 17
 accessing sample database for 20
 adding charting functionality for 18
 adding data sources to 500
 compared to BIRT RCP Designer 88
configuring 5
defined 587
extending functionality of 93, 369
installing 9–10, 362
integrating report engine with 237
integrating with ODA drivers 90, 478
overview 87
previewing reports and 109
requirements for 3, 5
scripting and 107
specifying JVM for 12
starting 10, 19, 25
testing installations for 10, 19, 24, 25
tracking method execution with 144
updating 14, 15
writing Java code and 155
BIRT Report Designer Full Eclipse Install software 9–10
BIRT Report Designer perspective 19
BIRT Report Designer tools 85
BIRT report object model 95
 See also ROM
BIRT reports. *See* reports
BIRT Samples archive 23
BIRT Samples package 23, 588
BIRT Source Code package 23, 24
 See also source code
BIRT technology 588
BIRT web site xix
BIRT.exe 11
birt.war 29
BIRT_FONT_PATH variable 33
BIRT_HOME variable 21, 239, 280, 323
BIRT_OVERWRITE_DOCUMENT parameter 53
BIRT_RESOURCE_PATH parameter 53
BIRT_VIEWER_CONFIG_FILE parameter 53
BIRT_VIEWER_DOCUMENT_FOLDER parameter 53
BIRT_VIEWER_IMAGE_DIR parameter 53
BIRT_VIEWER_LOCALE parameter 53
BIRT_VIEWER_LOG_DIR parameter 53
BIRT_VIEWER_LOG_LEVEL parameter 53
BIRT_VIEWER_MAX_ROWS parameter 54

BIRT_VIEWER_PRINT_SERVERSIDE parameter 54
BIRT_VIEWER_SCRIPTLIB_DIR parameter 54
BIRT_VIEWER_WORKING_FOLDER parameter 30, 31, 54
BirtBaseResponseHandler.js 57
BirtCommunicationManager.js 56
BirtConfirmationDialog.js 59
BirtDndManager.js 57
BirtEngineServlet application 45, 50
BirtEvent.js 57
BirtEventDispatcher class 55
BirtEventDispatcher.js 57
BirtExceptiondialog.js 59
BirtExportReportDialog.js 59
BirtGetUpdatedObjectsResponseHandler.js 57
BirtNavigationBar.js 58
BirtParameterDialog.js 59
BirtPosition.js 60
BirtPrintReportDialog.js 59
BirtPrintReportServerDialog.js 59
BirtProgressBar.js 58
BirtReportDocument class 56
BirtReportDocument.js 60
BirtSimpleExportDataDialog.js 59
BirtSoapRequest.js 57
BirtSoapResponse.js 57
BirtTabbedDialogBase.js 59
BirtToc.js 58
BirtToolbar.js 58
BirtUtility.js 60
block class 194
Block interface 324, 326, 327
_bookmark parameter 47
bookmark attribute
 parameterPage tag 72
 report tag 68
 viewer tag 65
bookmarks 287, 585
Boolean class 152
BOOLEAN data type 139
Boolean expressions 585
BoundsImpl objects 197
break reports. *See* grouped reports
breakpoints 585
bridge class 586
browsers. *See* web browsers

BrowserUtility.js 60
buffered emitters 442
bug reports 7
Bugzilla xix
Build All command 392
Build Automatically command 392
Build Configuration page 390
Build page (PDE Manifest Editor) 381, 390
build settings 381
build.properties file 370, 381
build.properties page (PDE Manifest Editor) 381
building
 BIRT open source projects 361–362
 design tools 244
 Hibernate driver plug-in 547
 Hibernate ODA UI plug-in 564
 ODA extensions 481
 plug-in extensions 390–392
 plug-in fragments 572, 577, 579
 report design tools 245
 rotated label report item extension 401
 update sites 396

builds 7
buildUI method 426
BundleActivator interface 375
Business Intelligence and Reporting Tools 83, 586
 See also BIRT

C

cache 118, 373
cache conflicts 12
calculated columns. *See* computed fields
calculated values. *See* computed values
cancelling running tasks 301
canInherit attribute 488
canLeave method 562
capitalization. *See* case sensitivity
captions (charts) 202, 258
carriage return characters 449
cascading parameter groups 289, 292
cascading parameters 289, 292, 293, 588
 See also report parameters
cascading style sheets 97, 278, 589
 See also styles
CASCADING_PARAMETER_GROUP value 290

case sensitivity 589
category 589
category axes. *See* axes values
category element 506
category series 332, 336
See also data series
category series items 332
category values 589
See also charts
Cell objects 167, 168
CellHandle class 248
cells
 adding grid elements and 102, 315
 adding list elements and 122
 building programmatically 167, 168, 442
 defined 589
 determining contents of 307
 placing label elements in 315
 writing to CSV files and 446, 448
changeLogLevel method 285, 286
changing
 charts 195, 201, 324, 325
 connection properties 140
 data set bindings 317, 318
 data sources 317
 locales 14
 plug-in project settings 380, 381
 queries 140
 report designs 132, 279, 302, 305
 report elements 131, 304
 report item properties 309, 310, 311
 sort order 312
 source code 367
 table of contents 40
 URL context roots 30
character sets 590
character strings. *See* strings
characters
 CSV output files and 448, 449
 defined 590
 JavaScript code and 138, 329
 trimming 493
chart APIs 90
chart areas 205, 326
chart attributes 265
chart blocks 324
See also chart areas
chart builder 91, 355
chart building phase (events) 189
chart classes 257, 331
chart components. *See* chart items
chart device package 259, 260
chart elements 324, 590
See also charts
chart engine
 configuring 322
 defined 586
 handling events for 183, 186, 193
 installing 17–18
 overview 90
chart engine API 235, 257
chart engine API library 278
Chart Engine archive 18, 321
chart engine classes 258
chart engine documentation 18
chart event handler methods 200
chart event package 259, 260
chart events 121, 183, 185, 193, 208
chart examples plug-in 351
chart exception package 262
chart factory package 262
chart generator 235
chart instance objects 201, 203
See also chart objects
Chart interface 257, 325
chart item extensions 89
chart items
 See also charts
 adding to designs 336
 creating 338
 defined 91
 displaying 89
 getting 212
 setting dimensions of 339
 setting properties for 338
chart model 189
chart model implementation classes 257
chart model packages 257, 259, 263
chart objects
 See also charts
 accessing 200
 getting 324, 325
 instantiating 257, 331, 338
 modifying 257
chart package 17, 235, 257
chart properties
 changing 325

chart properties (*continued*)
chart instance objects and 331–332
chart items and 338
charting applications and 325–328
getting 201
setting 202, 258
chart regions 205
See also chart areas
chart reportitem plug-in 401
chart script context objects 200
chart scripting 329
chart subtypes 203
chart types 202, 203, 258, 324, 331
chart wizard. *See* chart builder
Chart3DViewer application 354
chartengineapi.jar 278
charting APIs 203, 211, 321, 324, 330, 345
charting application sample plug-ins 588
charting applications 321, 322, 323, 340
charting examples 321, 351, 401
charting extensions 401, 591
charting library. *See* chart engine
ChartModels class 353
charts
See also chart elements; chart items;
chart objects
accessing 324
adding interactive features to 352
adding series to 328, 332–336
adding to designs 339, 351, 352, 355
applying styles to 353
binding data series to 186
binding to data sets 186, 330, 339
changing 195, 201, 324, 325
creating 91, 257, 330, 338, 345
customizing 321
defined 590
defining event handlers for 329, 353
defining sample data for 203, 330, 337
deploying 322
displaying 191
exporting to CSV files and 440
generating 87, 90, 353
getting data sets for 339
getting primary base axis for 332
getting type 202
initializing data structures for 186
outlining areas in 332
rendering as images 297
scripting for 200, 214, 273
setting properties for. *See* chart
properties
specifying type 203, 330, 331
ChartScriptContext objects 200
chart-viewer.jar 367, 368
ChartWithAxes interface 257, 325
ChartWithAxes type 328
ChartWithAxesImpl class 190, 324, 325
ChartWithoutAxes interface 257, 325
ChartWithoutAxesImpl class 190, 324
ChartWizardLauncher charting
example 355
ChartWizardLauncher class 355
cheat sheets 93
choice definitions 99
ChoiceType element 99
class attributes 388
class definitions 99
Class element 99
class element 409
class files 152, 159, 388
class hierarchy 591
class loaders 488
class method definitions 100
class names 151, 161, 371, 591
class property 409, 438
class variables 591
See also instance variables; variables
classes
accessing Java 151, 152, 159, 226, 295
accessing report parameters and 287
associating with report elements 160
building report designs and 244, 245,
248
changing chart objects and 257
compiling rotated label plug-in
and 402, 408, 409
creating 99
customizing ODA drivers and 477,
480, 534
defined 591
deploying applications and 277
deploying Java 227
developing with 235
generating CSV output and 441, 442,
453
generating reports and 238, 241, 275,
279

classes (*continued*)
handling events and 155, 156
hierarchical diagrams for 236
instantiating 154
loading 154, 544
naming conventions for 161
referencing 151, 226
registering 329
running plug-in instances and 375
scripting and 151, 152
setting attributes for 388
setting properties for 389
setting user-supplied values and 287
classes directory 152
Classic Models sample database
See also demo database
accessing 20
testing installation for 20
writing event handlers for 144
Classic Search page (PDE Manifest Editor) 385
ClassicModels directory 20
ClassLoader objects 544
CLASSPATH property 296
CLASSPATH variable 280
classpaths
CSV ODA driver extensions and 489
CSV rendering extension and 437
custom Java packages and 151
Hibernate ODA drivers and 525, 544
Java event handler classes and 156,
 159
plug-in extensions and 380
report engine and 295
scripted data sources and 227
_clean parameter 47
-clean option 12
clean-up code 142, 144, 227
clearSections method 203
CLI library 278
close events 113, 114
close method
 Hibernate drivers and 538, 542
 output files and 144
 report designs and 319
 scripted data sets and 114, 166, 227
 scripted data sources and 112, 113,
 165, 218
 task objects and 295
closing
connections 491, 493, 542
cursors 492
data sets 113, 164, 166, 227
data sources 112, 163, 165, 218, 221
output files 446, 468
code
accessing data sources and 217
accessing Java source 369
accessing sample 398
adding event handlers and 108, 124,
 127
changing 367
checking for errors in 146
compiling 362, 392
controlling report creation and 107
customizing web viewer and 78
defining executable 103
deploying applications and 277
developing applications and 235, 276,
 277
developing Hibernate drivers and 526
developing ODA extensions and 479,
 480
downloading source 364, 402
generating CSV files and 433, 434, 457
installing 23–24
loading 373, 616
running reports and 129, 135
setting run-time connections and 141
tracking method execution in 142, 144
Code page. *See* script editor
code points 591
codec library 278
codec.jar 278
collectCustomProperties method
 CSVFilePropertyPage 512
 CSVFileSelectionPageHelper 513
 CSVFileSelectionWizardPage 514
 HibernateDataSourceWizard 558
 HibernatePageHelper 555, 557
 HibernatePropertyPage 558
collectDataSetDesign method 564
collections 97, 289, 290, 291
color settings 96, 195, 335
column binding 119, 120, 318, 592
column binding definitions. *See* column
 binding expressions
column binding expressions 119

column headings 171
column names
 See also column headings
 accessing CSV data and 480
 getting 171, 499
 retrieving values and 138
column types 171
columnar layouts 102
columnBindingsIterator method 318
columnNumbers variable 447
columns
 See also fields; computed fields
 accessing 138, 316
 adding to tables 222
 counting 202
 defined 592
 defining output 221
 getting information about 139, 171
 getting number of 139, 140, 171
 iterating through 140
 retrieving values in 138, 492
comma (,) character 448
comma tag 449
command line applications 237
command line arguments 22
command line library 278
comma-separated values. *See* CSV files
comma-separated values rendering extension. *See* CSV report rendering extension
commit method 537
commit operations 491, 537
common.jar 278
CommonConstant class 493, 500, 535
commons-cli.jar 278
communications protocol 627
compiler preferences 362
compiling 159, 362, 391, 392
complex properties 307, 310
compliance settings 363
component hierarchy (BIRT) 83
component libraries 88, 92
component package 258, 268
component palettes. *See* Palette view
components
 See also report elements; report items
 accessing 246, 247
 adding report items and 90, 102
 saving 88
 setting properties for 103
compute method 192
computed columns. *See* computed fields
computed data. *See* computed values
computed fields 139, 171, 223, 592
computed values 223, 593
ComputedColumnHandle type 318
concatenation 329
Concurrent Versions System repository.
 See CVS repository
conditional expressions. *See* Boolean expressions
configuration files 30, 31, 556, 593
Configuration Markup section 386
configuration objects 238, 280
configuring
 BIRT home 279
 Eclipse workspace 362
 extension points 386
 Hibernate drivers 535, 544, 547, 555
 caution for 525
 report engine 238, 239, 245, 280, 281
 report viewer for alternate locations 30
 web viewer 52–55
connect method 229
Connection class 229, 230, 493, 494, 534, 536
connection information. *See* connection properties
Connection objects 491, 535, 593
 See also connections
connection pooling 34–37
connection profile types 507
connection profiles 478, 506
connection properties 140
connection wizards 509
connectionClass property 229
connectionFactory element 507
connectionProfile element 507
connectionProfile extension 506, 509, 548
connectionProperties parameter 229
connections
 external data sources and 103
 getting 499, 535
 Hibernate data sources and 534, 536, 542
 JDBC data sources and 34, 140
 JNDI applications and 34, 36

connections (*continued*)
 ODA data sources and 90, 140, 491,
 493
 ODA drivers and 491, 493, 494, 535
 report engine and 276
 web services data sources and 228, 230
constants 151, 593
Constants class 510, 511
Constants.js 60
constructor code 593
constructSessionFactory method 536,
 544, 545
container elements 102
containers 593
containment 594
containment hierarchy 594
content. *See* structured content
content package 442, 443
ContentEmitterVisitor objects 444, 466
ContentType property 442
context mapping 30
context objects 130, 169
context parameters 31
context root 30
context-param element 31
contributors 361
converters 594
core API library 278
core plug-ins 369
coreapi.jar 278
counters 145, 146
Create Ant Build File command 392
Create Java Project dialog 452
create method 257
create_classicmodels.sql 20
createAndInitCustomControl
 method 511, 555, 557, 558
createCustomControl method 512, 555
createFactoryObject method 283
createGetParameterDefinitionTask
 method 288, 289
createPageControl method 515, 558
createPageCustomControl method 514,
 555, 556, 558
createRenderTask method 295
createReportEngine method 283
createRotatedText method 422
createRunAndRenderTask method 295
createRunTask method 295
createSampleRuntimeSeries method 203
creating
 BIRT projects 361–362, 364–366
 charting applications 340
 charts 91, 257, 330, 338, 345
 Eclipse projects xix
 event handler classes 155, 156
 event handlers 107, 108, 127, 155, 161
 tutorial for 144–149
 Hibernate driver plus-in project 526–
 533
 Hibernate ODA extension 524
 Java applications 93, 237
 Java classes 99, 453
 lists 103
 master pages 102, 131, 298
 ODA driver extensions 479, 491, 524
 ODA driver plug-ins 481, 491
 ODA drivers 477, 480
 plug-in extensions 381–389
 property structure objects 312
 queries. *See* queries
 report designs 92, 244, 303, 319
 report elements 168, 315
 report engine 238, 239, 245, 283
 report item extensions 399, 412
 report items 91, 315, 399, 400, 421
 reporting applications 276, 277, 279
 reports 107, 276, 295
 ROM elements 100
 scripted data sets 217, 220
 scripted data sources 217, 218, 219
 stand-alone applications 93, 279
 tables 103
 update sites 396
criteria. *See* parameters
cross tabs 594
cross tabulation. *See* cross tabs; cross-tab
 reports
cross-tab elements 594
cross-tab reports 594
CSS files 278, 589
 See also cascading style sheets
cssClass attribute 75
CSV data structures 480
CSV files
 accessing data in 480, 493
 connecting to 493
 creating designs for 457

CSV files (*continued*)
developing ODA extensions for 477, 491
initializing output streams for 444
rendering options for 449
structuring 443
viewing content of 455
writing to 433, 444, 448, 455
CSV formats 434, 455
CSV ODA driver extension examples 398
CSV ODA driver extensions
compiling and debugging 481
creating 479, 491
downloading examples for 479
downloading plug-ins for 480
setting dependencies for 489
CSV ODA driver interfaces 491
CSV ODA driver plug-in project 477, 481, 501
CSV ODA driver plug-ins 481, 519
CSV ODA drivers 480, 487
CSV ODA extensions 480
CSV ODA interfaces 493
CSV ODA plug-ins 480
CSV ODA UI extension 480, 500, 509
CSV ODA UI plug-in 506, 509
csv package 443, 493
csv plug-in 450, 480
CSV report rendering extension
creating projects for 434
developing 434
implementing content interfaces for 442
launching 451, 452
overview 440
running 443, 454
setting dependencies for 437
testing 450
viewing output for 455
CSV report rendering plug-in
building 450
launching 451, 452
testing 450
csv ui plug-in 480, 501
csv ui wizards plug-in 510
CSV writer 444, 445, 446
CSVBufferReader class 493, 496
CSVFilePropertyPage class 510, 511
CSVFileSelectionPageHelper class 511, 512
CSVFileSelectionWizardPage class 511, 514
CSVPlugin class 443
CSVRenderOption class 443, 449
CSVReportEmitter class 443, 444, 446
CSVReportEmitter method 444
CSVTags class 443, 449
csvTest.reportdesign 457
CSVWriter class 444, 449
cubes 595
See also multidimensional data
current release 369
currentSession method 546
cursors 492, 498
CurveFittingViewer application 354
custom chart generator 235
custom data sources 478
See also ODA data sources
custom Java classes 227
custom report design tool 245
custom report designer 93, 98, 237
custom report generators 93, 235
custom status handlers 281
customizing
applications 93
charts 321
colors 96
ODA drivers 90, 94, 479
output formats 293, 400, 434
report emitters 94
report engine 237
report items 91, 94, 401
reports 279
table of contents 40
user interfaces 290
web viewer 63–75, 78
XML elements 97
CVS repository 369, 402

D

data

See also data elements; values
aggregating 90, 112
defined 595
exporting 41, 277, 434
extracting 240, 277, 434
filtering 90, 93, 287

data (continued)

- generating sample 203, 330, 337
- grouping 90, 93, 120, 123
- retrieving 90, 217, 226
- selecting 292
- sorting 90
- data access components 90
- data adapter API library 278
- data adapters. *See* adapter classes
- data binding. *See* column binding
- data components. *See* data elements
- data cubes 595
 - See also* multidimensional data
- data drivers 239
- data elements 103, 595
 - See also* data
- data engine 90, 478
- data engine extension 90
- Data Explorer 595
- data extension names 172
- data filters 606
- data package 258, 269
- data points 195, 324, 595
 - See also* charts
- data rows 595
 - See also* rows
- data series
 - See also* charts
 - adding 332–336
 - binding to charts 186
 - building queries for 334, 336
 - changing properties for 327
 - creating 327, 328
 - defined 324, 634
 - determining type 189
 - setting properties for 203, 335
 - setting type 258
- data series definition objects 186, 327
- data series identifier 188
- data series types 327
- data services 90
- data set adapters 164
- data set classes 316
- data set elements 104, 164, 488, 489
 - See also* data sets
- data set events 112, 113, 114, 116, 117, 118
- data set extension properties 172
- data set fields. *See* columns
- data set instance interface 172

data set objects 172

See also data sets

data set page (Hibernate UI plug-in) 525, 555

data set page (ODA UI plug-in) 480

data set parameters 596

See also parameters; report parameters

data set types 172, 492

data sets

See also data set elements; data set objects

- accessing 315
- accessing columns in 171
- binding charts to 186, 330, 339
- binding to report items 317, 318
- building programmatically 172
- building queries for 140, 172, 525
- caching 118
- changing data sources for 317
- changing properties for 317
- changing queries for 140
- closing 113, 164, 166, 227
- creating scripted 217, 220
- defined 596
- defining event handlers for 112, 113, 118, 127, 164, 165, 172
- developing ODA extensions for 90, 478, 493, 500
- fetching 172, 223, 226
- filtering data in 287
- getting data sources for 172
- getting metadata for 172
- getting names 172
- getting number of columns in 171
- getting properties for 316
- getting type 172
- opening 113, 164, 165, 226
- replacing null values in 188
- setting properties for 316
- sharing 112

data source adapters 163

data source classes 316

data source connection wizards 509

data source drivers. *See* drivers

data source elements 103, 163, 488

See also data sources

data source events 112, 113, 116, 117, 118

data source extension points 480, 487, 510, 548

data source objects 140, 596
See also data sources

data source page (Hibernate UI plug-in) 525, 555

data source page (ODA UI plug-in) 480

data source plug-ins 478, 486

data sources

See also specific data source type

accessing 103, 217, 315, 478

adding 500

changing 317

closing 112, 163, 165, 218, 221

connecting to. *See* connections

creating scripted 217, 218, 219

defined 596

defining event handlers for 112, 118, 124, 127, 163

developing ODA extensions for 478, 525

getting 172

opening 112, 163, 165, 221

retrieving data from 90, 217, 226

unsupported 90

data structures 310

Data Tools Platform 6

Data Tools Platform (BIRT)

See also DTP Integration package

Data Tools Platform (Eclipse) 90, 477, 478

data transform components 90

data type mappings 489

data types

columns and 139

CSV files and 493

defined 596

Hibernate data sources and 533

report parameters and 290

ROM metadata structures as 101

XML schemas and 370, 399

dataadapterapi.jar 278

database drivers. *See* JDBC drivers

database management systems 597

database schemas 633

databases 478

See also Classic Models sample

database; data sources

DataCharts charting example 351

DataChartsViewer application 351

datafeed package 258, 259

DataPointHints objects 195

DataSet element 104, 164, 488, 489

DataSetAdapter class 164

DataSetHandle class 315, 316

DataSetImpl class 333

DataSetMetaData class 493, 499, 534, 538

dataSetPage element 508

DataSetProcessorImpl class 188

dataSetUI element 508

dataSetUI page wizard 555

dataSetWizard attribute 508

DataSetWizardPage class 510, 558

DataSource element 103, 163, 488

dataSource extension point 480, 487, 508, 510, 548

DataSource objects 140

See also data sources

dataSource plug-in 478, 486

dataSource.exsd 478

DataSourceAdapter class 163

DataSourceEditorPage class 510

DataSourceHandle class 248, 315, 316

dataSourceUI element 508

DataSourceWizardPage class 510

datatools directory 480

dataTypeMapping elements 533

dataTypeMapping type 489

DataTypes class 535

date values 597

date-and-time data type 597

DATETIME data type 139

DBMS (defined) 597

Debug mode 451

Debug.js 60

debugger 108

debugging

applications 285

defined 597

Java event handlers 180

ODA driver extensions 481

DECIMAL data type 139

declarations 597

default engine configuration 238

default values 246, 290

DefaultDataProviderImpl class 355

defaultDisplayName attribute 487, 488

defaultDisplayName property 409

defaultValue attribute 488

defaultValue property 409

deleteGlobalVariable method 130, 169
deletePersistentGlobalVariable
method 130, 169
deleting
 cached information 12
 global variables 130, 169
 program files 15
 temporary files 238
demo database 20, 586
 See also Classic Models sample
 database
Demo Database archive 20
Demo Database package 19
dependencies (plug-ins) 381, 405
Dependencies option (PDE Editor) 381
Dependencies page (PDE Editor) 380,
 405
deploying
 applications 31, 277
 charts 322
 fragment package 572
 Hibernate ODA UI plug-in 564
 Java classes 227
 plug-in extensions 370, 393–398
 plug-in fragments 579
 report designs 302
 report item extensions 402, 429
 report viewer 7, 27
 reports 27, 156, 227
 web viewer 61, 76
Derby databases 20
derived classes. *See* descendant classes
descendant classes 597
design elements 102, 166
design engine 89, 91, 302
design engine API 235, 244, 248, 256, 275
 See also report model API
design engine api package 256
design engine class 245
design environments. *See* BIRT; Eclipse
design events 111
design files
 defined 628
 generating 87, 89, 93, 244
 generating reports from 237, 276, 286,
 295
installing report viewer and 30, 31
loading 239
naming 430
opening 98, 145, 276, 286, 304
 listing for 286, 305
overview 92
referencing in URLs 31
renaming 339
running 240, 295
setting location of 31
validating 89
design interfaces 167–168
design model objects 244
design perspective 10
design properties 102
design tools 87, 244, 245
DesignChoiceConstants interface 307,
 310
DesignConfig objects 304
DesignElement element 102
DesignElementHandle class 247, 309, 419
DesignEngine class 245, 304
DesignEngine objects 304
 __designer parameter 47
designer ui extensions package 419
designers 3, 87, 93, 237
 See also BIRT Report Designer; BIRT
 RCP Report Designer
designing reports 87
 See also designs
designs
 See also page layouts
 accessing 246, 275, 302, 304
 accessing properties for 247
 accessing ROM schema for 98
 adding charts to 339, 351, 352, 355
 adding report items to 305, 315
 changing 132, 279, 302, 305
 creating 244, 303, 319
 defined 597
 defining event handlers for 108, 127,
 132, 155, 166, 174
 deploying 302
 generating CSV files and 457
 getting parameters in 289
 initializing 143
 retrieving external data for 478
 reusing 88, 92
 saving 303, 318, 339
 setting location of 31
 setting properties for 286
 standardizing 92

designs (*continued*)
testing for parameters in 289
validating 98, 101
viewing report items in 117
desktop applications. *See Java applets*
desktop reporting application 237
destroy method 238
detail reports. *See subreports*
detail rows 119, 121
developing
applications 235, 275, 277, 369
Hibernate ODA UI extensions 548, 554
ODA extensions 477, 479, 524
plug-ins 370, 377
rendering extensions 433, 434, 441, 465
report designs 244
reports 107
development environments 108
development languages 613
See also scripting languages
development tools 369
device package 258, 259, 260
DHTML (defined) 599
dial charts 321
See also charts
DialChart interface 257, 325
DialChartImpl class 324, 325
DialChartViewer application 354
Dimension attribute (charts) 202, 203
dimension property 325, 326
dimensions 598
See also multidimensional data
directories
accessing fonts and 299
accessing Java classes and 227
accessing report designs and 30, 31
deploying applications and 277, 279
displaying reports and 28
installing language packs and 14
installing plug-ins and 369, 370, 372
installing report viewer and 30
Java classes and 152
running web viewer and 54, 56
saving temporary files and 281
unpacking program archives and 5, 6,
11
updating designers and 15
directory paths. *See paths*
disconnect method 230
disk writes 237
display names 488, 598
displaying
charts 89, 191
error messages 146, 149
extension point descriptions 385, 386
HTML pages 34
HTML reports 88
PDF files 34
PDF reports 301
project settings 380
property annotations 387
reports 33, 44, 87, 89
displaytext attribute
param tag 70
paramDef tag 75
displayText property 409
distributing reports. *See deploying*
_document parameter 47
document files
accessing data in 240
creating 277, 295
defined 628
generating reports from 109, 275, 276,
286, 287
opening 239, 276, 286
overview 92
writing to disk 240
document model (BIRT). *See ROM*
document object model. *See DOM*
document type definitions 598
documentation xxii, 18, 236
documentation attribute 370
documents 66, 275, 287, 301, 434
See also reports
DOM (defined) 598
domain names 599
double quotation mark ("") character
command line arguments and 22
JavaScript code and 329
downloading
Apache Tomcat servers 27
BIRT Full Eclipse Install 9
BIRT RCP Report Designer 11
BIRT Report Designer packages 17
CSV ODA driver examples 479
CSV ODA driver plug-in 480
designer updates 15
event handler classes 156

downloading (*continued*)
extension examples 398
language packs 13
source code 364, 402
Web Viewer application 44
`_dpi` parameter 47
`drawDataPointLabel` events 195, 196
Driver class 493, 494
driverClass attribute 487
drivers
 See also specific type
 accessing external sources and 478
 accessing web service data sources and 230
 creating ODA 90, 477, 480, 524
 customizing 94
 defined 599
 installing 30
 registering 544
 required 278
 setting location of 239
 specifying interfaces for 491
drivers directory 29, 30, 544
drivers subdirectory 278
DTD (defined) 598
`dteapi.jar` 278
DTP Integration archive 19
DTP Integration package 6, 19
DTP ODA classes 477
DTP ODA extension points 478
DTP ODA framework 509
DTP ODA interfaces 477, 491
dynamic hypertext markup language. *See* DHTML
dynamic text elements 599
dynamic variables 599

E

e.reports. *See* reports
Eclipse compiler 362
Eclipse Data Tools Platform 90, 477, 478
Eclipse environments 108, 599
Eclipse Modeling Framework 6, 600
 See also EMF libraries
Eclipse perspective 600
Eclipse platform 84, 369, 600
 See also Eclipse Rich Client Platform
Eclipse Plug-in Development Environment 369, 377, 601

See also PDE Workbench
Eclipse projects xix, 83, 601
 See also projects
Eclipse Rich Client Platform 601
 See also Rich Client Platforms
Eclipse SDK 6
Eclipse views 379, 601
Eclipse Workbench 107, 379, 429, 601
Eclipse workspace 362, 602
ECMAScript implementation 127
`ecore.jar` 278
Edit Script command 224
editor attribute 409
editor pages 508
editors. *See* specific application editor
EJBs 217, 603
element 371
element definitions (ROM) 100
Element element 100
element method definitions (ROM) 100
ElementDetailHandle class 254
ElementFactory class 315
ElementFactory objects 338
elements
 See also report elements; ROM elements
 accessing CSV files and 487
 customizing plug-ins and 370
 defined 602
 defining plug-in extension 385
 plug-in extension points and 388, 399
ellipsis (...) button 602
embeddable HTML output 297
embedded fonts 299, 300
embedded HTML 297
embedded report engine 237
EMF (defined) 600
EMF libraries 278
EMF software 6
emitter csv plug-in 443, 450
emitter extension points 434, 438
emitter extensions 400, 462, 465
emitter interfaces 440, 441
emitter objects 444
emitter plug-in 441
emitter xml plug-in 466
emitters
 adding 434
 customizing 94

emitters (*continued*)
defining page breaks and 120
extending functionality of 434
generating CSV output and 438, 443, 449
generating reports and 89, 124, 281, 282
generating XML output and 466
setting properties for 440, 465
emitters package 438
emitters plug-in 400
emitters.exsd 400, 434
encapsulation 602
encoding 640
end method 446, 468
endCell method 448
endRow method 449
engine API library 278
engine api package 235, 241, 242
engine APIs. *See also* chart engine API; report engine API
engine extension package 419
engine plug-ins 238, 245
engine servlet 45
engine task processes 108–110
engine variable 300
engineapi.jar 278
EngineConfig class 134, 238, 245
EngineConfig objects 238, 279, 280, 285
EngineConstants class 295
EngineException exceptions 300
EngineFragment plug-in 50, 51, 52
engines 84, 88
See also specific engine
EngineTask class 134
enterprise 602
Enterprise JavaBeans. *See* EJBs
enterprise reporting 603
enterprise systems 602
enumeration classes 257, 259
environment settings 34
environment-dependent processing 237
environments 625
error messages 146, 149, 493
errors 12, 146
evaluateQuery method 293
event handler classes
accessing 159
associating with report elements 160
creating 155, 156
downloading 156
naming 161
event handler examples 173
event handler interfaces 156, 161, 162, 166
event handlers
See also events
accessing 108
accessing JAR files for 156
accessing methods for 153
adding logging code to 124
adding to designs 108, 127, 166
associating context objects with 130, 169
building charts and 183, 188, 193, 200, 204, 208, 212, 329, 353
building data sets and 112, 113, 118, 164, 165, 172
building data sources and 112, 118, 163, 164
changing column bindings and 119
creating 107, 108, 127, 155, 161
tutorial for 144–149
debugging Java 180
defined 603
rendering report elements and 97
rendering report items and 97, 162
sharing data and 178
validating report parameters and 111
event listeners 604
event order sequence 115–126, 129
event package 258, 259, 260
event types 110, 115
events
See also event handlers
building data rows and 121
creating charts and 121
defined 603
designing reports and 111, 166, 174
displaying web pages and 55, 56
firing 97, 108, 110, 115, 117
generating reports and 110, 116, 117, 124
generating tables or lists and 122, 123
grouping data and 123
retrieving data and 112, 113, 116
running reports and 129
scripting for 107, 108, 144, 161

events (*continued*)
 subscribing to 375
example charting applications 18
example database
 See also Classic Models sample
 database
example extensions 398
exception package 258, 262
exceptions 309, 493, 604
executeQuery method 229, 496, 541
ExecuteReport class 455
executing reports 129
execution processes 129
experts. *See* wizards
Export Data dialog 41
export options 394
Export Report dialog 42
Export Wizard (PDE) 393
 __exportEncoding parameter 47
exporting data 41, 277, 434
exporting fragment extension 572
exporting plug-in extensions 393, 402
exporting reports 42
Exporting section (Overview) 394
expression builder 604
expression property type 97
expressions
 binding data and 119
 defined 604
 manipulating numeric values
 with 621
 manipulating string data with 637
 matching text patterns and 628
 returning Boolean values from 585
 setting properties and 97
.exsd files 370
extended-item name element 98
ExtendedItemHandle class 324, 419
ExtendedItemHandle objects 330, 338,
 339
extensible markup language. *See* XML
extension APIs 236
extension element 373
Extension Element Details section 385
extension IDs 408, 438, 486, 506
extension names 172, 371
extension package (report engine) 401
extension point schema definitions 370,
 385

Extension Point Selection page 383, 439,
 465
extension points
 accessing external sources and 478
 accessing Hibernate data sources
 and 548, 550
 adding report items and 399
 customizing ODA drivers and 90
 defined 605
 defining 381
 displaying descriptions of 385, 386
 finding 385
 generating output and 434, 438
 implementing 369, 399
 selecting 386, 410
Extension Points page (New
 Extension) 410
Extension Points page (PDE Editor) 381
extensionName property 409
extensionProperties array 140
extensions
 adding chart items and 91
 adding report items and 399, 401
 building plug-in 390–392
 creating 98, 381, 385, 399
 customizing report items and 91
 declaring 381
 defined 605
 deploying 370, 393, 398
 developing ODA 90, 477, 479, 524
 expanding BIRT framework and 93
 naming 408, 438, 506
 overview 369
 rendering reports and 433, 454
 sample projects for 398
 selecting export options for 394
 setting class attributes for 388
 setting contents of 386
 specifying 383
 structuring 381–384
 testing 392
extensions package 419
Extensions page (PDE Editor) 381, 383,
 385
external data sources 103, 478, 491
external libraries 151
external objects 201

F

factory method 240
factory package 258, 262
Factory processes 129, 133
Feature License dialog 15
Feature Updates dialog 14
Feature Updates page 395
features 7, 395
fetch events 113, 114
fetch method
 creating scripted data sets and 114
 retrieving data rows and 113, 218, 223,
 226
 writing event handlers and 165
field names. *See* column names
field types 171
field variables 605
 See also member variables; variables
fields
 See also columns; computed fields
 accessing CSV data and 480
 changing data sources and 317
 defined 605
 exporting CSV output and 440, 442
File class 493, 494
file objects 493, 494
file types 91, 606
FileDialog method 556
files
 See also specific type
 accessing resource 281
 changing source code and 367
 creating 143
 downloading designer 9, 11
 extracting program archive 5, 6, 11
 generating output and 275
 mapping to report viewer 31
 overview 91–92
 removing program 15
 rendering output and 276
 tracking method execution in 142, 144
 updating designers and 14
FileSelectionWizardPage class 511, 515
filter element 508
filtering data 90, 93, 287
filters 606
finalization code 142, 144, 227
Find and Install command 395

Find declaring extension point option 385
findDataSet method 315
findDataSource method 315
findElement method 306
finding extension points 385
finding program updates 14
findTOC method 287
 __fittopage parameter 47
flat file data sources 477
flat file plug-in 477
flat files 606
 See also flat file data sources; text files
FLOAT data type 139
flute.jar 278
folders 15, 391
font files 33
font style constants 310
fontFamily style specification 99
fonts 33, 299, 606
footer rows 121, 123
footers 298, 339, 606
forceOverwriteDocument attribute
 parameterPage tag 73
 viewer tag 65
form e-capable browser 607
 See also web browsers
 __format parameter 47
format attribute
 parameterPage tag 73
 report tag 68
 viewer tag 65
format property 438
format styles. *See* styles
FormatCharts charting example 352
FormatCharts class 352
FormatChartsViewer class 352
formatQueryText method 495
formats
 customizing report emitters for 89
 customizing report generators for 93
 defined 607
 developing report engine for 109, 433,
 462
 exporting data and 434
 generating reports and 275, 276, 293
 getting output 131, 170
 rendering images and 297
formatted output 275, 300, 301

Formula Editor. *See* expression builder
formulas. *See* expressions
fragment 571, 607
fragment example 572, 576
fragment projects 573
fragment tag 571
frame objects 152
frameborder attribute
 parameterPage tag 73
 report tag 68
 viewer tag 65
frameset servlet 34, 109
FramesetFragment plug-in 50, 52
framework 607
Full Eclipse Install (BIRT Report Designer) 9–10
full outer joins. *See* outer joins
function stubs 156
functions
 See also methods
 accessing 135
 defined 607
 handling chart events and 329
fundamental data types. *See* data types

G

GeneratedChartState objects 189, 191
generating
 charts 87, 90, 353
 CSV files 433, 444, 448, 455
 design files 244
 formatted output 300
 HTML reports 239, 241, 293, 297
 lists 122–124
 output 275, 433
 PDF documents 239, 241, 293
 report design files 89, 93, 244
 report elements 116
 report items 117
 reports 93, 116, 276, 279, 300
 sample data 337
 tables 122–124
generation engine 86
generation phase (events) 110, 116, 124
generation services 89
generators (custom) 235
genReport script 22
get method 306
getAllExtensionProperties method 172

getAppContext method 130, 134, 169
getBlock method 201
getBody method 306
getBundle method 153
getCategory method 213
getCategoryAxis method 214
getCategoryProvider method 425
getChartInstance method 201
getChildren method 287
getColumnAlias method 139, 171
getColumnCount method 139, 171, 173, 447
getColumnLabel method 139, 171
getColumnMetaData method 139, 140, 172
getColumnName method 139, 171, 173, 499
getColumnNativeTypeName
 method 139, 171
getColumnType method 139, 171
getColumnTypeName method 139, 171
getColumnValue method 172, 173
getConnection method 494, 499, 535
getContents method 290
getDataSet method 173
getDataSets method 316, 317, 339
getDataSource method 172
getDataSources method 316, 317
getDataType method 290
getDefaultValue method 290
getDefaultValues method 290
getDescription method 201, 213
getDesignHandle method 305
getDimension method 202, 213
getElementFactory method 338
getExportLabelTag method 469
getExtendedProperties method 202
getExtensionID method 172
getExtensionProperty method 172
getExternalContext method 200, 201
getFactory method 214
getGlobalVariable method 130, 169
getGridColumnCount method 202
getHibernateProp method 547
getHibernatePropTypes method 547
getHttpServletRequest method 130, 169
getInteractivity method 202
getLabel method 290, 421
getLabelPropValue method 470

getLegend method 202, 213, 331
getLocale method 130, 170, 201
getLogger method 201
getMaxConnections method 535
getMaxQueries method 537
getMeasure method 310
getMessage method 130, 170
getMetaData method 537, 541
getName method 172
getNext method 306
getOutline method 332
getOutputFormat method 130, 131, 170
getOutputType method 213
getPageNumber method 287
getParameterDefn method 289
getParameterDefns method 289
getParameterType method 290
getParameterValue method 130, 141, 170
getParameterValues method 291
getPersistentGlobalVariable method 131, 133, 170
getPlot method 202, 331
getPrimaryBaseAxes method 332
getPrimaryOrthogonalAxis method 332
getProperty method 286, 324
getPropertyHandle method 311
getQuery method 516
getQueryColumnNames method 517
getQueryString method 135
getQueryText method 172
getRenderOption method 170
getReportRunnable method 170
getResult method 542
getResultSetColumn method 318
getResultSetExpression method 318
getResultSetMetaData method 517, 563
getRow method 498
getRunTimeSeries method 187
getSampleData method 202
getScript method 202
getSelectionList method 290
getSelectionListForCascadingGroup method 293
getSeriesDefinitions method 335
getSeriesForLegend method 202
getSeriesPalette method 335
getSeriesThickness method 202
getSQLStateType method 538
getStatus method 301
getString method 499, 542
getStyle method 307
getStyles method 202
getSubType method 202
getSupportedImageFormats method 297
getTaskType method 170
getText method 307, 469
getTitle method 202, 213
getType method 198, 202
getUnits method 202
getURI method 307
getValue method 290
getValueAxes method 214
getValueSeries method 214
getVersion method 202
getWidth method 307, 310
global options (report engine) 238, 245
global variables 128, 133, 169, 607
See also variables
Glossary 583
glyph 608
See also character sets; fonts
grandchild classes. *See* descendant classes
grandparent classes. *See* ancestor classes
Graphical Editing Framework 6
graphical report design tool 87
graphical user interfaces. *See* user interfaces
graphics. *See* images
GraphicsUtil class 422
graphs. *See* charts
grid cells. *See* cells
grid elements 608
grid items 102, 117, 315
GridColumnCount attribute 202, 203
grids 102, 608
See also grid elements; grid items
group fields. *See* group keys
group footers 606
group headers 609
group keys 609
group sections 120
group slots 307
grouped reports 609
grouping data 90, 93, 120, 123
See also groups
GroupOnXSeries charting example 351
GroupOnXSeries.rptdesign 351
GroupOnYAxis charting example 352

GroupOnYAxis.rptdesign 352
groups
 adding page breaks for 120
 binding data and 120
 building data rows for 121
 defined 608
 generating tables or lists and 123
GUI components 90
 See also user interfaces

H

handle classes 248
handle objects 310
handler class 155
 See also event handler classes
hardware interfaces 612
HashMap value 241
header rows 121, 123
headers 299, 609
headings. *See* column headings
height attribute
 parameterPage tag 72
 report tag 68
 viewer tag 65
help 622
hexadecimal numbers 609
Hibernate Core for Java 478
Hibernate data sets 525, 554
Hibernate data sources 524
Hibernate data types 533
Hibernate libraries 547
Hibernate objects 217
Hibernate ODA driver plug-in 526, 547
Hibernate ODA drivers 524, 525, 535, 544, 547
Hibernate ODA extension example 534
Hibernate ODA extensions 524, 525
Hibernate ODA UI example 525, 526
Hibernate ODA UI extension points 550
Hibernate ODA UI extensions 548
Hibernate ODA UI plug-in
 building 564
 creating projects for 548
 deploying 564
 described 524
 developing 554
 launching 566
 specifying dependencies for 550
 specifying run-time settings for 550

testing 566–570
hibernate package 534
hibernate plug-in 524, 526, 547
Hibernate Query Language. *See* HQL statements
hibernate ui plug-in 524
hibernate.cfg 525
HibernateClassSelectionPage class 558
HibernateDataSourceWizard class 554, 558
HibernateDriver class 534, 535
HibernateHqlSelectionPage class 554
HibernatePageHelper class 554, 555, 558
HibernatePropertyPage class 554, 558
HibernateUtil class 535, 536, 544, 547
hibfiles directory 525, 544
hierarchy 609
highlight rules 312
HOME property 488
homeDir parameter 494
host instance (PDE Workbench) 379
HQL (defined) 478
HQL statements
 adding user interface for 554
 creating 524, 536
 executing 541
 retrieving data with 542
 verifying 560
HTML (defined) 610
HTML code 74
HTML elements 602
HTML emitters 89, 282
HTML formats 433
HTML frames 34
HTML reports
 See also web pages
 configuring properties for 300
 displaying 88
 generating 239, 241, 293, 297
 opening 301
 paginating 298
 referencing images in 282
 rendering unpaginated 241
 setting up rendering context for 297
 viewing 34
 writing to disk 237
HTML tables 299
HTML tags 298, 607

HTML_ENABLE_AGENTSTYLE
_ENGINE parameter 54
HTMLActionHandler class 299
HTMLCompleteImageHander class 282
HTMLRenderContext class 297
HTMLRenderContext objects 297
HTMLRenderOption class 276, 297
HTMLRenderOption objects 282
HTMLServerImageHandler class 282
HTTP (defined) 610
HTTP request objects 135
HttpServletRequest objects 130, 169
hyperlinks 299, 440, 443, 610
hypertext markup language pages. *See*
web pages
hypertext markup language. *See* HTML
hypertext transfer protocol. *See* HTTP

I

IBandContent interface 442
IBM WebSphere servers 27
ICascadingParameterGroup
interface 289
ICell interface 167
ICellContent interface 442
ICellInstance interface 168
IChart interface 212, 213
IChartScriptContext interface 201
IChartWithAxes interface 213, 214
IChartWithoutAxes interface 213, 214
IColumnMetaData interface 139, 171
IConnection interface 491, 534
icons 90, 409
IContainerContent interface 442
IContent interface 442
IContentEmitter interface 441
IContentVisitor interface 442
ICU library 278
icu.jar 278
_id parameter 47
id attribute
 dataSet UI extension 508
 ODA data set extension 488
 ODA data source extension 487
param tag 70
paramDef tag 76
parameterPage tag 72
report tag 68
viewer tag 65

ID property 408, 438, 486, 506
id property 438
IDataContent interface 442
IDataExtractionTask interface 240
IDataExtractionTask objects 277
IDataSetEventHandler interface 164, 165
IDataSetInstance interface 172
IDataSetMetaData interface 492, 534
IDataSetRow interface 172
IDataSourceEventHandler interface 163,
 164
identifiers 610
IDeviceRenderer interface 191
IDriver interface 487, 491, 534, 535
IEmitterServices interface 441
IEngineTask interface 240
IExternalContext objects 201
IForeignContent interface 442
IGetParameterDefinitionTask
 interface 240
IGetParameterDefinitionTask objects 289
IGetParameterDefnTask interface 288
IHTMLImageHandler interface 282
IHyperlinkAction interface 443
IIImageContent interface 442
ILabelContent interface 442
image constants 307
image elements 610
 See also images
image files 31, 282
image formats 297
image handlers 282, 307
ImageHandle objects 307
 _imageID parameter 47
images
 defined 610
 exporting to sample CSV report
 rendering extension and 440
 handling events for 175
 rendering context and 297, 298
 rendering rotated text as 422
 saving 282
IMessages interface 419
impl packages 257, 259
import statements 151
importing Java packages 151, 226
importPackage method 151, 226
in_count parameter 228
incrementing record counters 146

index.jsp 30
information. *See* data
inheritance 611
initCustomControl method 514, 555, 556
initialization code 142, 226
initialize events 111, 116, 124, 166
initialize method
 building report designs and 143, 166
 creating output streams and 444, 467
 importing Java packages and 152
 method execution sequence and 142,
 143
 writing event handlers and 111, 115
initializeControl method 560
initializing report elements 115
initSessionFactory method 544
inner joins 611
 See also joins
inner reports. *See* subreports
input parameters 227
input sources. *See* data sources
input streams 319
inputParams array 227, 228
installation
 BIRT Chart Engine 17–18
 BIRT components 5
 BIRT Full Eclipse Install 9
 BIRT RCP Report Designer 10–11
 BIRT Report Designer 9–10, 362
 BIRT Samples package 23
 JDBC drivers 30
 language packs 13–14
 plug-ins 277, 372
 report viewer 28–30
 source code 23–24
 testing 10, 11, 13, 20, 21
 troubleshooting 11–12
 TrueType fonts 33
 upgrades and 14, 15
instance interfaces 168
instance property 330, 338
instance variables 612
 See also class variables; variables
instances. *See* objects
instantiation 612
INTEGER data type 139
integrated debugger 108
interactive features (charts) 202
InteractivityCharts charting example 352

interfaces
 See also application programming
 interfaces; specific programming
 interface
 adapter classes compared to 162
 chart engine API and 257, 324
 CSV report rendering extension 441,
 442
 data row objects and 172, 173
 data set objects and 172
 defined 612
 design engine API 256
 design model objects and 244
 developing with 235
 event handlers and 155, 156, 162, 166
 extending adapter classes and 156
 Hibernate ODA drivers and 535
 hierarchical diagrams for 236
 naming conventions for 161
 ODA extensions and 477, 486, 491
 overview 166
 report elements and 167–169
 report engine API 242
 rotated label plug-in 419
 run-time drivers and 491
International Components for Unicode.
 See ICU library
internationalization 612
 See also locales
IP addresses 612
IPageContent interface 442
IParameterDefnBase interface 289
IParameterGroupDefn interface 289
IParameterMetaData interface 492
IParameterSelectionChoice interface 289,
 290
IPlatformContext interface 280, 281
IQuery interface 491, 534
IRenderOption interface 297
IRenderOption objects 276
IRenderTask interface 241, 295, 297
IRenderTask objects 276
IReportContent interface 443
IReportContext interface 169
IReportDocument interface 239
IReportDocument objects 276, 287
IReportElement interface 167, 168
IReportEngineFactory interface 283
IReportEventHandler interface 166

IReportItem interface 420
IReportItemFactory interface 420
IReportItemLabelProvider interface 419
IReportItemPresentation interface 419
IReportRunnable interface 239, 286
IReportRunnable objects 276, 286
IResultSet interface 492, 534
IResultSetMetaData interface 492, 534
IRowContent interface 442
IRowData interface 173
IRowInstance interface 169
IRowSet interface 419
IRunAndRenderTask interface 241, 295, 297
IRunAndRenderTask objects 276, 300
IRunTask interface 240, 295
IScalarParameterDefn interface 289
IScalarParameterDefn objects 290
isColorByCategory method 213
isComputedColumn method 139, 171
IScriptedDataSourceEventHandler interface 164, 165
isCustom attribute
 paramDef tag 76
 parameterPage tag 71, 73
isEmpty method 289
isEncryptable attribute 488
isHorizontal method 214
isHostPage attribute
 report tag 68
 viewer tag 65
 _islocale parameter 47
isLocale attribute
 param tag 70
 paramDef tag 76
 _isnull parameter 47
 _isreportlet parameter 48
isRowInFooterBand method 447
IStatusHandler interface 281
IStyle interface 443
IStyleModel interface 420
ITableContent interface 442
iterator method 306, 542
iterator objects 226
ITextContent interface 442
ITextItem interface 167

J

J2EE applications 21, 33

J2EE environments 613
J2SE environments 613
.jar files 12
 adding to classpaths 157
 building update sites and 396
 changing source code and 367
 charting applications and 322
 configuring report engine and 238, 245
 creating event handlers and 156, 204
 default location for 152
 defined 613
 deploying Java classes and 227
 deploying plug-ins and 370, 383, 393
 deploying to JBoss servers and 28
 developing ODA extensions and 481
 generating 370
 installing JDBC drivers and 30
 selecting external 452
JAR Selection dialog 452
Java. *See* Java programming language
Java 2 Enterprise Edition. *See* J2EE environments
Java 2 Runtime Standard Edition. *See* J2SE environments
Java APIs 88
Java applets 584
Java applications
 See also applications
 adding charting capabilities to 17, 321
 adding reporting capabilities to 21
 creating 93, 237
 generating designs and 89
Java archives. *See* .jar files
Java Attribute Editor 388
Java Build Path page 156
Java Class dialog 158
Java classes
 See also classes
 accessing 151, 152, 159, 226, 295
 associating with report elements 160
 creating 99, 453
 deploying 227
 developing ODA extensions and 480
 developing with 235
 handling events and 155, 156
 naming 158
 referencing 151, 226
 registering 329

Java classes (*continued*)
scripting for 151, 152
setting location of 152
setting properties for 389
Java code 108, 151, 153, 155, 613
Java command 452
Java compiler 159
Java Database Connectivity. *See* JDBC
Java development environment 108
Java Development Kit 7
Java Development Kit. *See* JDK software
Java editor 159
Java event handler classes 155, 159, 163
Java event handler examples 173
Java event handlers
building data sets and 164
building data sources and 164
compared with JavaScript 107
creating 108, 155
debugging 180
rendering charts and 204, 329
Java interfaces 166, 235, 612
Java methods 152
Java Naming and Directory Interface 614
Java naming conventions 161
Java objects 133, 152, 217, 226
Java packages 151, 226
Java perspective 452
Java programming language 613
Java programs 614
Java Project option 452
Java projects 157, 159
Java report generator 93
Java run-time API 375
Java Runtime Environment 27
Java Settings page 452
Java source files 158
Java Virtual Machines. *See* JVMs
`java.lang` package 152
JavaBeans 614
JavaScript
See also source code
accessing data sources and 217
accessing Java classes for 151
accessing ROM elements and 135
defined 614
entering 136
line breaks in 329
previewing 138
setting properties with 136
tracking method execution and 143
tutorial for 144
variables in 128, 133
wrapping Java code in 151, 153
writing event handlers and 107, 108, 127, 208, 329
JavaScript library 278
JavaScript names 138
JavaScript objects 152
JavaScript palette 147
JavaScriptViewer application 353
JavaServer Pages. *See* JSPs
JavaViewer application 353
JBoss servers 27, 28
JDBC (defined) 614
JDBC connections 140
JDBC data sources 34, 478
JDBC drivers 30, 525, 544
JDK software 362, 614
JNDI (defined) 614
JNDI connection objects 36
JNDI property 35
JNDI service providers 34, 35
`jndi.properties` file 35
join conditions 615
joins 104, 615
joint data sets 104, 611, 615
JointDataSet element 104
JRE software 27
`js.jar` 278
JSPs 615
JVMs 12, 614

K

keywords 615

L

label elements 615
label items 315
labels 196, 421, 615
LabelStyleProcessor class 353
language packs 13–14
language-specific environments. *See* locales
Launch an Eclipse application option 393
layout editor 400, 616
layout package 258, 270

Layout page. *See* layout editor
layouts 616
See also page layouts
lazy load 616
left attribute
 parameterPage tag 73
 report tag 68
 viewer tag 65
left outer joins. *See* outer joins
legend area (charts) 196, 202, 327, 331
Legend interface 327
legend line properties (charts) 332
Legend objects 331
LegendEntryRenderingHints objects 196
Level class 285
level-break listings. *See* grouped reports
lib directory 156
libraries
 See also component libraries
 accessing 92, 275
 accessing properties in 247
 building plug-in extensions and 380,
 390
 changing 92
 creating reporting applications
 and 275, 303
 customizing web viewer and 63
 defined 616
 deploying applications and 277
 deploying reports and 156
 developing plug-in fragments and 572
 naming 391
 required 277
 reusing designs and 88
 running Hibernate drivers and 547
 scripting and 151
 selecting 391
Libraries page (Java Settings) 452
library files 92, 303, 616
LibraryHandle class 247, 303
licenses 15
line break characters 329
line breaks (JavaScript) 329
line charts 321
 See also charts
LineAttribute objects 332
LineSeries objects 333, 334
links. *See* hyperlinks
list elements 102, 315, 617
list items 103, 117
listeners. *See* event listeners
Listing element 102
listing reports 123, 617
ListingGroup items 123
lists
 See also list elements; list items
 adding to reports 103
 building data rows for 121
 generating 122–124
 getting column information for 138
 handling events for 119, 122, 123
load_classicmodels.sql 20
loadClass method 154
LoadExportSchema class 466, 472
loading
 class files 388
 document files 239
 plug-ins 382
 report designs 145, 239, 276, 286, 304
 code example for 286, 305
 source code 373
local variables 128, 617
 See also global variables; variables
_locale parameter 48
locale attribute
 parameterPage tag 73
 report tag 68
 viewer tag 65
locale codes 14
Locale objects 201
locale-independent formats 289
locales
 building charts for 200
 changing 14
 converting strings and 289
 defined 617
 getting 130, 170
 installing language packs for 13–14
 managing programmatically 240
localization 13, 618
 See also locales
locating extension points 385
locating program updates 14
log files 31, 280, 285
log messages 201, 239, 285
log package 258, 263
Logger class 285
Logger objects 201

logging classes 285
logging code 124
logging configurations 286
logging levels 286
Logging property type 280
logging threshold 285

M

Manage Configuration command 395
manifest files 373, 377, 379, 381, 618
manifest headers 379
`manifest.mf` 379, 381
mapping to report viewer 31
margins 298
markers (charts) 202
markup languages 599, 605, 610
 See also elements; tags
`Mask.js` 57
master pages
 adding report items to 315
 creating 102, 131, 298
 defined 618
 preparing 116
 setting margins for 298
 `_masterpage` parameter 48
`MasterPage` element 102
Math class 152
matrix reports. *See* cross-tab reports
 `_maxrows` parameter 48
measures 618
Member element 101
member variables 619
 See also field variables; variables
members 101, 619
messages 130, 170
 See also error messages; log messages
Messages class
 CSV ODA driver extension 493, 499
 CSV ODA UI extension 509
 Hibernate ODA extension 535, 555
messages.properties file 509, 555
metadata
 accessing Hibernate data sources
 and 541
 accessing ODA data sources and 492, 493
 defined 619
 defining ROM elements and 98–101
 examining data row 240

getting column information from 139, 171
getting data set 172
metadata interface 171
meter charts 326, 327
 See also charts
MeterChartExample application 356
method definitions 100
Method element 100
Method metadata definition (ROM) 100
Method property 103
methods
 See also functions
accessing column information
 and 139, 171
accessing data sets and 164, 165, 172, 218
accessing data sources and 163, 165
accessing JavaScript objects and 152
accessing report components and 247
accessing report context and 130
accessing report designs and 246
accessing report items and 305
building charts and 200, 201, 202, 203, 213, 326
building data rows and 172, 173
calling 153
creating event handlers and 155, 161, 169
creating report designs and 166
defined 619
defining ROM elements and 100
executing reports and 115
generating report elements and 167, 168
generating report items and 162
importing Java packages and 152
overriding 161
providing external values for 281
rendering CSV output and 444, 446
rendering XML output and 466, 468
running reports and 135
running rotated text plug-in and 375
scripting and 151
selecting 136, 137
tracking execution of 142–144
viewing arguments for 162
methods array 96

Microsoft Windows. *See* Windows systems
milestone build 7
milestone releases 15
MIME types 438
mimeType property 438
modal windows 620
mode 620
model attribute 371
model element 408
model extension element 408
model packages (charts) 258, 259, 263
modelapi.jar 278
modeless windows 620
modelodaapi.jar 278
ModifyListener method 555
ModuleHandle class 246
Mozilla Rhino 127, 151, 153
multicolumn page layouts 102
multidimensional data 620
multilingual reports. *See* locales;
localization
multipage reports 44
Multipurpose Internet Mail Extensions.
See MIME types
multithreaded applications 620
myChart.chart file 323
mysql command line interface 20
MySQL databases 20
MySQL installation scripts 547

N

name attribute
 dataSource element 488
 param tag 70
 paramDef tag 76
 parameterPage tag 73
name collisions 152
name element 409
Name property 408, 438, 487, 506
name property 409
name variable 300
names
 See also aliases; display names
 changing context root 30
 retrieving values and 172
name-value pairs 96
naming
 event handler classes 161

Java classes 158
ODA data source extensions 487
ODA data source UI extensions 506
output files 297
plug-in libraries 391
report design files 430
report item extensions 371, 408
report items 306
report rendering extensions 438
scripted data sets 220
scripted data sources 220
naming conventions 161
National Language fragment 572, 576
nativeDataType property 489
nativeDataTypeCode property 489
 navigationbar parameter 48
Navigator 145, 620
Navigator command 145
New Chart wizard. *See* chart builder
New Class command 453
New Data Set wizard 220
New Data Source wizard 219
New Extension wizard 383
New Java Class wizard 453
New Java Project wizard 452
new line characters 449
new line tag 449
New Plug-in Project wizard 377, 378, 379
New Project wizard 377
New Report command 430
New Report wizard 430
New Source Folder dialog 391
New Update Site wizard 396
newElement method 315
newExtendedItem method 338
newInstance method 154
newQuery method 494, 536
newReportItem method 421
newSession method 245, 305
newsgroups xix
newWizard element 507
next method 498, 542
nightly build 7
-nl command line argument 14
NL1 fragment 572, 576
node 621
NonGroupOnXSeries.rptdesign 351
NonGroupOnYAxis.rptdesign 352
non-scripted data set events 113

non-scripted data sets 217
non-scripted data source events 112
non-scripted data sources 217
non-visual elements 95, 304
null values 139, 188, 621
Number class 152
numeric data types 621
numeric expressions 621
N-up reports. *See* multicolumn page layouts

O

Object class 152
object libraries. *See* component libraries
object references 227
object-oriented programming 621
objects
 accessing 217, 226
 adding to application context 134
 defined 621
 instantiating 226, 373
 registering service 375
 saving 170, 171
octal numbers 621
ODA (defined) 622
ODA API 477, 478, 491
ODA API Reference 316, 479
oda csv package 493
oda csv plug-in 480
ODA data set extension elements 488, 489
ODA data set extensions. *See* ODA extensions
ODA data sets 317, 488, 492, 493, 500
ODA data source extension elements 487, 488
ODA data source extension points 486, 548
ODA data source extensions. *See* ODA extensions
ODA data sources
 accessing 478, 486
 adding 500
 committing changes to 491
 connecting to 491, 493
 getting columns in 492
 moving through rows in 492
 querying 491, 493, 496
 retrieving data from 477, 491

setting connection properties for 140
oda dataSource extension point 480
oda dataSource plug-in 478, 486
ODA driver constants 500
ODA driver extension examples 398, 479
ODA driver extensions
 compiling and debugging 481
 creating 479, 491
 setting dependencies for 489
ODA driver interfaces 491
ODA driver plug-in project 477, 481, 501
ODA driver plug-ins 481, 519
ODA drivers
 adding connection profile for 478
 adding connection properties page for 479
 adding to BIRT framework 478
 adding user interface for 478, 487
 connecting to 491, 493
 creating 90, 477, 480, 524
 customizing 94, 479
 defined 622
 defining error messages for 493
 setting extension elements for 487
 specifying run-time interface for 491
ODA extension example (Hibernate) 526
ODA extension identifiers 487
ODA extension points 478
ODA extensions
 See also ODA driver extensions
 accessing CSV data sources and 480
 adding user interfaces for 480
 building 481
 data type mappings and 489
 defining run-time settings for 487
 developing 477, 479, 524
 display names for 488
 downloading code for 479, 480
 overview 478
 run-time properties and 140
 setting display names for 488
 setting properties for 486
ODA framework 90
ODA interfaces 487, 491, 493
ODA packages 478
oda plug-in 480, 486, 491
ODA plug-ins 488, 491, 493
ODA Runtime driver wizard 490
oda ui dataSource plug-in 478, 479

ODA UI extension 480
ODA UI extension points 506
ODA UI extensions 500–524
ODA UI plug-ins 480, 506, 509
ODA user interface 478, 479, 480
ODA_DEFAULT_CHARSET variable 511
OdaDataSetHandle class 317
OdaDataSourceHandle class 316
odadesignapi.jar 278
odaDriverClass property 141
ODAHOME variable 511
odaPassword property 141
odaScalarDataType property 489
odaURL property 141
odaUser property 141
odaVersion attribute 487
ODBC (defined) 622
OLAP (defined) 622
onCreate events
 bound columns and 119
 report items and 97, 114
 reports and 116
 tables and 122
onCreate method 114, 163
onFetch events 114, 164
onFetch method 113, 114, 138, 164
Online Analytical Processing systems. *See*
 OLAP
online documentation. *See*
 documentation
online help 622
online reports. *See* web pages
onPageBreak events 97, 115, 117, 120, 123
onPageBreak method 115, 163
onPrepare events 97, 114, 116, 123
onPrepare method 114, 116, 162
onRender events
 report content and 151
 report elements and 116
 report items and 97, 115, 124
 tables and 122
onRender method 115, 151, 163
onRowSets method 422
open data access 622
 See also ODA
open database connectivity. *See* ODBC
Open Debug Dialog command 180
open events 113, 114

Open extension point description option 386
open method
 Hibernate drivers and 536
 iterator objects and 226
 scripted data sets and 165, 221, 226
 scripted data sources and 165, 218
 writing event handlers and 112, 113, 114
Open Perspective command 10
Open Services Gateway Initiative. *See*
 OSGi
open source projects xix
 See also projects
open source software development. *See*
 Eclipse
opening
 class files 388
 configuration files 31
 connections 491, 493
 cursors 492
 data sets 113, 164, 165, 226
 data sources 112, 163, 165, 221
 document files 239, 276, 286
 Export Wizard 394
 HTML reports 301
 Navigator 145
 output files 445, 467
 PDF documents 301
Plug-in Development perspective 377
report designs 98, 145, 276, 286, 304
 code example for 286, 305
openReportDesign method 239, 276, 286
openReportDocument method 239, 276, 286
operators 623
option objects 297
options (rendering) 297
orthogonal axis values 332, 589
 See also pie charts
orthogonal series items 334
OSGi Alliance web site 375
OSGi API 375
OSGi class loader 488
OSGi configuration property type 280
OSGi error messages 323
OSGi platform 375, 379
OSGi resource bundles 375, 379
out_msg parameter 228

outer joins 623
 See also joins
outline attribute 409
Outline view (Eclipse) 379, 623
outlining chart legends 332
output
 accessing formatted 301
 changing source code and 367
 creating CSV files and 433, 444, 448, 455
 creating HTML reports and 297
 creating XML files and 466
 generating 275, 300, 433
 HTTP requests and 135
 rendering 89, 116, 124, 223, 400
 setting options for 297
 setting rendering options for 297
 validating 98, 101
 writing to disk 237
output columns 221
Output Columns view 221
output files
 accessing 301
 closing 446, 468
 displaying 455
 method execution calls in 142
 naming 297
 opening 445, 467
output formats
 customizing 293, 400, 434
 customizing report generators for 93
 developing report engine for 109, 433, 462
 generating reports and 120, 275, 276, 297
 getting 131, 170
 handling events for 115
 setting 276, 438
 specifying MIME types for 438
output parameters 227
output streams 143, 275, 297, 301, 444, 467
output types 214, 259
 See also output formats
outputParams array 228
overriding methods 161
overriding report parameters 22
Overview page (PDE Editor) 380, 393
_overwrite parameter 48

P

-p command line argument 22
package (defined) 624
Package Explorer 379
packages 3, 17, 236
Packages object 151
Packages prefix 151
—page parameter 48
page break events 120
page breaks 44, 115
page element 507
page footers 298, 606
page headers 299, 609
page layouts
 See also designs; master pages
 building multicolumn 102
 creating lists and 103
 creating tables and 103
 defined 616
—pagebreakonly parameter 49
pageNum attribute
 report tag 68
 viewer tag 65
—pagerange parameter 49
pageRange attribute
 report tag 68
 viewer tag 65
palette element 408
Palette page. *See* Palette view
Palette view 90, 624
param tag 69, 70
paramDef tag 74, 75
parameter definitions 240, 289
parameter events 110
parameter groups 289, 290
Parameter page (web viewer) 71, 74
PARAMETER_GROUP value 290
—parameterpage parameter 49
parameterPage tag 70, 72, 74
parameters
 See also data set parameters; report parameters
 accessing information about 240
 creating scripted data sets and 227
 defined 624
 defining session-specific 245
 developing ODA extensions and 492
 getting values for 141, 170, 277, 290

parameters (*continued*)
grouping. *See* parameter groups
running reports and 34, 70, 135
setting values for 69, 131, 170, 290, 295
 code example for 291, 292
 validating 240, 241
parameterValues parameter 229
param-value element 32
parent class. *See* superclasses
parse method 291
passwords 141, 624
paths
 context mapping 31
 fonts and 299
 image files and 282
 report design files 30
 report engine and 279
 scripted data sources and 227
pattern parameter 49
pattern attribute
 param tag 70
 paramDef tag 76
 parameterPage tag 73
 viewer tag 65
patterns 624
 See also object-oriented programming
PDE (defined) 601
PDE Manifest Editor 380, 451
PDE Workbench 379–398
 See also Eclipse Plug-in Development Environment
PDF documents
 building charts for 353
 displaying 34
 generating 239, 241, 293
 installing fonts for 33
 missing content in 33
 opening 301
 setting up rendering context for 299
 writing to disk 237
PDF emitter 89
PDF formats 433
PDF reports. *See* PDF documents
PDFChartGenerator charting
 example 353
PDFChartGenerator class 353
PDFRenderContext class 299
PDFRenderContext objects 297
persistent variables 131, 169, 170
perspectives 600
pictures. *See* images
pie charts 321, 327
 See also charts
platform 625
Platform class 283
platform context 281, 283
Platform context property type 280
Platform Plug-in Developer Guide 398
PlatformServletContext objects 282
plot area (charts) 202, 326, 331
Plot interface 326
Plot objects 331
plot properties (charts) 326, 331
plot property 325
Plugin class 375
Plug-in Development Environment 369, 377, 601
 See also PDE Workbench
Plug-in Development perspective 377
plug-in directories 369
plug-in drivers 478
plugin element 373
plug-in extensions
 building 390–392
 creating 381–389
 customizing BIRT and 369
 deploying 370, 393–398
 examples for 398
 selecting export options for 394
 setting class attributes for 388
 specifying 383, 439
 testing 392–393
 viewing descriptions of 385, 386
plug-in fragments 571, 625
plug-in manifest files 373
Plugin package 419, 441
plug-in registry 373, 434
plug-in run-time class 373, 375
Plug-in Selection dialog 381, 406
plugin.xml 370, 372, 373
Plug-in.xml page (PDE Manifest Editor) 381
plug-ins
 accessing CSV data sources and 480, 481
 building CSV report rendering extension and 437

plug-ins (*continued*)

- building rotated label report item extension and 402, 405
- caching conflicts and 12
- calling methods in 153
- compiling code for 392
- creating Hibernate driver 526, 547, 554
- creating Hibernate ODA UI 564
- defined 625
- defining extension points in 399
- deploying applications and 277
- deploying web viewer as 76
- developing 370, 377
- developing ODA extensions for 478, 488, 491, 493
- editing project settings for 380, 381
- extending application development and 369
- extending functionality of 381
- installing 277, 372
- instantiating objects in 373
- integrating with Eclipse 83
- loading 382
- running report engine and 277, 433
- selecting run-time libraries for 391
- setting dependencies for 381, 405
- setting properties for 379
- setting up projects for 377–379
- testing 370, 379, 451
- updating 394, 395
- verifying run-time archive for 383
- viewing information about 380
- viewing project settings for 380

plugins folder 481

plug-ins subdirectory 277

polymorphism 625

populate method 188

portal 625

portlets 626

position attribute

- parameterPage tag 73
- report tag 68
- viewer tag 65

predefined layouts. *See* master pages

predefined values 593

preference example (charts) 353

preferences (Eclipse workspace) 362

PreferenceServlet charting example 353

preparation phase (events) 110, 115, 124

prepare method 495, 539

prepareMetaData method 495

prerelease product updates 15

presentation engine

- components of 86
- customizing output formats for 400, 434
- overview 89

presentation extensions 400

presentation phase (events) 110

presentation plug-in 408, 409

presentation services 89

presentations 421

preview servlet 109

previewer 626

previewing

- reports 109, 237
- scripts 138
- source code 146

primary base axis (charts) 332

Print Report dialog 42

Printer.js 60

printing reports 42, 44, 102

println method 144

PrintWriter objects 143, 144

private styles 310

PrivateStyleHandle class 310

procedures 626

- See also* functions; methods

process 626

Product Configuration dialog 396

product upgrades 14

program requirements

- BIRT Report Designer 5–7

program updates 393, 395

programming environments 108

programming interfaces 166, 612

- See also* application programming interfaces; interfaces

programming languages 613

- See also* scripting languages

programming tools 369

projects

- adding event handler classes to 156, 159
- building BIRT 361–362, 364–366
- building plug-in extension 390, 392
- building update site 396
- changing settings for 380, 381

- projects (*continued*)
creating
 CSV ODA driver 481
 CSV report rendering 434
 Eclipse xix
 fragment 573–575
 Hibernate ODA driver 526, 527
 Hibernate UI plug-in 548
 plug-in 377–379
 report item extension 403
 rotated label report item 403–405
 XML report rendering 463
developing ODA extensions and 479
selecting 157
testing 366
viewing settings for 380
.properties files 13, 35, 170
properties
 accessing 246, 247, 307
 charting applications and 324, 325–328
 charts and 201, 202, 258, 331–332, 338
 customizing 96
 data sets and 172, 316, 317
 data-source connections and 140
 defined 627
 event handlers and 135, 136
 Hibernate data sources and 556, 557, 558
 Java classes and 389
 Java packages and 151
 libraries and 247
 ODA extensions and 486
 plug-ins and 379
 report designs and 102, 247, 286
 report elements and 168, 304, 312
 report emitters and 438, 440, 465
 report engine and 279, 280, 295
 report item extensions and 408
 report items and 103, 136, 307, 309, 311
 ROM elements and 96, 98, 102
 selecting 136
 user sessions and 246
properties list 626
Properties page. *See* Properties view
Properties view 626
 property annotations 387
 property collections 97
 property definitions 96
Property Editor 627
property editors 370, 501
property element 98
property handles 310, 311
property pages 506, 507, 510, 554
Property property 103
property property type 96
property sheets 99
 See also Properties view
property structure objects 311, 312
property types 96, 103
PropertyHandle class 311
property-list property type 96
propertyPage extension point 548
propertyPages extension point 507
protocol 627
prototype.js 57
public classes 158
publish 627
- ## Q
- queries
 See also HQL statements; SQL statements
 accessing Hibernate data sources and 539, 541, 542, 554, 560
 accessing ODA data sources and 34
 accessing web service data sources and 229
 changing 140
 defined 627
 defining chart series and 327, 334, 336
 developing ODA extensions for 478, 491, 493
 executing 496
 extending functionality of 400
 retrieving data with 292
Query class 493, 494
query preparation extensions 400
query strings (request objects) 135
QueryImpl class 333
queryProperties parameter 229
queryText parameter 229
queryText property 140
quotation mark characters. *See* double quotation mark character
- ## R
- range 627

RCP (defined) 601
 See also Rich Client Platforms
readLine method 496
readSchemaFile method 472, 473
record counters 145, 146
records. *See* rows
referencing
 Java classes 151, 226
 report designs 31
registering service objects 375
registry (plug-ins) 373, 434
regular expressions 628
relational databases 478
 See also databases
relative paths 30
release build 7
removing. *See* deleting
renaming report design files 339
render option classes 297
render option objects 241, 297
render package 258, 272, 273
render processes 129, 133
render tasks 109
rendering classes 443, 466
rendering environments 237, 275
rendering extension API 433
rendering extensions
 creating projects for 434, 463
 defined 628
 developing 433, 434, 441, 465
 naming 438
 overview 433
 running 454
 sample for 433, 440
 setting dependencies for 437
 viewing output for 455
rendering option objects 297
rendering options 297, 449
rendering output 89, 116, 124
rendering phase (charts) 191, 192, 193
rendering plug-ins 440
rendering services 237
renderLegendGraphic method 192
renderSeries method 192
RenderTask object 295
RenderTask objects 109, 115
replacing null values 188
_report parameter 49
report components 246, 247

See also report elements; report items
report context objects 130, 131, 132, 169
report descriptions 92, 95
report design elements 102, 166
report design engine 89, 91, 302
report design engine API 235, 244, 248, 275
 See also report model API
report design engine class 245
report design environments. *See* BIRT; Eclipse
report design events 111
report design files
 defined 628
 generating 87, 89, 93, 244
 generating reports from 237, 276, 286, 295
 installing report viewer and 30, 31
 loading 239
 naming 430
 opening 98, 145, 276, 286, 304
 listing for 286, 305
 overview 92
 referencing in URLs 31
 renaming 339
 running 240, 295
 setting location of 31
 validating 89
Report Design perspective 10
report design properties 102
report design tools 87, 244, 245
report designer packages 3
report designer ui extensions
 package 419
report designers 3, 87, 93, 237
 See also BIRT Report Designer; BIRT RCP Report Designer
report designs
 See also page layouts
 accessing 246, 275, 302, 304
 accessing properties for 247
 accessing ROM schema for 98
 adding charts to 339, 351, 352, 355
 adding report items to 305, 315
 changing 132, 279, 302, 305
 creating 244, 303, 319
 defined 597
 defining event handlers for 108, 127, 132, 155, 166, 174

report designs (*continued*)
deploying 302
generating CSV files and 457
getting parameters in 289
initializing 143
retrieving external data for 478
reusing 88, 92
saving 303, 318, 339
setting location of 31
setting properties for 286
standardizing 92
testing for parameters in 289
validating 98, 101
viewing report items in 117

report document files
accessing data in 240
creating 277, 295
defined 628
generating reports from 109, 275, 276, 286, 287
opening 239, 276, 286
overview 92
writing to disk 240

report document model (BIRT). *See* ROM

report documents. *See* documents; reports

report editor 629

Report element 97

report element handles 312

report element interfaces 167–169

report elements
See also specific type
accessing 275, 302
adding 102
applying styles to 97
associating event handlers with 127, 160
changing 131, 304
creating 168, 315
defined 629
defining at run time 168
developing design interface for 167
developing ODA extensions for 487, 488
generating 116
handling events for 116, 131, 155
initializing 115
loading property definitions of 245
operations unique to 248
providing services for 247

scripting for 135, 161
setting properties for 168, 304, 312
validating 89

report emitters. *See* emitters

report engine
accessing external classes and 295
accessing external data sources and 491
cancelling running tasks and 301
configuring 238, 239, 245, 280, 281
configuring HTML emitter for 281
connecting to data sources and 276
creating 238, 239, 245, 279, 283
customizing 237
defined 587
defining default location for 279
defining platform context for 280, 281
deploying as web application 283
deploying report viewer and 27
extending report items and 91
firing events and 115, 116
generating output and 462
generating report items and 400
generating reports and 109
getting output formats for 434
installing 21
overview 89
platform context for 281
services provided by 237
setting global options for 238, 245
setting properties for 279, 280
setting up as stand-alone application 283
setting up as web application 284
shutting down 276
testing installation for 21

report engine API 237, 238, 242, 275

report engine API package 235, 241

Report Engine API Reference 275

report engine classes 239, 241, 245

report engine content package 442

report engine emitters package 438

report engine emitters plug-in 400

report engine extension package 401, 419

report engine hierarchy 86

report engine home directory 277

report engine package 277

report examples (charts) 355

report executable files 629
 See also report object executable files

report file types 91, 606

report files 91–92
 See also specific type

report generation services 237

report generators 93, 235

report item API 401

report item elements 102–103
 See also report items

report item emitter extension point 400

report item events 114, 162, 175

report item extension points 399, 408, 410

report item extension sample plug-in 398, 401, 588

report item extensions

- creating 399, 412
- defined 629
- defining dependencies for 405
- deploying 402, 429, 572
- designing reports and 401
- developing 91, 401–418
- displaying reports and 89
- downloading sample code for 398
- implementing classes and interfaces for 419
- implementing presentation plug-in for 408, 409
- naming 371, 408
- overview 399–401
- setting up projects for 403
- testing 429

XML schema definitions for 370, 386

report item interfaces 419

report item model extension points 399

report item palette 400

report item query extension point 400

report item run-time extension point 400

report item UI elements 408

report item UI extension points 400

report items

- See also* report elements
- accessing 275, 303, 306, 308
- adding 305, 315, 399, 421
- applying styles to 310
- binding data sets to 317, 318
- changing properties for 309, 310, 311
- creating 91, 315, 400
- customizing 91, 94, 401

defined 102, 629

examining 117, 307

extending functionality of 94, 98, 399

generating 117, 400

getting handles to 246, 247

getting properties for 307

handling events for 97, 113, 121, 162, 175

naming 306

overview 90–91

setting properties for 103, 136, 309, 310

sharing data bindings and 120

writing to CSV files 444

report layouts. *See* report designs; page layouts

report level events 173

report library files 92, 303, 616
 See also libraries

report model API 244

report model api extension package 419

report model api package 235, 244, 248, 419

report object model. *See* ROM

report objects. *See* reports

report parameter collections 291

Report Parameter page (web viewer) 73

report parameters

- See also* cascading parameters
- accessing 287, 288
- converting 289
- defined 630
- getting attributes of 290
- getting default values for 290
- getting value of 141, 290
- overriding 22
- running reports and 34
- setting at run time 70, 74
- setting values for 290, 291, 292, 295, 300
- testing for 289, 290
- validating 111, 115, 241

report projects. *See* projects

Report Rendering Extension API. *See* rendering extension API

report rendering sample plug-in 398

report sections 120, 634

report specifications. *See* report designs

report tag 66, 68

report template files 92, 631

report templates 93, 639
Report templates section (New Report) 430
report viewer
 accessing reports for 30, 33
 changing configurations for 31
 components of 86
 default location for 30
 deploying 7, 27
 developing for 237
 generating charts for 90
 generating reports from 276
 installing 28–30
 integrating report item extensions with 429
 mapping to 31
 overview 88
 placing Java classes for 152
 referencing report designs for 31
 running on Apache servers 28, 30, 32
 setting context parameters for 31
 starting 28, 32
 testing installation of 30
report viewer plug-in 88
report viewer servlet 88, 631
ReportComponentIdRegistry.js 60
reportContainer attribute 68
reportContext objects 130, 131, 132, 169, 200
reportDesign attribute
 parameterPage tag 73
 report tag 69
 viewer tag 65
ReportDesign element 102, 166
ReportDesign events 166
ReportDesignHandle class 247, 248, 303, 305
ReportDesignHandle objects 246, 304
reportDocument attribute
 parameterPage tag 73
 report tag 69
 viewer tag 66
ReportElement element 102
ReportElementHandle class 250, 251
ReportEngine class 239, 245, 279, 288
ReportEngine objects 238, 279, 283
 See also report engine
ReportEventAdapter class 166
reporting applications 237
reporting applications. *See* applications reporting platform. *See* BIRT
ReportItem element 102
reportitem plug-in (charts) 401
ReportItemFactory class 421
reportitemGeneration plug-in 400
reportitemGeneration.exsd 400
ReportItemHandle class 253
ReportItemLabelProvider class 421
reportItemLabelUI element 387, 408
reportItemModel plug-in 399, 402, 408, 409
reportItemModel.exsd 399
reportitemPresentation package 409
reportitemPresentation plug-in 400, 402, 408
reportitemPresentation.exsd 400
ReportItemPresentationBase class 421
reportitemQuery plug-in 400
reportitemQuery.exsd 400
reportitemUI plug-in 385, 400, 401, 402, 408
reportitemUI.exsd 370, 400
reportletId attribute
 parameterPage tag 73
 report tag 69
 viewer tag 66
reportlets 48
ReportParameterConverter class 289
reports
 accessing 30, 33, 301
 associating styles with 101, 102
 creating 107, 276, 295
 customizing 279
 defined 628
 deploying 27, 156, 227
 designing. *See* designing reports; designs
 developing 107
 displaying 33, 44, 87, 89
 exporting 42
 generating 93, 116, 276, 279, 300
 previewing 109, 237
 printing 42, 44, 102
 rendering environments for 275
 rendering output for 89, 116, 237
 rendering specific pages for 240
 running 129
 selecting language for 14

reports (*continued*)
 setting default styles for 97
 testing report viewer for 30
 writing to disk 237
request objects 135
RequestorFragment plug-in 50, 52
requests 130, 169, 631
Required Plug-ins section 381, 382
reserved words. *See* keywords
resizing web pages 299
resource bundles 375, 499
resource files 130, 281, 304, 632
Resource files property type 281
resource folder 49
resource keys 632
 `_resourceFolder` parameter 49
resourceFolder attribute
 parameterPage tag 73
 report tag 69
 viewer tag 66
resources 15, 394
response 631
response messages 631
result sets
 See also data sets; queries
 accessing 492
 building charts and 186
 defined 632
 developing ODA extensions for 478, 491, 492, 493
 getting Hibernate data source 538, 542
 getting number of columns in 139, 140
 returning 104
ResultSet class 493, 497, 534, 542
ResultSetMetaData class 493, 498, 534
Rich Client Platforms 76, 87, 601
right outer joins. *See* outer joins
rollback method 537
rollback operations 491, 537
ROM 91, 95, 630
ROM API Reference 275
ROM definition file 630
ROM Definitions Reference 95
ROM element design interfaces 161
ROM element handles 303
ROM element instance interfaces 161
ROM elements
 accessing 302
 applying styles to 97
 as visual components 102
 creating 100
 customizing properties for 96
 customizing XML code for 97
 defined 630
 defining event handlers for 97
 defining executable code for 103
 overview 95, 101, 103
 scripting for 135, 161
 setting properties for 96, 99
 viewing metadata definitions for 98–101
ROM report item elements 102–103
ROM report item extensions 399, 408
ROM schemas 98, 630
ROM slots 97
ROM specification 91, 95
ROM types 96
rom.def 98, 99, 630
rotated label extension points 389
rotated label plug-in 375, 401, 402, 418
rotated label plug-in project 379
rotated label report item extension
 creating projects for 403–405
 defining dependencies for 405
 deploying 402
 developing 401, 412, 418
 downloading BIRT plug-ins for 402, 405
 downloading source code for 402
 implementing 401
 overview 401
 specifying extension points for 407
 viewing property annotations in 387
rotated text items 421, 422
rotated text report item extension. *See*
 rotated label report item extension
rotatedlabel plug-in 375, 401, 402, 418
RotatedLabelCategoryProviderFactory
 class 425
RotatedLabelGeneralPage class 426
RotatedLabelItemFactoryImpl class 421
RotatedLabelPlugin class 375
RotatedLabelPresentationImpl class 421
RotatedLabelUI class 421
rotateImage method 422, 424
rotationAngle property 409
Row element 104
row execution sequence (events) 121

Row instance interface 169
row objects
 See also rows
 fetching data sets and 172
 getting column information from 138
 getting query statements from 140
 populating 164, 165
rows
 accessing columns in 138
 building 119, 121, 173, 224
 defined 632
 getting information for 172
 grouping 123
 incrementing cursors for 498
 iterating through 226, 492
 returning from Java objects 226
 returning result sets and 104
.rptdesign files. *See* report design files
.rptdocument files. *See* report document files
.rptlibrary files. *See* report library files
.rpttemplate files. *See* report template files
 rtl parameter 49
rtl attribute
 parameterPage tag 73
 report tag 69
 viewer tag 66
run 633
Run dialog 454
run method 300
Run mode 451
run report project 452
run servlet 34, 109
run tasks 109
RunAndRenderTask events 117
RunAndRenderTask objects 109, 110, 115, 124, 295
RunFragment plug-in 50
Runnable variable 300
running reports 129
running tasks, cancelling 301
RunReport class 453
RunTask events 117
RunTask objects 109, 115, 124, 295
run-time archives (plug-ins) 370, 383
run-time connections 140
run-time drivers 491
run-time environments 373

Runtime Information section 390
run-time instance (PDE Workbench) 379
run-time libraries 391
Runtime page (PDE Manifest Editor) 380, 383
runtime tag 572
run-time workbench 429

S

sac.jar 278
SalesReport application 356
sample charting applications 18
sample data 203, 330, 337
Sample Data containment reference (charts) 203
sample database
 See also Classic Models sample database
sample extensions 398
sample fragment plug-in 572, 576
sample XML report rendering extension 462, 463, 464, 465, 466, 474
Samples package 23
save method 318
saveAs method 318
savePage method 518, 562
saving
 images 282
 report components 88
 report designs 303, 318, 339
Scalable Vector Graphics. *See* svg attribute
scalar data types 489
scalar parameters 289, 290
scatter charts 321
 See also charts
schema directory 370
schema element 370
schema-aware tools 98
schemas
 BIRT extension points and 438
 custom formats and 400
 defined 633
 Eclipse extensions and 385
 ODA extensions and 478
 presentation extensions and 400
 query extensions and 400
 report item extensions and 408, 412
 report item generation and 400

schemas (*continued*)
report item user interfaces and 400
report object model and 98
report rendering extensions and 434
ROM report items and 399
validating designs and 98
scope 633
script API library 278
Script attribute (charts) 202, 203
script editor 127, 633
script package 258, 273
Script page. *See* script editor
Script tab 127
scriptable external objects 201
scriptable objects 239
scriptapi.jar 156, 278
ScriptCharts class 353
ScriptDataSetHandle class 317
ScriptDataSourceHandle class 316
scripted data set elements 104
scripted data set events 113
scripted data sets
See also data sets
accessing Java classes for 226
accessing ODA data sources for 317
accessing parameters for 227–228
closing 113, 166, 227
creating 217, 220
defining event handlers for 112, 113, 165
defining output columns for 221
initializing 221, 226
opening 113, 165, 226
scripted data source elements 104, 164, 165
scripted data source events 112
Scripted Data Source option 219
scripted data sources
See also data sources
accessing Java objects and 226
closing 112, 165, 218, 221
creating 217, 218, 219
defined 217
defining event handlers for 112, 164
initializing 218
opening 112, 165, 221
tutorial for 219–224
ScriptedDataSet element 104
ScriptedDataSet interface 165
ScriptedDataSource element 104, 164, 165
ScriptedDataSource interface 164
ScriptedDataSourceAdapter class 164, 165
ScriptedDataSrc.rptdesign 219
ScriptHandler class 259
Scripting configuration property type 281
scripting context 240
scripting languages 107, 127, 633
scripting specifications 127
ScriptingJava.html 153
scripts
accessing Java classes and 151, 152, 159
accessing ROM elements and 135
additional references for 153
building charts and 200, 214, 273, 329
concatenating code for 329
creating 151, 153
tutorial for 144–149
creating reports and 107
event execution sequence and 129
grouping data and 120, 123
importing Java packages and 151, 152, 226
overview 107, 127
previewing 138
providing external values for 281
referencing Java classes and 151, 226
returning parameter values and 141
tracking method execution and 142–144
validating report parameters and 111
variables in 128, 133
writing event handlers and 107, 108, 127, 161
ScriptViewer charting example 353
scrollable properties list 137
scrolling attribute
parameterPage tag 73
report tag 69
viewer tag 66
SDK package 633
SDO Runtime component 6
search paths 33
searching for extension point information 385

searching for program updates 14
sections 120, 634
security 93
select 634
SELECT statements. *See* SQL statements
selection formulas. *See* parameters
SelectionAdapter method 556
semantic validators 101
SemanticValidator element 101
_sep parameter 49
sequential files. *See* flat files
serializable objects 170
serialize method 319
series
 See also charts
 adding 332–336
 binding to charts 186
 building queries for 334, 336
 changing properties for 327
 creating 327, 328
 defined 324, 634
 determining type 189
 setting properties for 203, 335
 setting type 258
series definition objects 186, 327
series identifier 188
Series Thickness attribute (charts) 202, 203
series types 327
SeriesDefinition objects 327, 335
SeriesDefinitionImpl interface 335
SeriesDefinitions objects 187
SeriesImpl class 258, 332
SeriesPalette objects 335
server.xml 30
servers, deploying to 27, 28
service applications 375
Service Data Objects component 6
service objects 375
service registry 375
services 110, 247, 369, 375
ServletContext class 282
servlets 45, 635
session handles 304, 305
session parameters 245
SessionFactory objects 544
SessionFactory operations 525, 536
SessionHandle class 246, 319
SessionHandle objects 245, 304, 305
setAbsolute method 310
setActionHandler method 299
setAttribute method 134
setBaseURL method 299
setBIRTHome method 279
setBlock method 202
setChartInstance method 201
setColorByCategory method 213
setConfigurationVariable method 281
setDataSet method 315, 317
setDataSource method 317
setDescription method 202
setDimension method 202, 203, 213
setEmbeddable method 298
setEmbeddedFont method 300
setEmitterConfiguration method 282
setEnabledAgentStyleEngine method 299
setEngineContext method 280, 281
setExtensionProperty method 172
setExternalContext method 201
setFontDirectory method 299
setGlobalVariable method 131, 170
setGridColumnCount method 202
setHorizontal method 214
setHtmlPagination method 298
setHtmlRtLFlag method 298
setHtmlTitle method 298
setImageDirectory method 298
setImageHandler method 282, 298
setInitialProperties method 556, 558
setInteractivity method 202
setLayoutPreference method 299
setLogConfig method 280, 285
setLogger method 201
setMasterPageContent method 298
setName method 306
setOutputMasterPageMargins
 method 298
setOutputType method 214
setPageFooterFloatFlag method 298
setParameterValue method 131, 170, 290, 291, 295
setParameterValues method 295
setPersistentGlobalVariable method 131, 133, 171, 178
setQueryText method 172
setResourcePath method 281
setResultSetMetaData method 518, 563
setSampleData method 203

setScript method 203, 329
setSeriesThickness method 203
setStatusHandler method 281
setSubType method 203
setSupportedImageFormats method 297
setTempDir method 281, 282
setThreadContextClassLoader attribute 488
settings. *See* properties
setType method 203
setULocale method 201
setUnits method 203, 326
setupConfigLocation method 555
setupFileLocation method 513
setURI method 309
setVersion method 203
shared styles 310
SharedStyleHandle class 310
Show extension point description option 385
showNavigationBar attribute
 parameterPage tag 73
 viewer tag 66
showParameterPage attribute
 report tag 69
 viewer tag 66
_showTitle parameter 49
showTitle attribute
 parameterPage tag 74
 viewer tag 66
showToolBar attribute
 parameterPage tag 74
 viewer tag 66
shutdown method 283
simple object access protocol. *See* SOAP
simple properties 307
singleton pattern 624
site.xml 396
slot handles 306, 307, 318
Slot property 103
slots 97, 306, 635
SOAP (defined) 635
Software Development Kit 633
 See also JDK software; SDK package
software interfaces 612
software requirements
 BIRT Report Designer 5–7
Software Updates command 395
sort 635
sort fields. *See* sort keys
sort keys 635
sort order 312
sort-and-group-by fields. *See* group keys
sorting data 90
source archives 390
Source Build section 390
source code
 accessing data sources and 217
 accessing Java 369
 accessing sample 398
 adding event handlers and 108, 124, 127
 changing 367
 checking for errors in 146
 compiling 362, 392
 controlling report creation and 107
 customizing web viewer and 78
 defining executable 103
 deploying applications and 277
 developing applications and 235, 276, 277
 developing Hibernate drivers and 526
 developing ODA extensions and 479, 480
 downloading 364, 402
 generating CSV files and 433, 434, 457
 installing 23–24
 loading 373, 616
 running reports and 129, 135
 setting run-time connections and 141
 tracking method execution in 142, 144
source data. *See* data sources
source extension point 390
source files 158
SQL (defined) 636
SQL data sources 217
SQL databases 20
SQL language 636
SQL statements
 See also queries
 accessing Hibernate data sources and 478
 changing 140
 defined 636
 getting 140, 172
src attribute 283
stable build 7
stand-alone applications 93, 257, 279

stand-alone environments 280
stand-alone report engine 237
STANDALONE variable 323
stand-alone web pages 34
Standard Viewer 636
 See also report viewer
Standard Widget Toolkit 77
start method 445, 467
starting
 BIRT RCP Report Designer 11
 BIRT Report Designer 10, 19, 25
 report viewer 28, 32
startLabel method 469
startRow method 447
startTable method 446
startText method 448, 469
startup method 283
state. *See* instance variables
Statement class 534, 538
statements 492, 636
static constants 151
static variables 636
 See also dynamic variables; variables
Status handling property type 281
STATUS_CANCELLED value 301
STATUS_FAILED value 301
STATUS_NOT_STARTED value 301
STATUS_RUNNING value 301
STATUS_SUCCEEDED value 301
stock charts 321
 See also charts
StockReport application 356
streams. *See* input streams; output streams
String class 152
STRING data type 139
String data type 637
string expressions 637
string properties 310
strings
 concatenating 329
 converting 289
 HTTP requests and 135
 writing to CSV files and 443
Structure element 101
structure property objects 311
structure property type 97
structured content 637
Structured Query Language. *See* SQL
structured report items 307
StructureFactory class 312
StructureHandle class 255
structures 101
stubs 162
style attribute
 paramDef tag 76
 parameterPage tag 74
 report tag 69
 viewer tag 66
style attributes 98, 99, 103, 309
Style class 420
style definitions 101
 See also style attributes
Style element 101
style elements 97
style properties 246
style property 97
style sheets 97, 278, 589
StyleChartViewer application 353
StyleHandle class 310
StyleProcessor charting example 353
StyleProcessor class 353
StyleProperty property 103
styles
 accessing 247
 applying to report items 310
 associating with reports 101, 102
 charts and 353
 creating 97
 defined 637
 generating HTML and 299
 getting 307, 310
 setting attributes for 99, 103, 309
Sub Type attribute (charts) 202, 203
subclasses 637
 See also descendant classes
subreports 638
subroutines. *See* procedures
summary data. *See* aggregate values
Superclass Selection dialog 158
superclasses 158, 638
 See also ancestor classes
supportsMultipleResultSets method 538
__svg parameter 49
svg attribute 66, 69, 74
SvgInteractivityViewer application 352
Swing applications 345, 352
Swing renderer 191

SwingChartViewersSelector
application 354
SwingInteractivityViewer
application 352
SwingLiveChartViewer application 355
SWT applications 352, 353
SWT browsers 77
SWTchartViewerSelector application 355
SwtInteractivityViewer application 352
syntax (programming languages) 638
syntax conventions
(documentation) xxvii

T

table cells. *See* cells
table elements 102, 315, 638
See also tables
table execution sequence (events) 122
table items 103, 117
table of contents 40, 287
table of contents markers 240
tables
See also table elements; table items
adding columns to 222
adding page breaks for 120
building data rows for 121
creating 103
defined 638
generating 122–124
generating HTML content and 299
getting column information for 138
handling events for 119, 120, 122, 123,
 178
tabs 638
See also page
tab-separated values. *See* TSV formats
tabular layouts 103
tabular result sets 104
tag libraries 63
tags 638
See also elements
target attribute 74
task classes 294
task objects 109, 294
task processes 108–110
tasks, cancelling 301
template files 92, 631
templates 88, 93, 275, 303, 639

Temporary file location property
type 281
temporary files 238, 281
Test Suite. *See* BIRT Test Suite
test.rptdesign 30
testCharts.chart 355
testing
 BIRT installations 10, 11, 13
 BIRT projects 366
 chart attributes 265
 charting applications 323
 CSV ODA UI plug-in 519
 CSV report rendering extension 450
 Demo Database installations 20
 Hibernate ODA UI plug-in 566–570
 plug-in extensions 392–393
 plug-in fragments 579
 plug-ins 370, 379, 451
 report engine installations 21
 report item extensions 429
 report viewer installations 30, 32
 source code installation 24
 XML report rendering extension 474,
 475
Testing section (PDE Manifest
Editor) 429, 451
text 307
See also text elements; text items
text elements 401, 639
text file data sources. *See* text files
text files
See also CSV files
 creating rendering extensions for 434
 tracking method execution in 142, 144
text item design interface 167
text items
 rendering as images 422
 rotating 401, 422
text objects. *See* text
text strings. *See* strings
TextItem objects 167
themes 247, 640
See also styles
this object 136
thread context class loader 488
tick 640
tick interval 640
time data type 597
time values 597
__title parameter 49

title attribute
 paramDef tag 76
 parameterPage tag 74
 viewer tag 66
title tag 298
Tomcat manager accounts 32
Tomcat servers 27, 28, 32, 35
 _toolbar parameter 49
toolbars 34, 109, 640
top attribute
 parameterPage tag 74
 report tag 69
 viewer tag 66
top-level report items 102, 117
transactions 491, 537
transient files. *See* temporary files
translations. *See* locales; localization
translators. *See* converters
troubleshooting installation 11–12
TrueType fonts 33
TSV formats 434
tutorials
 creating event handlers 144–149
 creating scripted data sources 219–224
.txt files. *See* text files
type attribute 488
Type attribute (charts) 202, 203
Type elements 103
type package 258, 271
type property 409
types. *See* data types
typographic conventions
 (documentation) xxvi

U

ui extensions package 419
ui plug-in 480
UiPlugin class 509
ULocale objects 201
Unicode encoding 640
Unicode standard 640
Uniform Resource Locators. *See* URLs
Units attribute (charts) 202, 203
universal hyperlinks. *See* hyperlinks
Universal Resource Identifiers. *See* URIs
UNIX platforms 12
unpacking BIRT archives 5, 6, 11
unpaginated HTML formats 295
unsetDimension method 203

unsetGridColumn method 203
unsetSeriesThickness method 203
unsetVersion method 203
unsupported data sources 90
Update Manager 14
Update Site Editor 393
Update Site Map page 397
Update Site Project page 396
update site projects 396
update sites 393, 395, 396
updates 393, 395
updating
 plug-ins 394, 395
 report designers 14, 15
upgrades 14
uploading update sites 396
URIs 307, 309, 641
URL parameters
 charts 353
 web viewer 46
URLClassLoader objects 488
URLs
 accessing reports and 30, 33, 135
 changing context roots for 30
 defined 641
 HTML reports and 299
 image files and 283
 localized reports and 14
 Mozilla Rhino specification and 127
 report design files and 31
 report parameters in 34
 report viewer and 30
user interface extensions 371
user interfaces
 accessing ODA data sources and 90
 adding data sources to 500
 creating chart reports and 91
 creating custom report designer
 and 93
 customizing 290
 HQL queries and 554
 ODA drivers and 478, 479, 480
 replicating parameters and 289
 report design tools and 245
 report item extensions and 370, 400,
 408
 user names 141
 user sessions 245, 246, 304
 user-defined property definitions 96

UserID parameter 141
userProperties array 96
UserProperty objects 96
util package 258, 274

V

validate events 110, 116
validate method 115
validateParameters method 240, 241
validateQueryText method 495
validating report designs 98, 101
validating report output 98, 101
validator classes 101
validator definitions (ROM) 101
value attribute
 param tag 70
 paramDef tag 76
value axes. *See* axes values
value series. *See* data series
values
 See also data
 defined 641
 displaying external 287
 getting parameter 130, 141, 170, 290
 getting property 286
 overriding parameter 22
 retrieving 119, 138
 setting parameter 69, 131, 170, 290, 295
 code sample for 291, 292
 setting property 310
var identifier 128
variables
 See also specific type
 defined 641
 deleting 130, 169
 getting 130, 169
 scripting and 128, 133
 setting 131, 170
verifyQuery method 560
Version attribute (charts) 202, 203
version numbers 369
viewer applications (charts) 352
Viewer charting examples 354
viewer servlet 45
viewer tag 64, 65
viewer. *See* report viewer; web viewer
ViewerServlet application 45, 50
viewing
 charts 89, 191

error messages 146, 149
extension point descriptions 385, 386
HTML pages 34
HTML reports 88
PDF files 34
PDF reports 301
project settings 380
property annotations 387
 reports 33, 44, 87, 89
views 601, 641
viewservlets.jar 367
visitor interface 442
visitor objects 444, 466
visual components 90, 102, 103
 See also report items
visual elements 95, 102

W

.war files
 application deployment and 31
 defined 642
 engine context and 280
 report viewer installations 28, 29, 32
 Tomcat servers and 28
web applications
 accessing report viewer for 30
 configuring BIRT home for 280
 generating reports and 237
 integrating custom report generator
 with 93
web archive files. *See* .war files
web browsers 33, 77, 299, 301, 607
web context objects 78
web pages
 See also HTML reports
 defined 642
 defining toolbars for 109
 displaying 34, 39
 printing from 42, 44
 resizing 299
web servers 27, 642
web services 229, 642
web services data sources 228–231
web sites xix, 393
web standards 643
Web Tools Integration archive 25
Web Tools Platform 7
 See also WTP plug-ins

web viewer

See also web browsers
communicating with 55
configuring 52–55
customizing 63–75, 78
deploying 61, 76
overview 39
Web Viewer application 39, 44, 109
Web Viewer Deployment wizard 61
web viewer fragments 50–52
web viewer plug-in 76
web viewer servlets 45
web viewer toolbar 39
web.xml 28
webapps directory 28
web-based reports 109, 237
See also web applications
WebLogic servers 27
WebSphere servers 27
WebViewer utility class 76
WebViewerExample directory 54
well-formed XML 642
width attribute
 parameterPage tag 74
 report tag 69
 viewer tag 66
windows 136, 620
Windows platforms 12
wizardPageClass property 555
wizards 377, 500, 548
wizards package 510
wizards plug-in 509
workbench projects 107, 379
See also Eclipse workbench; projects
WORKING_FOLDER_ACCESS_ONLY
 parameter 54
workspace. *See* Eclipse workspace
workspace directory 15
World Wide Web Consortium (W3C) 643
wrapping Java objects 217
writer objects 444, 467
WTP plug-ins 7, 25

X

x series items 335, 336
See also data series
x-axis items 258
x-axis values. *See* axes values
xmi.jar 278

XML (defined) 605
XML documents 642
XML elements 95, 97, 370, 602
XML Extension-Point Schema Definition files. *See* XML schema files
XML files 95, 98
XML formats 434
XML manifest files 377, 379
xml package 466
XML PATH language 643
XML report rendering extension 462, 463, 464, 465, 466, 474
XML schema files 370, 466, 472
XML Schema language 98
XML schemas
 BIRT extension points and 438
 creating extensions and 370, 385
 custom formats and 400
 defined 633
 ODA extensions and 478
 overview 370
 presentation extensions and 400
 query extensions and 400
 report item extensions and 408, 412
 report item generation and 400
 report item user interfaces and 400
 report rendering extension and 434
 ROM report items and 399
 validating designs and 98
XML streams 217
XML tags 471
XML writer 440, 441, 465, 467
XMLFileWriter class 466, 471
XMLPlugin class 466
xml-property property type 97
XMLRenderOption class 466, 471
XMLReportEmitter class 466, 468
XMLReportEmitter method 466
XMLSpy utility 98
XMLTags class 466, 471
XPath expressions 643

Y

y series items 333, 334, 335, 336
See also data series
y-axis values. *See* axes values

Z

.zip files. *See* archive files